**Tailwind in Laravel: New Homepage Design**

We will create a website with three static pages:

* Homepage
* About us
* Contact

I asked ChatGPT to generate the HTML/CSS code for this quick demonstration. It did a good job!

## Tailwind: From CDN to Compiled Styles

In the Blade code, you may notice that Tailwind is loaded from CDN:

<head>

<script src="https://cdn.tailwindcss.com"></script>

</head>

This is not the best practice, right? We need to **compile** Tailwind classes locally to include only the ones that we actually use.

If you look at the default welcome.blade.php, there's a Blade if-statement for this:

<head>

<!-- Styles / Scripts -->

@if (file\_exists(public\_path('build/manifest.json')) || file\_exists(public\_path('hot')))

@vite(['resources/css/app.css', 'resources/js/app.js'])

@else

<style>

/\*! tailwindcss v4.0.7 | MIT License | https://tailwindcss.com \*/@layer theme{:root,:host{--font-sans ...

</style>

@endif

</head>

This means that if we have compiled CSS classes (i.e., if you have run npm run dev or npm run build), we load them via [Vite](https://vite.dev/). Otherwise, we just load Tailwind 4 from inline styles, listing only the classes we need for our homepage.

Laravel is pre-configured to work with Vite and Tailwind v4 automatically, so you don't need to install anything for it.

But, if we just copy-paste the same logic to our home.blade.php, it wouldn't work:

**resources/views/home.blade.php**:

<head>

<script src="https://cdn.tailwindcss.com"></script> {{-- [tl! --] --}}

<!-- Styles / Scripts -->

@if (file\_exists(public\_path('build/manifest.json')) || file\_exists(public\_path('hot')))

@vite(['resources/css/app.css', 'resources/js/app.js'])

@else

<style>

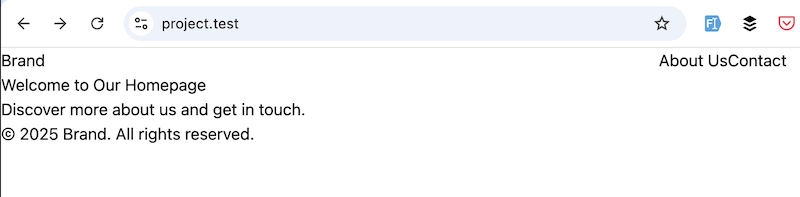
/\*! tailwindcss v4.0.7 | MIT License | https://tailwindcss.com \*/@layer theme{:root,:host{--font-sans ...

</style>

@endif

</head>

If we refresh the homepage, it won't load our styles:



The reason is that we haven't **re-compiled** the styles.

Now, if we run npm run build:



We have our homepage styled again!

## Prefer Bootstrap Over Tailwind?

By default, Laravel homepage comes with Tailwind. But generally, there's no restriction: you can create your Blade file, load Bootstrap from CDN, and create your custom pages.

You can also use any other CSS framework or no framework at all.

In the previous version of this course, for Laravel 11, we had a lesson showing [how to use Bootstrap simple blog theme in Laravel](https://laraveldaily.com/lesson/laravel-beginners/html-blade-design-css-assets), so you can use that as an example reference.

**Navigation and Reusable Main Layout**

Lesson 04/15 · 6 min read

Summary of this lesson:

- Create a main layout with Blade template inheritance  
- Extend layout for multiple pages without code duplication  
- Add Contact and About pages using shared layout  
- Use named routes for flexible navigation

In this lesson, we will add two more pages: "About" and "Contact us".

Of course, we could use File -> Save as in the IDE from home.blade.php and change text where needed, but this would create a lot of **duplicate** code, and changing something in the future would require changing every file.

Instead, we need to create **a main layout** and extend it on every individual page.

Layouts can be created in two ways:

* With [Blade components](https://laravel.com/docs/blade#layouts-using-components)
* Or, with [layouts inheritance](https://laravel.com/docs/blade#layouts-using-template-inheritance).

This is a personal preference, but in this course, we will use the second option, layouts inheritance, mainly to get to the result quicker and to avoid explaining another new term of Blade components. My course goal is usually to help students get to the actual practical result as soon as possible.

## Bonus Tip: Route Names

I want to teach you one good practice about routes: adding **names** to them, in addition to the URLs.

It's a good practice: when using names, if the URL is changed in the future, you won't need to change the URL on every file where it was used. Instead, after changing only in the Routes file, it will change automatically everywhere else.

**routes/web.php**:

Route::view('/', 'home')->name('home');

Route::view('contact', 'contact')->name('contact');

Route::view('about', 'about')->name('about');

In the main layout, where the navigation is, we can use these Route names within the Laravel helper route(), providing route **names** as parameters.

**resources/views/layouts/app.blade.php**:

// ...

<a href="{{ route('home') }}"><h1 class="text-xl font-bold">Brand</h1></a>

<nav>

<ul class="flex space-x-6">

<li><a href="{{ route('about') }}" class="hover:text-blue-500">About Us</a></li>

<li><a href="{{ route('contact') }}" class="hover:text-blue-500">Contact</a></li>

</ul>

</nav>

// ...

# New Design Layout for Blog Project

Lesson 05/15 · 7 min read

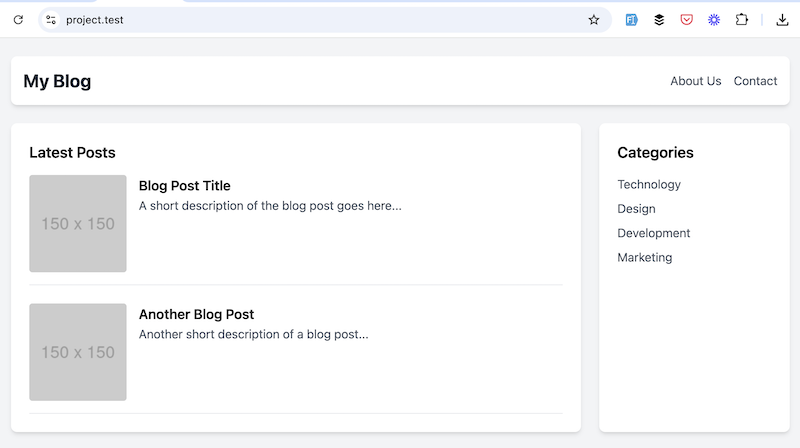
Summary of this lesson:

- Create a new blog layout with header and navigation  
- Design homepage with blog posts and sidebar  
- Implement article page with flexible layout  
- Use `asset()` function for image handling  
- Extend new layout for About and Contact pages

For the next section of this course, we will show the data from the database. We will take a simple Blog with posts and categories as an example.

For that, we need a bit different page layout. Again, I asked ChatGPT to generate something simple with Tailwind, but this time, with the right sidebar.

Here's what our blog will look like.



It's not very fancy, I know, but the visual look is not the point of this Laravel course. You can choose or generate any other design for your projects.

With this design, we will learn a few more things about layouts.

## Blog Layout

Here's the code of our main layout file.

**resources/views/layouts/blog.blade.php**:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Blog Homepage</title>

<script src="https://cdn.tailwindcss.com"></script>

</head>

<body class="bg-gray-100">

<header class="bg-white shadow-md p-4 flex justify-between items-center container mx-auto mt-6 rounded-lg">

<a href="{{ route('home') }}" class="text-2xl font-bold text-gray-800">My Blog</a>

<nav>

<ul class="flex space-x-4">

<li><a href="{{ route('about') }}" class="text-gray-600 hover:text-gray-800">About Us</a></li>

<li><a href="{{ route('contact') }}" class="text-gray-600 hover:text-gray-800">Contact</a></li>

</ul>

</nav>

</header>

@yield('content')

</body>

</html>

## Homepage Design

Then, we have a new homepage extending that new layout with a sidebar.

**resources/views/home.blade.php**:

@extends('layouts.blog')

@section('content')

<main class="container mx-auto mt-6 flex gap-6">

<!-- Blog Posts Section -->

<section class="w-3/4 bg-white p-6 shadow-md rounded-lg">

<h2 class="text-xl font-semibold mb-4">Latest Posts</h2>

<div class="space-y-6">

<article class="flex gap-4 border-b pb-4">

<img src="{{ asset('images/placeholder-150x150.png') }}" alt="Post Image" class="w-32 h-32 object-cover rounded">

<div>

<h3 class="text-lg font-semibold"><a href="#" class="hover:underline">Blog Post Title</a></h3>

<p class="text-gray-600">A short description of the blog post goes here...</p>

</div>

</article>

<article class="flex gap-4 border-b pb-4">

<img src="{{ asset('images/placeholder-150x150.png') }}" alt="Post Image" class="w-32 h-32 object-cover rounded">

<div>

<h3 class="text-lg font-semibold"><a href="#" class="hover:underline">Another Blog Post</a></h3>

<p class="text-gray-600">Another short description of a blog post...</p>

</div>

</article>

</div>

</section>

<!-- Sidebar Section -->

<aside class="w-1/4 bg-white p-6 shadow-md rounded-lg">

<h2 class="text-xl font-semibold mb-4">Categories</h2>

<ul class="space-y-2">

<li><a href="#" class="text-gray-600 hover:text-gray-800">Technology</a></li>

<li><a href="#" class="text-gray-600 hover:text-gray-800">Design</a></li>

<li><a href="#" class="text-gray-600 hover:text-gray-800">Development</a></li>

<li><a href="#" class="text-gray-600 hover:text-gray-800">Marketing</a></li>

</ul>

</aside>

</main>

@endsection

For now, all posts/categories are hard-coded. In the next lesson, we will get them from DB.

You may also have noticed a new function called asset(). This is how you load images into your Blade files.

1. You put the image itself into the /public folder. In my case, it's /public/images/placeholder-150x150.png.
2. Then, to show the image, you call asset() and provide the path relative to that /public folder. In my case, it's images/placeholder-150x150.png.

## Blog Article Page

Our second new page will be for the blog article. It will also extend the new blog layout but without the sidebar.

**resources/views/article.blade.php**:

@extends('layouts.blog')

@section('content')

<main class="container mx-auto mt-6 flex justify-center">

<!-- Blog Article Section -->

<section class="w-3/5 bg-white p-6 shadow-md rounded-lg">

<h1 class="text-2xl font-bold mb-4">Blog Post Title</h1>

<img src="{{ asset('images/placeholder-800x400.png') }}" alt="Post Image" class="w-full object-cover rounded mb-4">

<p class="text-gray-600 mb-4">Published on <span class="font-semibold">March 2, 2025</span></p>

<div class="text-gray-800 space-y-4">

<p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam hendrerit, sapien eget euismod

convallis, augue elit ultrices augue, eget egestas metus ipsum eu odio.</p>

<p>Vestibulum in libero euismod, luctus velit id, sollicitudin lacus. Sed sagittis tristique justo nec

interdum. Fusce auctor purus non massa hendrerit, in vehicula tortor pharetra.</p>

<p>Curabitur sit amet nunc ac odio accumsan malesuada. Proin sit amet magna vitae eros convallis

bibendum ac et libero. Sed scelerisque erat et mauris cursus, eget tincidunt nulla blandit.</p>

</div>

</section>

</main>

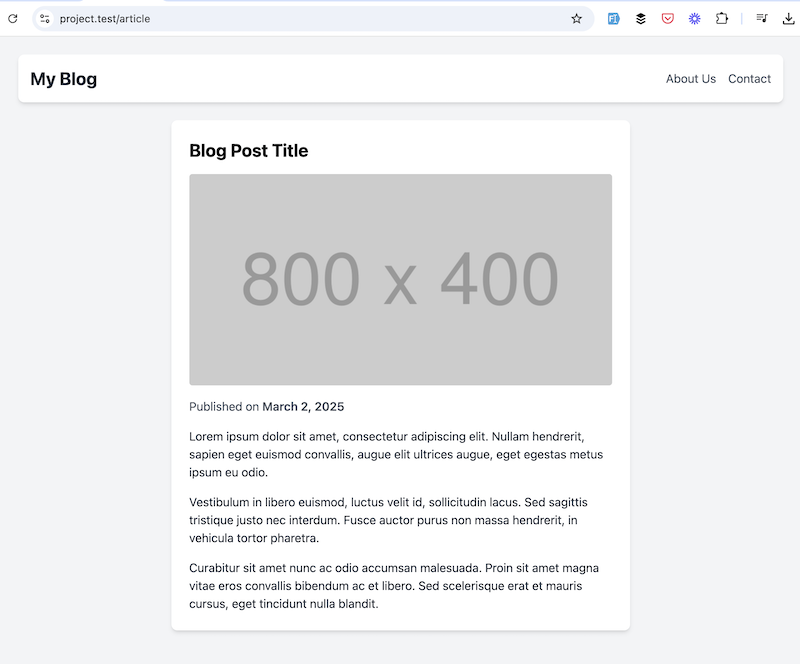
@endsection

We also need a new route for that. For now, that will also be hard-coded as a static article. We will add parameters in the next lesson.

**routes/web.php**:

Route::view('article', 'article')->name('article');

Here's the visual result:



## About/Contact Pages

The final detail: I've also updated the about.blade.php and contact.blade.php to use the new Blog layout.

**resources/views/about.blade.php**

@extends('layouts.blog')

@section('content')

<main class="container mx-auto mt-6 flex justify-center">

<section class="w-3/5 bg-white p-6 shadow-md rounded-lg">

<h1 class="text-2xl font-bold mb-4">About us</h1>

<p class="mt-4 text-lg text-gray-600">We're the best company in the world!</p>

</section>

</main>

@endsection

**resources/views/contact.blade.php**

@extends('layouts.blog')

@section('content')

<main class="container mx-auto mt-6 flex justify-center">

<section class="w-3/5 bg-white p-6 shadow-md rounded-lg">

<h1 class="text-2xl font-bold mb-4">Contact us</h1>

<p class="mt-4 text-lg text-gray-600">Email: support@website.com</p>

</section>

</main>

@endsection

Here's the [GitHub commit](https://github.com/LaravelDaily/Laravel-12-Beginners-Project/commit/5f51c127b96f5773d631445768ce0b8a0951ff7b) for all the changes in this lesson.

# Database Structure and Migrations

Lesson 06/15 · 8 min read

Summary of this lesson:

- Understand Laravel migrations for database schema management  
- Switch from SQLite to MySQL database configuration  
- Create new database tables using Artisan migration commands  
- Learn how to roll back migrations with `down()` method

We have built static pages. Now, let's talk about the backend, database structure and data, and how Laravel deals with that.

You don't need to manually create DB tables with columns. For that, Laravel uses a concept called [migrations](https://laravel.com/docs/migrations). You describe your migration schema in a migration file and then run the migrations, creating tables with columns.

The migrations are added in the database/migrations folder. When you create a new Laravel project, there are three migration files. The only important Migration file when learning Laravel is the first one where the Users table is created. Usually, one Migration file would create one table. The framework creates three tables in one migration file, all corresponding to users.

**database/migrations/0001\_01\_01\_000000\_create\_users\_table.php**:

public function up(): void

{

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

$table->id();

$table->string('name');

$table->string('email')->unique();

$table->timestamp('email\_verified\_at')->nullable();

$table->string('password');

$table->rememberToken();

$table->timestamps();

});

Schema::create('password\_reset\_tokens', function (Blueprint $table) {

$table->string('email')->primary();

$table->string('token');

$table->timestamp('created\_at')->nullable();

});

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

$table->string('id')->primary();

$table->foreignId('user\_id')->nullable()->index();

$table->string('ip\_address', 45)->nullable();

$table->text('user\_agent')->nullable();

$table->longText('payload');

$table->integer('last\_activity')->index();

});

}

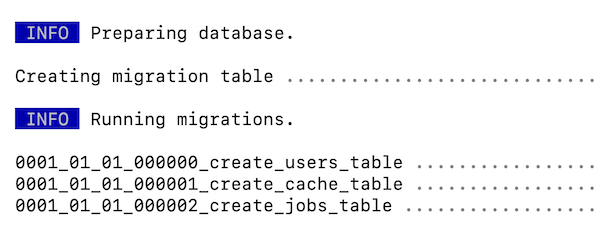
They are pretty readable, even if you don't know how Migrations work. First, you define a table name and all the columns inside a closure.

Let's see what fields are inside the users table:

* id field, which is automatically a primary key and auto-incremented
* String for name and email fields. Also, email is unique, so it cannot be repeated.
* The rememberToken() is a Laravel-specific field for checking the "Remember me" checkbox.
* The timestamps() is a shortcut for two fields: created\_at and updated\_at. They are filled in automatically. We will see that later.

## Switch to MySQL and Execute Migrations

If you use the default database driver SQLite, migrations are run automatically after creating a new project, as you saw in the first lesson about installation:



But let's try to use **MySQL** and create a fresh database.

I've created an empty MySQL DB manually in an SQL client called TablePlus:



How do we connect to it from Laravel?

DB configuration options are stored in the config/database.php file. We can see the connections key with different available DB drivers out-of-the-box: SQLite, MySQL, MariaDB, PostgreSQL, and SQL Server.

**config/database.php**:

return [

'default' => env('DB\_CONNECTION', 'sqlite'),

'connections' => [

'sqlite' => [

'driver' => 'sqlite',

'url' => env('DB\_URL'),

'database' => env('DB\_DATABASE', database\_path('database.sqlite')),

// ... other settings

],

'mysql' => [

'driver' => 'mysql',

'url' => env('DB\_URL'),

'host' => env('DB\_HOST', '127.0.0.1'),

'port' => env('DB\_PORT', '3306'),

'database' => env('DB\_DATABASE', 'laravel'),

'username' => env('DB\_USERNAME', 'root'),

'password' => env('DB\_PASSWORD', ''),

// ... other settings

],

'mariadb' => [

'driver' => 'mariadb',

// ... other settings

],

'pgsql' => [

'driver' => 'pgsql',

// ... other settings

],

'sqlsrv' => [

'driver' => 'sqlsrv',

// ... other settings

],

],

];

Pay attention to this config value that defines the default connection:

'default' => env('DB\_CONNECTION', 'sqlite'),

This env() helper function takes values from the .env file, which are called environment variables. If the .env value isn't set, the second parameter is the default value, which is sqlite in our case.

This is our current .env file for SQLite, generated after Laravel installation:

**.env**

DB\_CONNECTION=sqlite

# DB\_HOST=127.0.0.1

# DB\_PORT=3306

# DB\_DATABASE=laravel

# DB\_USERNAME=root

# DB\_PASSWORD=

So, only the connection is specified as sqlite, and all other values are commented out, as they are not needed for SQLite.

If we want to use MySQL instead of SQLite, we should first change the DB\_CONNECTION value in the .env to mysql.

Then, uncomment the DB\_xxxxx keys and set their values to your DB server details. In my case, locally, it's this:

**.env**:

DB\_CONNECTION=mysql

DB\_HOST=127.0.0.1

DB\_PORT=3306

DB\_DATABASE=project

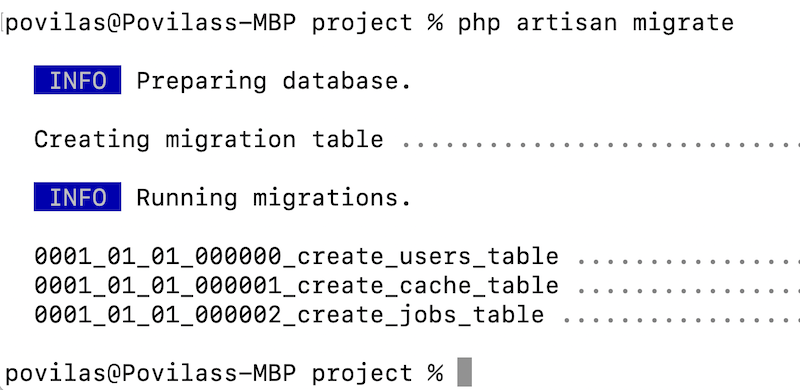
DB\_USERNAME=root

DB\_PASSWORD=

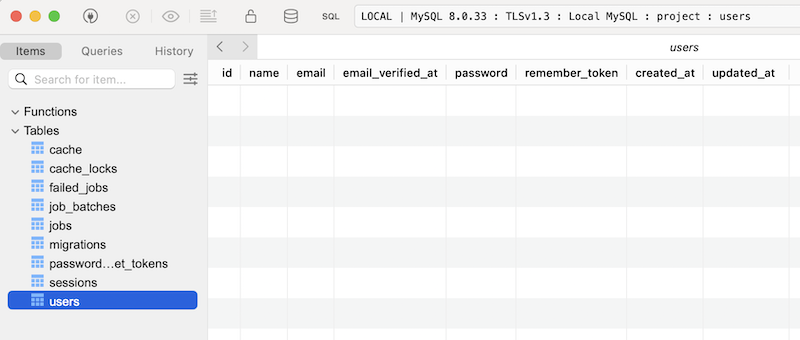
**Warning**: this .env file should **NOT** be pushed to the GitHub repository or published anywhere on the server. It's in .gitignore in the default Laravel. Every environment (local, staging, production) should have its own .env file with different credentials, which are not visible to the public.

Now, Laravel should be working with our MySQL database, which is called project. Let's create DB tables in it from Migration files.

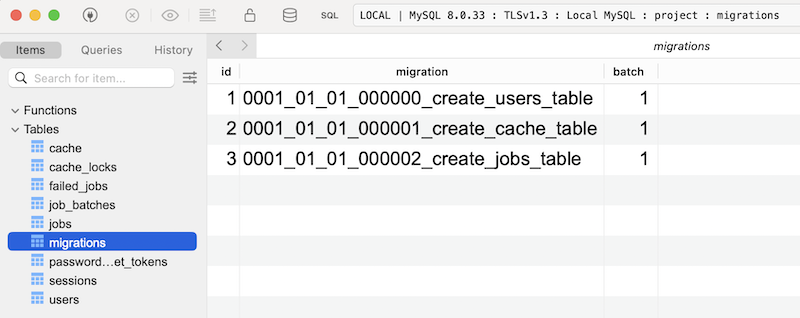
To run migrations, use the artisan command php artisan migrate in the Terminal. This command will run all the migrations from the database/migrations folder.



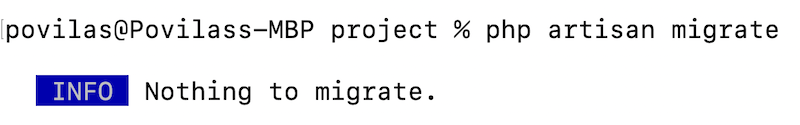
In the DB client, we can now see all the tables created.



Also, there's a special DB table called migrations, which stores the information about which migrations have already been executed:



So, if we run php artisan migrate again now, we will get a message "Nothing to migrate":



That's because Laravel checks the migrations DB table and tries to execute only **new** migration files that were not previously executed.

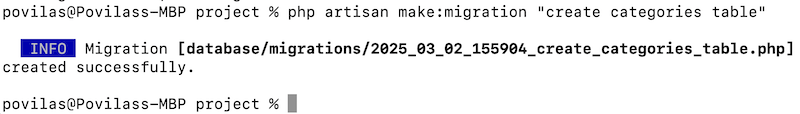
Ok, great, so we've migrated from SQLite to MySQL with all the default Laravel DB tables. Now, it is time to create a new custom DB table.

## Create a New Migration File

In our small project, we will work with blog categories and posts.

In this lesson, let's create a Migration for the DB table categories. We can do it using an artisan command php artisan make:migration. Then, as a parameter, we specify "create [table name] table" in quotation marks.

php artisan make:migration "create categories table"



You may also see a few other syntax options, like separating the actions with the underscore symbol instead of words with quotes:

php artisan make:migration create\_categories\_table

The command creates a new Migration class inside the database/migrations folder with two default columns: id() and timestamps().

**database/migrations/xxx\_create\_categories\_table.php**:

public function up(): void

{

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

$table->id();

// ... Your custom fields should go here

$table->timestamps();

});

}

For example, a category will have only a name, so we use a string name as a column.

**database/migrations/xxx\_create\_categories\_table.php**:

public function up(): void

{

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

$table->id();

$table->string('name');

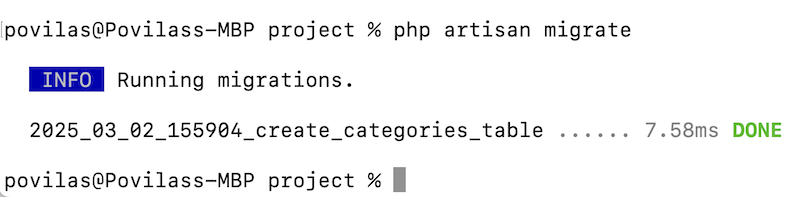
$table->timestamps();

});

}

You can check the [official Laravel docs](https://laravel.com/docs/migrations#columns) for all possible column types and syntax options.

Let's create that categories table in our MySQL DB by running php artisan migrate:



Great, our table is in the DB!

Here's the [GitHub commit](https://github.com/LaravelDaily/Laravel-12-Beginners-Project/commit/1a2b920b0898165973a036b3b4dd91e5cd721a34) for this change.

## "Undo": Rolling Back Migrations

Every migration class has two methods: up() and down().

**database/migrations/xxx\_create\_categories\_table.php**:

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

return new class extends Migration

{

/\*\*

\* Run the migrations.

\*/

public function up(): void

{

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

$table->id();

$table->string('name');

$table->timestamps();

});

}

/\*\*

\* Reverse the migrations.

\*/

public function down(): void

{

Schema::dropIfExists('categories');

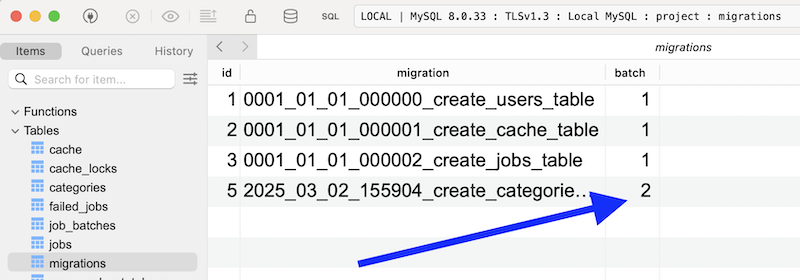
}

};

The down() method can be run to **roll back** the migrations by executing php artisan migrate:rollback. How does that command work?

In the migrations DB table, the executed migrations are saved with their **batch number**.

Whenever we run php artisan migrate, new records are inserted for all the executed migrations with a **new** auto-incremented batch number. In the case of categories DB table migration, we have a batch number **2** for it.



So, the php artisan migrate:rollback command will only roll back migrations **of the last batch**.

In our case, it would only execute the down() method in the migration for the categories table, as the last batch had number 2. So, it would drop the table categories and remove its batch records from the migrations DB table like they never existed.

If you're following along and executing the commands, don't run migrate:rollback on your server. I just wanted to explain how it works. Instead, we will move forward and create more tables in the next lesson.

## Migration System Is Too Complex?

This is a personal note. I've seen some people comment that such a migration system is too complex. Instead, you could create tables **manually** with an SQL client like TablePlus or phpMyAdmin, right?

But what about when you need to deploy to production? Or if you work in a team and need to share the schema of the tables.

Exporting/importing the SQL is very inconvenient. So, migrations help transfer all the DB schema changes and check their history quickly across all environments: local, staging, and production.

So, we've created DB tables. In the next lesson, we will learn how to get data from them with Eloquent ORM and how to show it on the page with Controllers.

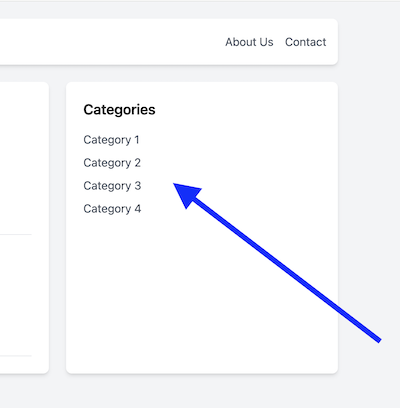
# MVC, DB Queries, and Eloquent Models

Lesson 07/15 · 8 min read

Summary of this lesson:

- Understand MVC architecture in Laravel  
- Create Controllers to handle database retrieval  
- Use Query Builder and Eloquent Models  
- Learn routing between Controllers and Views

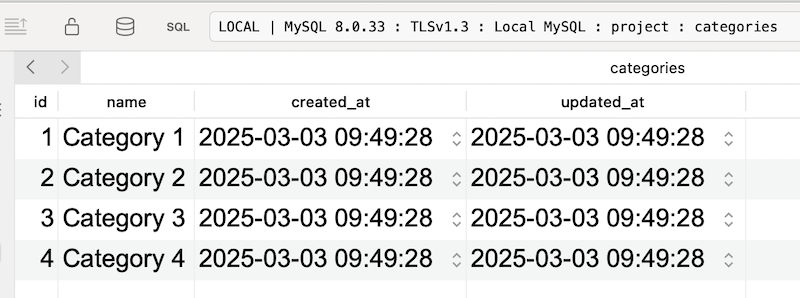
Now that we have created Migrations and have a table in the DB, let's show real categories on the sidebar.



In this lesson, we will learn about **MVC** (Model View Controller): the three layers responsible for viewing data, getting data, and routing the inputs.

## Getting Categories From DB

I manually created these four categories in the categories DB table. Later in this course, we will create a panel to manage them.



To get them from the DB, you have two options:

* Use DB facade class with method DB::table() and construct custom query with so-called **Query Builder**
* Or, create a so-called **Eloquent Model** that would work with specific DB table

Both options are viable, depending on your situation.

I will show you the Query Builder for now, and later, we'll get to Eloquent.

To get all categories from the DB, you need to run this:

// Include this in the "use" section on top

use Illuminate\Support\Facades\DB;

// ...

$categories = DB::table('categories')->get();

Now, the question is, **where** do we run this code, and how do we pass that $categories variable to be available in the home.blade.php Blade View?

## Logic in Route Callback Functions

For now, in the Routes, we used only Route::get() with a callback function or Route::view() to show a static Blade page:

**routes/web.php**:

Route::view('/', 'home')->name('home');

It's time to introduce more logic to get the categories from the database.

The first option is to do it as a part of the callback function:

**routes/web.php**:

use Illuminate\Support\Facades\Route;

use Illuminate\Support\Facades\DB;

// Route::view('/', 'home')->name('home');

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

$categories = DB::table('categories')->get();

return view('home', ['categories' => $categories]);

})->name('home');

As you can see, the function view() accepts the **second** parameter: the array of key-value pairs that would be available in that Blade view.

Here's the [GitHub commit](https://github.com/LaravelDaily/Laravel-12-Beginners-Project/commit/4d95f5abc22284819e0e9e7fcf8e762072ff0fe6) for this change.

However, the logic is typically more complicated in real projects. So, if we add it all to the Routes file, it would become huge.

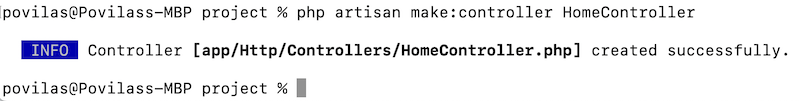
That's why the logic is placed in separate files called Controllers.

## Introducing Controllers

If you want **dynamic** data, you need to introduce a [Controller](https://laravel.com/docs/controllers), which would get that dynamic data and pass it to the View.

Here, we have another example of Artisan command. Get used to them in Laravel. The Controller also can be created using an Artisan command:

php artisan make:controller HomeController



Controllers are created in the app/Http/Controllers folder.

Here's the default empty Controller with no methods:

**app/Http/Controllers/HomeController.php**:

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller

{

//

}

Now, for every Route you want to process, you will create one public method in the Controller.

Let's create the index() method in the Controller, which will get the DB data and return the View with it.

**app/Http/Controllers/HomeController.php**:

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;

class HomeController extends Controller

{

public function index()

{

$categories = DB::table('categories')->get();

return view('home', ['categories' => $categories]);

}

}

In this case, data will be accessible as a $categories variable in the home.blade.php View file.

We can have a @foreach loop in the View file to show these categories. In the View files, many Blade directives start with an @ sign, which we saw previously in the welcome.blade.php.

**resources/views/home.blade.php** - BEFORE:

<!-- Sidebar Section -->

<aside class="w-1/4 bg-white p-6 shadow-md rounded-lg">

<h2 class="text-xl font-semibold mb-4">Categories</h2>

<ul class="space-y-2">

<li><a href="#" class="text-gray-600 hover:text-gray-800">Technology</a></li>

<li><a href="#" class="text-gray-600 hover:text-gray-800">Design</a></li>

<li><a href="#" class="text-gray-600 hover:text-gray-800">Development</a></li>

<li><a href="#" class="text-gray-600 hover:text-gray-800">Marketing</a></li>

</ul>

</aside>

**resources/views/home.blade.php** - AFTER:

<!-- Sidebar Section -->

<aside class="w-1/4 bg-white p-6 shadow-md rounded-lg">

<h2 class="text-xl font-semibold mb-4">Categories</h2>

<ul class="space-y-2">

@foreach($categories as $category)

<li>

<a href="/?category\_id={{ $category->id }}"

class="text-gray-600 hover:text-gray-800">

{{ $category->name }}

</a>

</li>

@endforeach

</ul>

</aside>

The [@foreach()](https://laravel.com/docs/blade#loops) Blade directive is identical to the PHP foreach syntax. We have already seen the syntax with curly brackets to show variables: {{ $variable }}.

So, we have a View file to show data and a Controller to pass the data. The final piece to the puzzle is the Route.

## Route To Controller

Instead of a closure function, we will use an array parameter.

The first key will be the path to the Controller, and the second will be the method in that Controller.

**routes/web.php**:

use Illuminate\Support\Facades\DB;

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

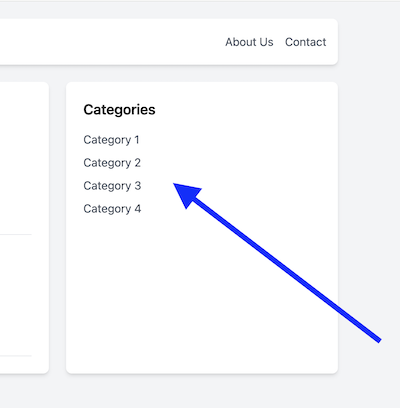
$categories = DB::table('categories')->get();

return view('home', ['categories' => $categories]);

})->name('home');

Route::get('/', [\App\Http\Controllers\HomeController::class, 'index'])->name('home');

After refreshing the home page, we see our new four categories from the DB.



It works!

One more thing to polish. See that long path name to the HomeController? We can avoid having such long paths in each Route if we load them on top in the use section.

**routes/web.php**:

// Usually, all Controllers go on top in the `use`

use App\Http\Controllers\HomeController;

// ...

// And then the Routes contain only the Class name

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

->name('home');

So, this is how the MVC works with Routing, Controller, and the View.

Here's the [GitHub commit](https://github.com/LaravelDaily/Laravel-12-Beginners-Project/commit/75783b8e09e4568165ef32b47d6bc4d403504075) for this section of the lesson.

## Eloquent Models to Work with DB Tables

Earlier, I mentioned the DB::table() alternative, called **Eloquent Model**.

[Eloquent](https://laravel.com/docs/eloquent) is a Laravel ORM engine to work with DB tables. It's a more appropriate and common way to deal with DB data.

It has many features and methods, so using Eloquent instead of a Query Builder is more convenient. To create a Model, run an Artisan command.

php artisan make:model Category



A Model name is typically a **database table name in a singular form**.

All model classes are created in the app/Models category, and they extend a Model class from Eloquent:

**app/Models/Category.php**:

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Category extends Model

{

//

}

It looks empty, doesn't it? But it's already a fully functioning Model.

By default, if you use Laravel naming, the Category Model will know to use the categories table as a plural form of the word "category". If you have another name, you can specify the table name in the Model with the protected $table property.

**app/Models/Category.php**:

class Category extends Model

{

protected $table = 'categories';

}

In the Controller, we can use the Category Model and call the all() method to fetch all records from the categories table.

**app/Http/Controllers/HomeController.php**:

use App\Models\Category;

class HomeController extends Controller

{

public function index()

{

$categories = DB::table('categories')->get();

$categories = Category::all();

return view('home', ['categories' => $categories]);

}

}

And that's it. We don't need to specify any tables. After refreshing the home page, we should see the same four categories in the sidebar.

We will talk about Eloquent Models more in this course. This was just an introduction with the most simple example.

Here's the [GitHub commit](https://github.com/LaravelDaily/Laravel-12-Beginners-Project/commit/b2f963fb59c231c6895320b3a28bf3c6e23f16aa) for this section.

## Summary

This lesson explains one of the most essential concepts to understand in Laravel: the whole workflow and request lifecycle of MVC: Route, Controller, Model, and View.

So, routes have a Controller and a method. In the Controller's method, you get the data and pass it to the View, which then displays it.

In the next lesson, we will work on getting Posts by Category from the URL parameter ?category\_id=XXXXX.