Pizza Ordering System

Built with Laravel 13

A Beginner's Step-by-Step Tutorial

PHP 8.3 - MySQL - Eloquent - Blade - Bootstrap 5

Breeze Auth - Live Updates - Git & GitHub - 2026 Edition

Build a complete Laravel application from scratch

Customer registration and login using Laravel Breeze.
Admin login with role-based middleware (Gate / Policy).
Eloquent models, migrations, and seeders for the menu.
Blade components for clean, reusable UI.
Customer and admin dashboards with live updates (5s polling).
Form Requests, validation, and CSRF protection out of the box.
Tailwind/Bootstrap 5 hybrid styling - minimum custom CSS.
Version control with Git and a GitHub portfolio repo.

egotechworld.com

Free coding tutorials, project source code, and AI tools

Beginner-Friendly - Step by Step - 2026 Edition

Table of Contents

Chapter 1 - Why Laravel and What You Will Build

Welcome! In this tutorial you will build a complete **Pizza Ordering System** using **Laravel 13**, the most popular PHP framework in 2026. By the end you will have a working application where customers register, log in, browse a menu of two pizzas, place orders, and watch the status update *live* on their dashboard. An admin can sign in to a separate dashboard, see new orders as they come in, and approve them with one click.

Why Laravel?

Plain PHP works for small projects, but as soon as your app grows you start writing the same code over and over - login forms, password hashing, form validation, database queries. Laravel gives you all of that **for free**, written by experts, tested by millions of developers.

- **Eloquent ORM** - work with databases as PHP objects, no SQL needed for most queries.
- **Blade templates** - clean, fast HTML templates with reusable components.
- **Artisan CLI** - generate models, controllers, and migrations with one command.
- **Built-in auth** - registration, login, password reset, all in 60 seconds via Breeze.
- **Migrations** - version control for your database schema.
- **Validation** - one line of code blocks bad input.
- **Middleware** - protect routes (admin only, logged-in only) in three lines.
- **Massive community** - the answer to almost any question is on Google or Laracasts.

> **TIP**
>
> If you learnt plain PHP first (like in our PHP version of this tutorial at egotechworld.com), you will fall in love with Laravel. Everything you struggled to write yourself is one Artisan command away.

Who this tutorial is for

- You know basic PHP and HTML/CSS but have never used a framework.
- You have heard of Laravel and want a real project to learn it on.
- You want a clean, modern app you can put on GitHub for your portfolio.
- You want to understand **why** we do each step, not just copy code.

What the finished system does

- Two pizzas on the menu (Margherita, Pepperoni) at different prices.
- Customer registration and login powered by **Laravel Breeze**.
- Each customer has a private dashboard showing their order history.
- Admin logs in through the same form but is redirected to a different dashboard.
- Admin clicks **Approve** and the customer's screen updates within 5 seconds.
- Bootstrap 5 for layout, Blade components for reusable UI pieces.

What you will learn along the way

- How to install Laravel 13 with **Composer** and run it with Artisan.
- **MVC architecture** - what models, views, and controllers actually do.
- How to design a database with **migrations** and seed it with sample data.
- How to use **Eloquent** relationships (User has many Orders, Order belongs to Pizza).
- How to build forms with **Form Requests** for validation in one place.
- How to protect admin pages with **middleware**.
- How to use **AJAX polling** with Laravel API routes for live updates.
- How to push your project to GitHub and deploy it.

> **TIP**
>
> This tutorial is hands-on. Open your terminal and code editor while you read. Type the code yourself instead of copy-pasting - your fingers learn faster than your eyes.

What the finished app looks like

Before we touch any code, here is a preview of every screen you will build. Keep these pictures in mind as you work through each chapter - they are your destination.

1. Home page

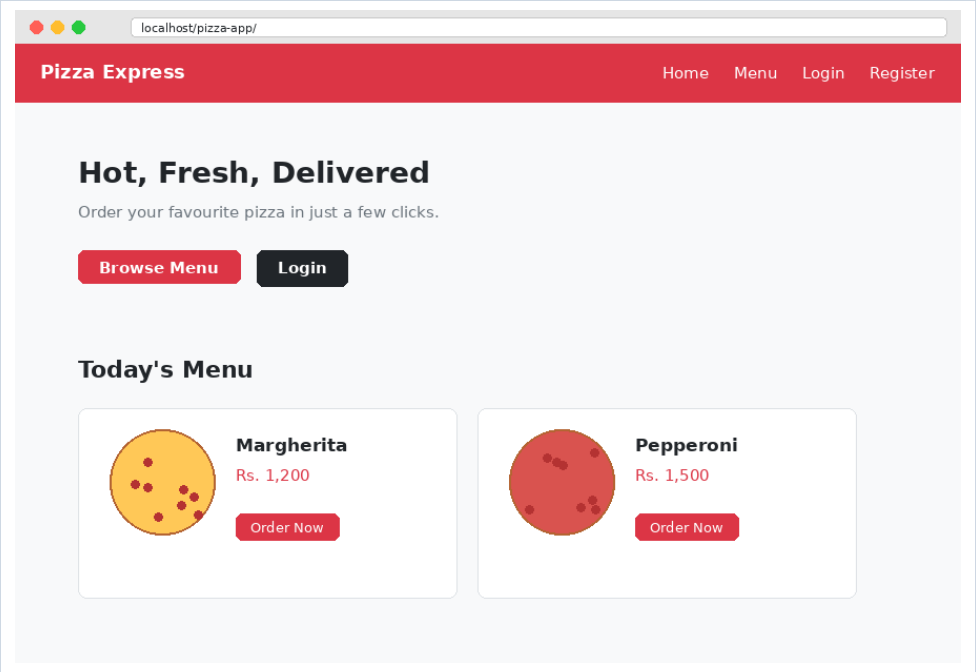


Figure 1 - Public home page. Anyone can visit this URL.

2. Customer login

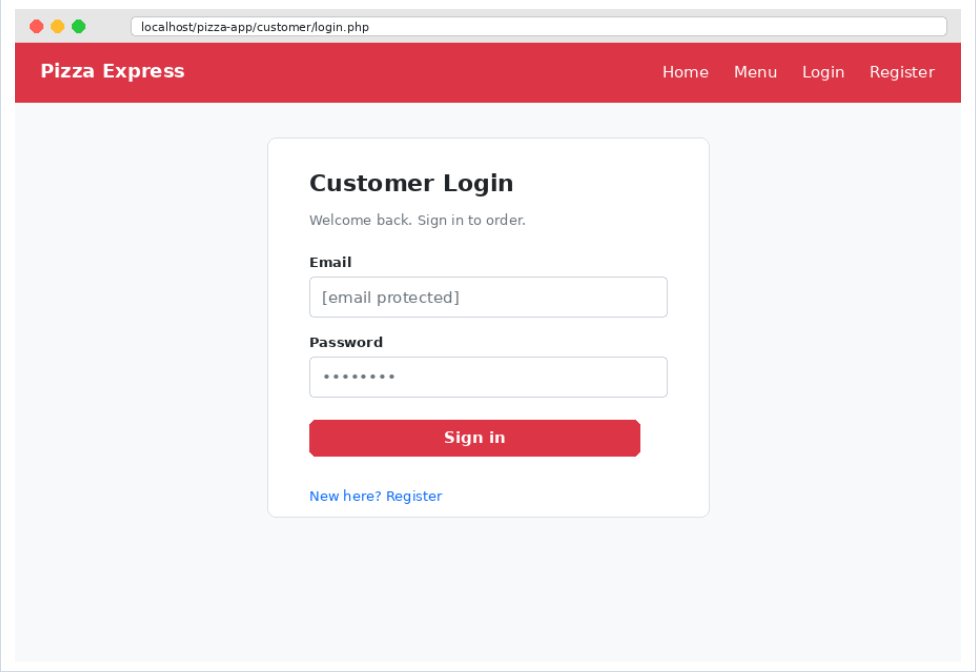


Figure 2 - Login form. Laravel Breeze gives us this for free.

3. Menu page (logged-in customer)

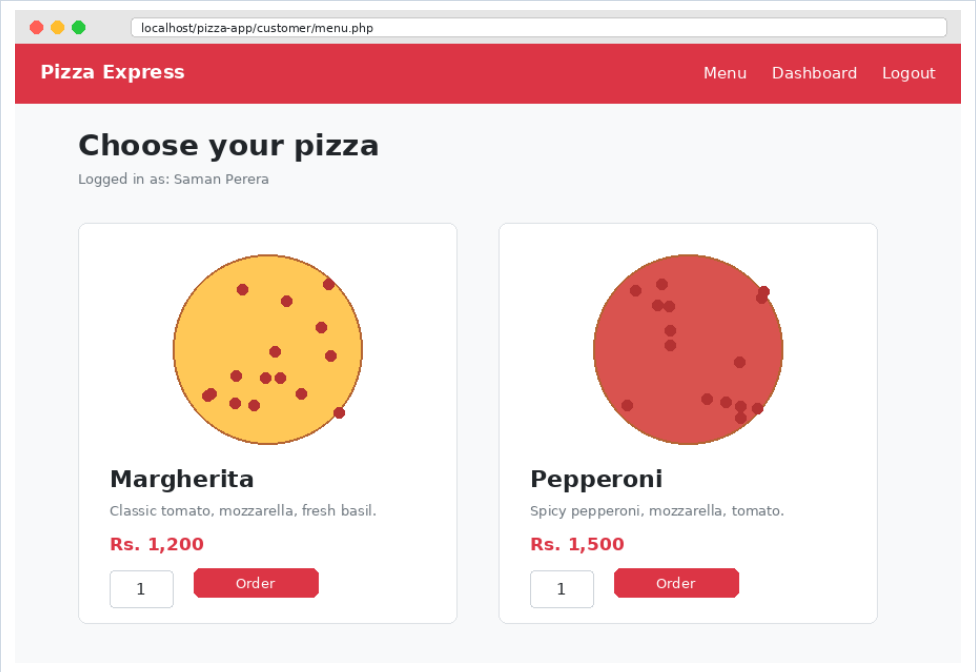


Figure 3 - Menu rendered from the database via the Pizza Eloquent model.

4. Customer dashboard with live updates

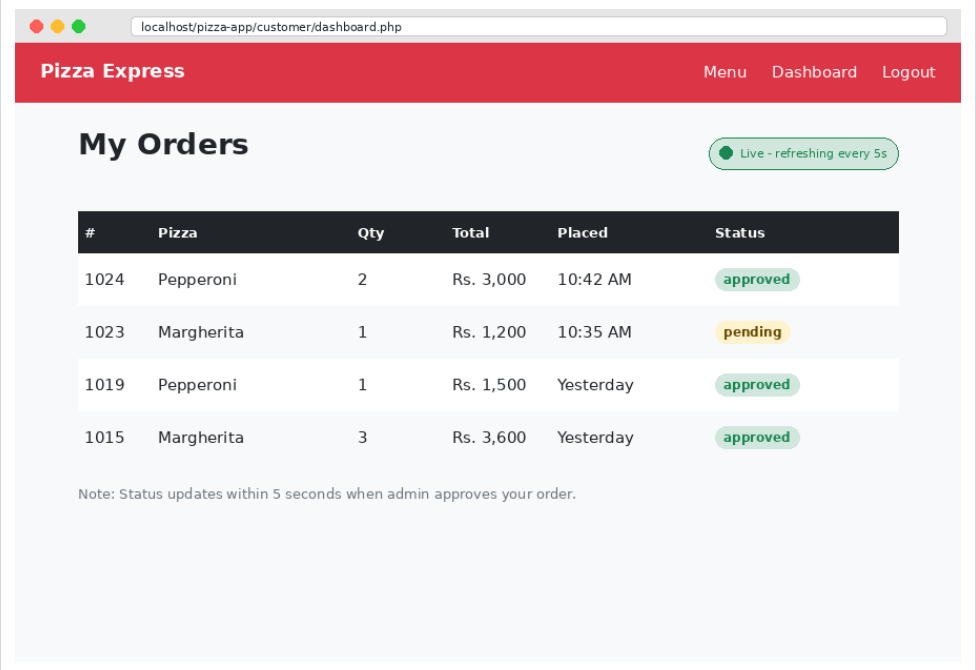


Figure 4 - Customer's order history. Status badges flip from yellow 'pending' to green 'approved' within 5 seconds, no page reload.

5. Admin login

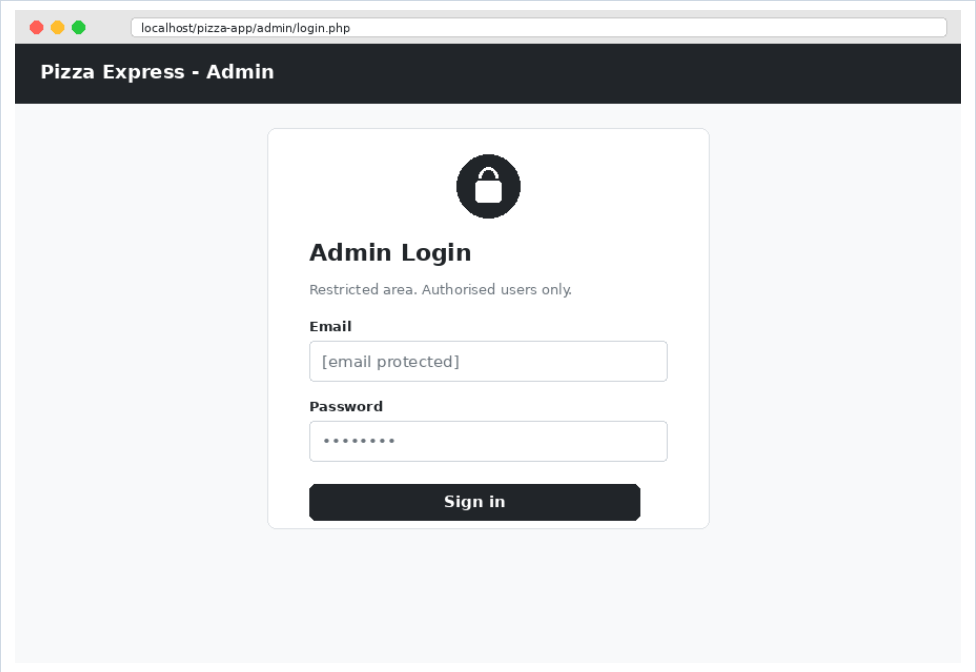


Figure 5 - Admin login. Same Breeze form, different post-login redirect.

6. Admin dashboard

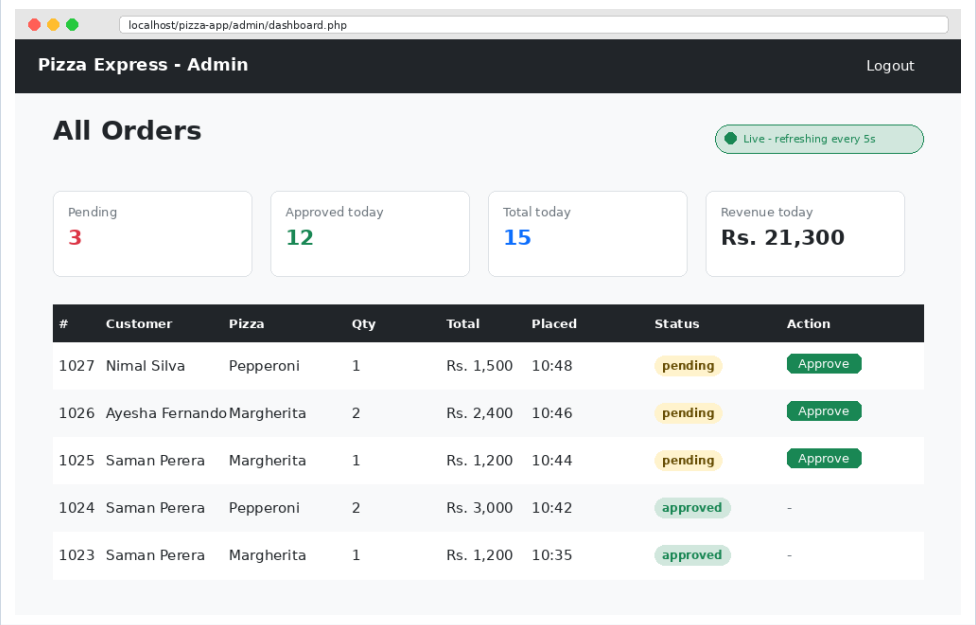


Figure 6 - Admin sees every order. The Approve button fires off an Eloquent update; the customer's dashboard catches it on the next poll.

> **NOTE**
>
> All source code from this tutorial will be made available at **egotechworld.com** in the projects section. If you get stuck, compare your code to the reference there.

Chapter 2 - How Laravel Apps Are Organised (MVC + Routes)

Laravel follows the **MVC** pattern - **M**odel, **V**iew, **C**ontroller. Each piece has one clear job. Once you understand this split, every Laravel project will start looking like a familiar house instead of a maze.

The three building blocks

Piece	Job	Lives in
Model	Talks to the database. One model per table.	app/Models/
View	The HTML the user sees. Built with Blade templates.	resources/views/
Controller	Glue between user requests and models/views.	app/Http/Controllers/

A simple analogy - the restaurant kitchen

Imagine a restaurant. A customer walks in and orders a pizza. The **waiter** takes the order, walks to the kitchen, asks the **chef** for the pizza, and brings it back to the table.

- The customer is the **browser**.
- The waiter is the **Controller** - takes the request, asks for what's needed, returns the result.
- The chef is the **Model** - does the real work (in our case, fetches data).
- The plate of pizza is the **View** - the formatted output the customer sees.

The waiter does not cook. The chef does not greet customers. The plate does not take orders. Each one has a single job. That is MVC in plain English, and it is why Laravel projects stay maintainable even when they grow to hundreds of files.

Plus one more thing: Routes

Before MVC even kicks in, Laravel needs to know which URL goes where. That mapping lives in **routes/web.php**. Think of it as a switchboard, or the menu at the front of our restaurant: a request comes in for /menu, the route file says "that goes to the MenuController", and Laravel hands the request over.

The full request flow (memorise this)

1. User types a URL or clicks a link in the browser.
2. Laravel checks **routes/web.php** to find a matching route.
3. The route hands the request to a **Controller** method.
4. The controller asks a **Model** for data (e.g. all pizzas).
5. The model runs a database query and returns the result.
6. The controller passes that data to a **View** (Blade file).
7. The view renders HTML and sends it back to the browser.
8. The browser displays the page.

Every page in every Laravel app you ever build follows this flow. Once you can recognise these eight steps in any project, you have internalised the framework.

A tiny example of the flow

Let us see what each piece looks like for a single URL: **/menu**. Read each file and notice how short each one is - the magic is in the separation, not in any single piece being clever.

```
// routes/web.php
Route::get('/menu', [MenuController::class, 'index']);

// app/Http/Controllers/MenuController.php
public function index() {
    $pizzas = Pizza::all();        // ask the Model
    return view('menu', [          // pass to the View
        'pizzas' => $pizzas,
    ]);
}

// resources/views/menu.blade.php
@foreach ($pizzas as $pizza)
    <h3>{{ $pizza->name }}</h3>
    <p>Rs. {{ $pizza->price }}</p>
@endforeach
```

Reading the example line by line

Here is what each line is actually doing, in plain English:

• **Route::get('/menu', ...)** - when somebody visits /menu with a GET request (a normal page load), call the *index* method on *MenuController*.

• **Pizza::all()** - ask the Pizza model for every row in the pizzas table. Returns a Laravel collection (like a smart array).

• **return view('menu', [...])** - render the file *resources/views/menu.blade.php*, passing the variable *$pizzas* into it.

• **@foreach ($pizzas as $pizza)** - Blade's loop syntax. Same as PHP's foreach but with an @ symbol so Blade can clean it up.

• **{{ $pizza->name }}** - print the value, automatically escaped against XSS attacks.

> **TIP**
>
> Notice we wrote zero SQL. We did not open a database connection. We did not call mysqli_query. Eloquent did all of that quietly behind *Pizza::all()*. This is the productivity gain Laravel is famous for.

> **TIP**
>
> Keep controllers thin. The rule is: *controllers orchestrate, models do the work, views just display*. If your controller is doing complex calculations or 50 lines of database logic, move that into the model.

Where everything lives in a Laravel project

Folder	What's in it
app/Models	Your Eloquent classes (User, Pizza, Order).
app/Http/Controllers	Controllers that handle web requests.
app/Http/Middleware	Filters that run before requests (auth, admin).
app/Http/Requests	Form Request classes for validation.
routes/web.php	Browser routes (HTML pages).
routes/api.php	API routes (JSON endpoints).
resources/views	Blade templates.
resources/css and js	Frontend source files.
database/migrations	Schema definitions, version-controlled.
database/seeders	Sample data scripts.
public	Files actually served to the browser. index.php is here.
.env	Secret config (database password, app key). Never commit.

A friendly Laravel glossary

You will see these terms over and over in this tutorial and in the Laravel docs. Skim this once now - you do not need to memorise it. Come back to it whenever a word feels unfamiliar.

Term	What it means in plain English
Artisan	Laravel's command-line helper. Anything starting with *php artisan* is an Artisan command.
Composer	The PHP package manager. Installs Laravel itself and any extra packages your project needs.
Eloquent	Laravel's tool for talking to the database with PHP objects instead of raw SQL queries.
Migration	A PHP file that describes a change to your database (create a table, add a column, etc).
Seeder	A PHP file that inserts sample data into your database.
Blade	Laravel's template engine. Lets you write HTML with @if, @foreach, and {{ ... }} placeholders.
Middleware	A filter that runs before a controller. Used for things like "only logged-in users" or "admin only".
Form Request	A class that holds validation rules for an incoming form.
Facade	A short name like *Route* or *DB* that lets you call a deeper service. Just convenient shortcuts.
Service Provider	A class that wires up things when Laravel starts. You rarely touch these as a beginner.
Resource	A class that turns a model into JSON for an API. We do not use these in this project but you will meet them later.
.env	A text file holding secrets like your database password. Never commit it to Git.

> **TIP**
>
> Do not stress about understanding every term right now. By the time you finish chapter 12, you will have used most of them in real code, and they will feel natural.

Chapter 3 - Tools You Need to Install

Laravel 13 needs a slightly newer toolchain than plain PHP projects. Install everything below before chapter 4 - it is all free, all open source, and works on Windows, Mac, and Linux. Take your time. Half the battle in software is getting your tools set up correctly.

What you need and why

Tool	Why we need it	Where to get it
PHP 8.3+	Laravel 13 minimum requirement.	php.net or via XAMPP/Herd
Composer	PHP package manager. Installs Laravel.	getcomposer.org
Node.js 20+	Builds frontend assets (CSS / JS).	nodejs.org
MySQL 8	The database. XAMPP includes it.	Comes with XAMPP
VS Code	Code editor with great PHP and Blade support.	code.visualstudio.com
Git	Version control.	git-scm.com
GitHub account	Online home for your code.	github.com

The easy path - Laravel Herd (recommended for 2026)

Laravel Herd is a free desktop app that bundles PHP, Composer, Node, and a tiny web server. One installer, zero configuration. It is available for Windows and Mac. If you are starting fresh in 2026, this is what we recommend.

Why Herd instead of XAMPP?

- Herd starts your apps automatically at **http://your-app-name.test** with no Apache config.
- It bundles multiple PHP versions - switch with one click.
- It includes the Laravel installer globally - *laravel new* works out of the box.
- It plays nicely with Mac and Windows file permissions.
- Tiny memory footprint compared to XAMPP's full Apache + MySQL stack.

Installing Herd (Windows or Mac)

1. Visit **herd.laravel.com**.
2. Click **Download** and run the installer.
3. Click Next, agree to the licence, click Install.
4. Open the Herd app from your Start menu (Windows) or Applications (Mac).
5. Choose a folder for your Laravel projects (e.g. *C:\\code* or *~/code*).
6. Click **Add Path**. Done.

> **NOTE**
>
> Herd does not include MySQL. For the database you can either install **DBngin** (free, by the same team as Herd) or use the MySQL that ships with XAMPP alongside Herd. Both work fine.

Alternative path - XAMPP (still works in 2026)

If you already followed our PHP tutorial at egotechworld.com or you are on Linux where Herd is not available, XAMPP is a fine choice. Make sure your XAMPP version includes PHP 8.3 or newer.

1. Visit **apachefriends.org** and download XAMPP.
2. Run the installer; tick Apache, MySQL, and PHP at minimum.
3. Open the XAMPP Control Panel and click **Start** next to Apache and MySQL.
4. Visit **http://localhost** in your browser - if you see the XAMPP welcome page, you are set.

> **WARNING**
>
> On Windows, port 80 is sometimes used by Skype or IIS. If Apache won't start, open Config -> Apache (httpd.conf), search for **Listen 80** and change it to **Listen 8080**. Then visit **http://localhost:8080** instead.

Installing Composer

Composer is to PHP what npm is to JavaScript - it downloads and manages libraries. Laravel itself is just a Composer package.

On Windows

1. Visit **getcomposer.org/download**.
2. Download **Composer-Setup.exe**.
3. Run it. When asked, point it to your PHP executable (e.g. *C:\\xampp\\php\\php.exe* or the path Herd shows you).
4. Tick "Add to PATH" when offered.
5. Open a NEW terminal and run *composer --version*.

On Mac or Linux

```
# In a terminal:
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer
composer --version
```

Installing Node.js

Laravel uses Node only to compile your CSS and JavaScript with a tool called **Vite**. You will not write any Node.js code - we just need it installed.

1. Visit **nodejs.org** and download the LTS version (currently 20.x or 22.x).
2. Run the installer with default options.
3. Open a NEW terminal and run *node --version* and *npm --version*.
4. Both should print version numbers.

Quick check - all tools at once

Open a terminal and run each of these commands:

```
php --version       # should say 8.3 or higher
composer --version  # should say 2.x
node --version      # should say v20 or higher
npm --version       # any 10.x or higher
git --version       # any version is fine
```

What if a command says 'not found'?

• Close the terminal and open a fresh one. Tools added to your PATH are not picked up by terminals that were already open.

• On Windows: search "environment variables" in the Start menu, click **Edit the system environment variables**, then **Environment Variables**. Make sure your tool's bin folder is in the **Path** list.

• On Mac/Linux: edit *~/.zshrc* or *~/.bashrc* and add the tool's path with *export PATH="$PATH:/path/to/tool"*.

• Last resort: re-install with the "Add to PATH" checkbox ticked.

VS Code extensions worth installing

• **Laravel Blade Snippets** - autocomplete for Blade directives.

• **Laravel Extra Intellisense** - autocompletes route names, view names, configs.

• **PHP Intelephense** - real PHP intellisense, not just basic.

• **Tailwind CSS IntelliSense** - if you use Tailwind (Breeze does).

• **GitLens** - see who wrote each line and when.

> **TIP**
>
> All these extensions are free. Search the Extensions panel in VS Code (Ctrl+Shift+X) and install with one click each.

Chapter 4 - Installing Laravel 13 - Your First Project

Time to create the project. We will use Composer to download Laravel 13, then start the built-in web server with Artisan. Total time: about two minutes.

Step 1 - Create the project

In your terminal, navigate to the folder where you keep code projects (create one called *code/* in your home directory if you don't have one), then run:

```
cd ~/code
composer create-project laravel/laravel pizza-app

# wait about 30 seconds while Composer downloads everything

cd pizza-app
```

When it finishes you will have a folder called **pizza-app** with the entire Laravel skeleton inside.

Step 2 - Start the development server

```
php artisan serve
```

Artisan prints something like *Server running on [http://127.0.0.1:8000]*. Open that URL in your browser - you will see the Laravel welcome page. If you see it, congratulations: you just installed and ran a Laravel app.

What just happened?

Composer downloaded the entire Laravel framework (a few hundred PHP files spread across the *vendor/* folder) plus a starter project skeleton. Then Artisan started PHP's built-in web server pointed at the *public/* folder. The first request triggered *public/index.php*, which booted Laravel, which matched the URL / against *routes/web.php* and rendered the welcome view.

You will not run Apache or Nginx during development - *php artisan serve* is enough. In production we will use a real web server, but that is a chapter 14 problem.

> **TIP**
>
> Keep that terminal window open. Artisan stays running and reloads your code automatically. Open a SECOND terminal in the same folder for running other commands. You will end up with three terminals soon - one for serve, one for npm, one for free commands.

Step 3 - Tour of the folders

Run *code .* from inside **pizza-app/** to open the project in VS Code. You will see a folder tree like the one below. Do not panic at the size of it - 90 percent of your work happens in just five folders, marked with comments.

```
pizza-app/
+-- app/
|   +-- Http/Controllers/      # YOU WILL EDIT - controllers here
|   `-- Models/                # YOU WILL EDIT - User.php already exists
+-- bootstrap/                 # Laravel boots itself here (rarely touch)
+-- config/                    # Sometimes - config files (database, mail)
+-- database/
|   +-- migrations/            # YOU WILL EDIT - schema definitions
|   +-- seeders/               # YOU WILL EDIT - sample data
|   `-- factories/             # for tests (later)
+-- public/                    # The browser only sees this folder
|   `-- index.php              # THE entry point of the whole app
+-- resources/
|   +-- views/                 # YOU WILL EDIT - Blade templates
|   +-- css/                   # Tailwind / your CSS
|   `-- js/                    # JavaScript source
+-- routes/
|   +-- web.php                # YOU WILL EDIT - browser routes
|   +-- api.php                # JSON routes (we use web.php instead)
|   `-- console.php            # custom Artisan commands
+-- storage/                   # logs, uploaded files, caches
+-- tests/                     # test files
+-- vendor/                    # Composer-managed (NEVER touch)
+-- .env                       # YOU WILL EDIT - secret config - never commit
+-- artisan                    # the command-line tool
+-- composer.json              # PHP dependencies
`-- package.json               # JS dependencies
```

The five folders you actually live in

- **app/Models** - one PHP file per database table.
- **app/Http/Controllers** - one PHP file per group of pages.
- **resources/views** - one Blade file per HTML page.
- **routes/web.php** - the URL-to-controller mapping.
- **database/migrations** - schema changes, version controlled.

If you can keep these five locations in your head, the rest of the framework folders will only matter occasionally.

Step 4 - Configure the .env file

Open **.env** in VS Code. This is where database credentials and other secrets live. The file is plain text with KEY=value lines. Find the database lines and update them to match the database we will create next.

```
APP_NAME="Pizza Express"
APP_ENV=local
APP_DEBUG=true
APP_URL=http://localhost:8000

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=pizza_app
DB_USERNAME=root
DB_PASSWORD=
```

What each .env line means

Key	What it does
APP_NAME	The name shown in your app's title bar and emails.
APP_ENV	**local** for development, **production** when deployed.
APP_DEBUG	**true** shows error details on screen. ALWAYS false in production.
APP_URL	The base URL of your app - used when generating links in emails.
DB_CONNECTION	Which database driver to use (mysql, sqlite, pgsql).
DB_HOST / DB_PORT	Where the database lives. 127.0.0.1:3306 for local MySQL.
DB_DATABASE	The name of the database to connect to.
DB_USERNAME / DB_PASSWORD	Login credentials for the database.

> **WARNING**
>
> **.env** contains secrets. Laravel's default .gitignore already excludes it from Git, but never paste your real production .env into chats, screenshots, or commits.

Step 5 - Create the database

Open phpMyAdmin (or your favourite DB tool). Create a new database named **pizza_app** with collation **utf8mb4_unicode_ci**. Leave it empty - Laravel migrations will create the tables in the next chapter.

Test that Laravel can connect by running:

```
php artisan migrate
```

If you see *Migration table created successfully* followed by a list of default migrations, you are connected. If you see an error about access denied, double-check the DB_USERNAME and DB_PASSWORD lines in .env.

Chapter 5 - Database Setup, Migrations, and Eloquent Models

Now we design our database. Laravel calls these designs **migrations** - they are PHP files that describe table structures. The big advantage: if a teammate clones your project, they run *php artisan migrate* and instantly have the same tables you do. No more emailing SQL files.

Our three tables

Table	Holds	Key columns
users	Customers and the admin (Laravel creates this for us).	id, name, email, password, role
pizzas	The two pizzas on the menu.	id, name, description, price
orders	Every order placed.	id, user_id, pizza_id, quantity, status, timestamps

Step 1 - Add a 'role' column to users

Laravel already created a *create_users_table* migration. We will add a tiny migration that adds a **role** column. Run:

```
php artisan make:migration add_role_to_users_table --table=users
```

Open the new file in **database/migrations/** (the one with today's timestamp and 'add_role_to_users') and replace its content with:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration {
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('role')->default('customer')->after('email');
        });
    }

    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('role');
        });
    }
};
```

Step 2 - Create the Pizza model and migration

One Artisan command creates the model, migration, factory, and seeder all at once:

```
php artisan make:model Pizza -mfs

# -m = migration
# -f = factory (for testing)
# -s = seeder
```

Open **database/migrations/...create_pizzas_table.php** and fill in the schema:

```
public function up(): void
{
    Schema::create('pizzas', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->text('description');
        $table->decimal('price', 8, 2);
        $table->timestamps();
    });
}
```

Reading the migration line by line

- **$table->id()** - creates an auto-incrementing integer primary key called *id*. Standard for every Laravel table.
- **$table->string('name')** - VARCHAR(255). Use for short text.
- **$table->text('description')** - TEXT type. Use for longer text up to 64KB.
- **$table->decimal('price', 8, 2)** - decimal with 8 total digits, 2 after the point. Up to 999,999.99 - perfect for prices.
- **$table->timestamps()** - adds *created_at* and *updated_at* columns that Laravel manages automatically.

> **WARNING**
>
> Never use **float** for money. Floating-point numbers cannot represent values like 0.10 exactly, leading to weird off-by-one-cent bugs. Always use **decimal** for currency.

Step 3 - Create the Order model and migration

```
php artisan make:model Order -mfs
```

Edit the generated **...create_orders_table.php**:

```
public function up(): void
{
    Schema::create('orders', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id')->constrained()->cascadeOnDelete();
        $table->foreignId('pizza_id')->constrained();
        $table->unsignedInteger('quantity');
        $table->string('status')->default('pending');
        $table->timestamps();
    });
}
```

What is happening on each line

- **foreignId('user_id')->constrained()** - shorthand that creates a *user_id* integer column AND a foreign key linking it to the *users* table's *id* column.
- **cascadeOnDelete()** - if a user is deleted, every order they made is deleted too. Saves us from orphan rows.
- **foreignId('pizza_id')->constrained()** - same pattern for the pizza link. We do NOT cascade delete on this one - we may want to soft-delete pizzas while keeping order history.
- **unsignedInteger('quantity')** - integer that cannot be negative. You cannot order minus 3 pizzas.
- **string('status')->default('pending')** - new orders start as pending. Admin will flip this to 'approved' later.

> **NOTE**
>
> **foreignId(...)->constrained()** only works because we are following Laravel's naming convention - column *user_id* automatically links to table *users*. If you name a column differently you must spell out the relationship explicitly.

Step 4 - Run all migrations

```
php artisan migrate
```

Check phpMyAdmin - you should see *users*, *pizzas*, *orders*, plus a few Laravel internal tables. The *users* table now has a *role* column.

Step 5 - Wire up the model relationships

Eloquent models can know about each other. Open **app/Models/User.php** and add this method inside the class:

```
public function orders()
{
    return $this->hasMany(Order::class);
}

public function isAdmin(): bool
{
    return $this->role === 'admin';
}
```

Open **app/Models/Pizza.php** and add:

```
protected $fillable = ['name', 'description', 'price'];

public function orders()
{
    return $this->hasMany(Order::class);
}
```

Open **app/Models/Order.php** and add:

```
protected $fillable = ['user_id', 'pizza_id', 'quantity', 'status'];

public function user()
{
    return $this->belongsTo(User::class);
}

public function pizza()
{
    return $this->belongsTo(Pizza::class);
}

public function total(): float
{
    return $this->pizza->price * $this->quantity;
}
```

> **TIP**
>
> Once relationships are defined, you can write things like **$order->user->name** or **$user->orders** directly in Blade templates. No JOINs, no queries you have to remember to write. This is the magic of Eloquent.

Chapter 6 - Seeding the Pizzas Table

Migrations created the empty pizzas table. Now we need to put two actual pizzas in it. We could open phpMyAdmin and click around, but Laravel offers a cleaner way: **seeders**. Seeders are PHP files that insert sample data with a single Artisan command.

Why use seeders?

- Anyone who clones your project can run *php artisan db:seed* and instantly get the same demo data.
- If you ever wipe your database during development, one command rebuilds it.
- Seeders live in Git, so the data is version-controlled along with the code.
- When deploying to production, you can choose to run seeders or not.

Step 1 - Open the PizzaSeeder

We already created it in chapter 5 with the *-s* flag. Open **database/seeders/PizzaSeeder.php** and replace its contents:

```
<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;
use App\Models\Pizza;

class PizzaSeeder extends Seeder
{
    public function run(): void
    {
        Pizza::create([
            'name' => 'Margherita',
            'description' => 'Classic tomato, mozzarella, and fresh basil.',
            'price' => 1200.00,
        ]);

        Pizza::create([
            'name' => 'Pepperoni',
            'description' => 'Spicy pepperoni with mozzarella and tomato sauce.',
            'price' => 1500.00,
        ]);
    }
}
```

Step 2 - Register the seeder

Open **database/seeders/DatabaseSeeder.php** and find the *run* method. Add a call to our new seeder, plus an admin user:

```
public function run(): void
{
    // Create the admin account
    \App\Models\User::factory()->create([
        'name' => 'Site Admin',
        'email' => '[email protected]',
        'password' => bcrypt('admin123'),
        'role' => 'admin',
    ]);

    // Seed pizzas
    $this->call([
        PizzaSeeder::class,
    ]);
}
```

Step 3 - Run the seeders

```
php artisan db:seed
```

Open phpMyAdmin and check the **pizzas** table - two rows. Check **users** - one admin row. We are ready to build the user-facing pages.

> **TIP**
>
> To wipe everything and re-run all migrations and seeders in one go, use **php artisan migrate:fresh --seed**. This is your best friend during development. Never run it in production.

Chapter 7 - Authentication with Laravel Breeze

Authentication - registration, login, password reset - is the most common feature in any web app, and the easiest place to make security mistakes. Laravel solves both problems in one shot with **Breeze**, an official starter kit that gives you all of these screens in less than a minute.

What Breeze gives us for free

- Registration page (with email + password validation).
- Login page (with remember-me checkbox).
- Forgot-password and password-reset flow.
- Email verification (optional).
- Profile page where users update their name/email/password.
- Logout route.
- Middleware that protects routes (only logged-in users can access).

Why this matters

Authentication is a magnet for security mistakes. Storing passwords in plain text, weak hashing algorithms, missing CSRF protection, session fixation, timing attacks - new bugs are discovered every year. Breeze is written and maintained by the Laravel core team and audited by thousands of developers. Using it gets you closer to industry-standard security than 99 percent of hand-rolled login systems.

Step 1 - Install Breeze

```
composer require laravel/breeze --dev
php artisan breeze:install
```

What does each command do?

- **composer require laravel/breeze --dev** - downloads the Breeze package. The *--dev* flag puts it in the dev dependencies because we only need it during installation.
- **php artisan breeze:install** - copies all the auth views, controllers, routes, and tests into your project. After this command Breeze itself is no longer needed.

Breeze will ask which frontend stack you want. Choose **Blade with Alpine** - the simplest option, perfect for beginners. It uses Tailwind CSS by default. Then it asks about dark mode (yes if you want it) and Pest/PHPUnit for tests (Pest is the modern choice).

Step 2 - Install JS dependencies and build

```
npm install
npm run dev
```

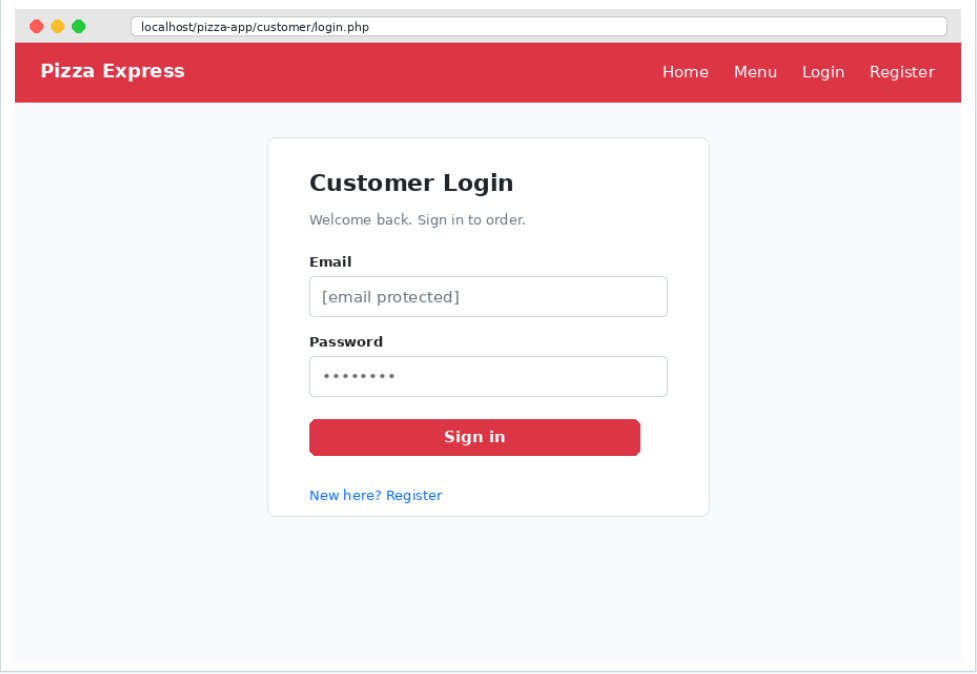
npm run dev starts a watcher that rebuilds CSS and JS whenever you save a Blade file. Keep this terminal running alongside *php artisan serve*. You now need three terminals open at once:

- Terminal 1: *php artisan serve* - the dev server.
- Terminal 2: *npm run dev* - the asset watcher.
- Terminal 3: free for running Artisan commands as needed.

> **TIP**
>
> If you would rather use Bootstrap 5 (matching the screenshots in this tutorial), uninstall Tailwind later or skip Breeze and write the auth controllers manually. We will continue with Tailwind for simplicity but the principles are identical.

Step 3 - Try it out

Visit **http://localhost:8000/register** in your browser. You should see a clean registration form. Create an account with any email and a password of at least 8 characters. After registering, you land on a **/dashboard** page Breeze created.



What our login page will look like after we restyle Breeze with Bootstrap 5 (or you can keep the default Tailwind look).

Test these scenarios

Spend ten minutes playing with what Breeze gave you. Try each of these and notice how robust the default behaviour is:

1. Register with an invalid email - it shows a friendly error.
2. Register with a 4-character password - it complains.

3. Try to register with the same email twice - it tells you the email is already taken.

4. Log out, then visit **/dashboard** directly - you are bounced to the login page.

5. Log in with the wrong password three times - notice it does not say "wrong password" specifically (security best practice).

6. Use the **Forgot Password** link - it sends a reset email (check *storage/logs/laravel.log* for the email content during local development).

Step 4 - Look at what Breeze added

Breeze created a lot of files. The most important ones to know about:

File	What it does
routes/auth.php	All auth routes (login, register, password reset).
app/Http/Controllers/Auth/	Controllers for each auth action.
resources/views/auth/	Blade views for login, register, etc.
app/Http/Requests/Auth/LoginRequest.php	Form Request for the login form.
resources/views/dashboard.blade.php	The post-login landing page.
resources/views/components/	Reusable Blade UI pieces like buttons and inputs.
tests/Feature/Auth/	Pest tests for the entire auth flow.

We will not modify Breeze's files much - they are battle-tested. We will use them as building blocks and add our pizza-specific pages on top.

Chapter 8 - Routes, Controllers, and the Menu Page

Time to build our first real feature: the menu page. We will create a controller, a route, and a Blade view. Pay close attention - this is the same pattern you will repeat for every page in every Laravel app you ever write.

Step 1 - Create the MenuController

```
php artisan make:controller MenuController
```

Open **app/Http/Controllers/MenuController.php** and write the *index* method:

```
<?php

namespace App\Http\Controllers;

use App\Models\Pizza;

class MenuController extends Controller
{
    public function index()
    {
        $pizzas = Pizza::orderBy('id')->get();

        return view('menu.index', [
            'pizzas' => $pizzas,
        ]);
    }
}
```

Step 2 - Register the route

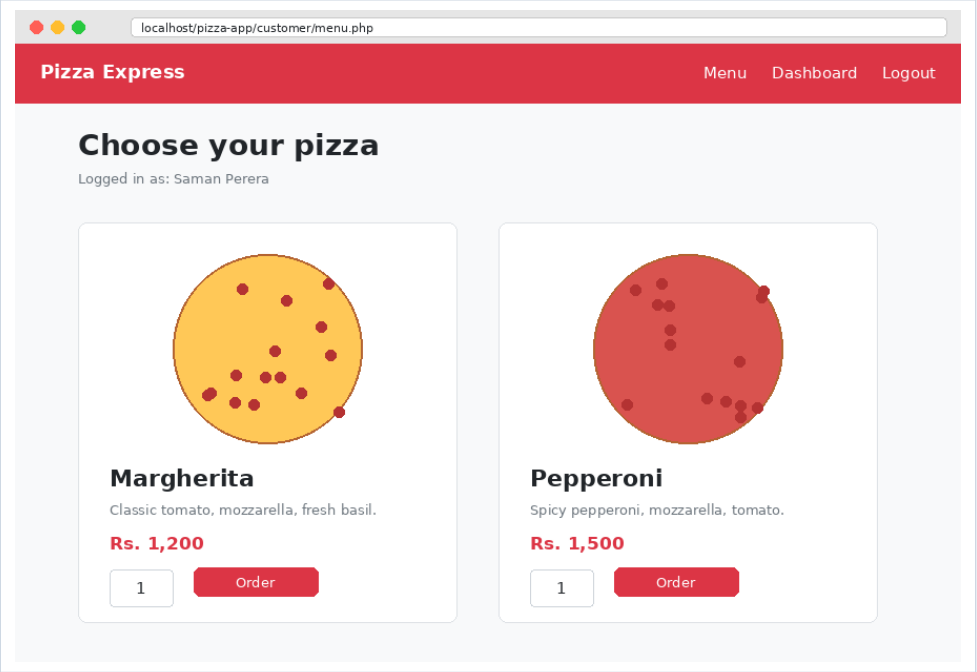
Open **routes/web.php** and add a route. The whole file should look like this:

```
<?php

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\MenuController;

Route::get('/', function () {
    return view('welcome');
});

Route::get('/menu', [MenuController::class, 'index'])
    ->middleware('auth')
    ->name('menu.index');

Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');

require __DIR__ . '/auth.php';
```

> **NOTE**
>
> **->middleware('auth')** means "only logged-in users can see this page". **->name('menu.index')** gives the route a nickname so we can refer to it as **route('menu.index')** in views instead of hard-coding URLs. Always name your routes.

Step 3 - Create the Blade view

Now let's build the screen we previewed:



Menu page - what we are about to build.

Create a folder **resources/views/menu/** and inside it a file called **index.blade.php**:

```
<x-app-layout>
    <x-slot name="header">
        <h2 class="font-bold text-xl">Choose your pizza</h2>
    </x-slot>

    <div class="container py-4">
        <p class="text-muted mb-4">
            Logged in as: {{ auth()->user()->name }}
        </p>

        <div class="row g-4">
            @foreach ($pizzas as $pizza)
                <div class="col-md-6">
                    <div class="card h-100 shadow-sm">
                        <div class="card-body">
                            <h3 class="card-title">{{ $pizza->name }}</h3>
                            <p class="text-muted">{{ $pizza->description }}</p>
                            <p class="h4 text-danger fw-bold">
                                Rs. {{ number_format($pizza->price) }}
                            </p>

                            <form method="POST" action="{{ route('orders.store') }}"
                                  class="d-flex gap-2">
                                @csrf
```

```
                                <input type="hidden" name="pizza_id" value="{{ $pizza->id }}">
                                <input type="number" name="quantity"
                                       value="1" min="1" max="10"
                                       class="form-control" style="width: 80px;">
                                <button class="btn btn-danger flex-grow-1">
                                    Order
                                </button>
                            </form>
                        </div>
                    </div>
                </div>
            @endforeach
        </div>
    </div>
</x-app-layout>
```

Step 4 - Add Bootstrap to the layout

Breeze ships with Tailwind, but our screenshots use Bootstrap. The easy way: include Bootstrap from a CDN. Open **resources/views/layouts/app.blade.php** and add this line just before the closing **</head>** tag:

```
<link href="https://cdn.jsdelivr.net/npm/[email protected]/dist/css/bootstrap.min.css"
      rel="stylesheet">
```

> **NOTE**
>
> Mixing Tailwind and Bootstrap is not ideal for production - they can fight over the same class names. For learning purposes, it works fine. In a real project, pick one and stick with it.

Step 5 - Visit the page

Make sure both *php artisan serve* and *npm run dev* are still running. In the browser, log in as your test user, then visit **http://localhost:8000/menu**. You should see both pizzas with Order buttons. Beautiful.

Chapter 9 - Placing an Order with Form Requests

Now we handle what happens when a customer clicks **Order**. The Order form posts to **/orders**, which we will route to an **OrderController**. Along the way we will meet one of Laravel's best ideas - the **Form Request**.

What is a Form Request?

When a user submits a form, you have to do two things before saving anything to the database:

1. **Authorise** - is this user allowed to do this action at all? (Are they logged in? Do they own the resource?)
2. **Validate** - is the data they sent actually valid? (Is the quantity a positive number? Does that pizza ID really exist?)

In plain PHP we would write a long pile of if-statements at the top of every controller method to check these things. Laravel offers a much cleaner solution: a Form Request is a tiny class that holds the rules in one place. The controller stays focused on the actual work.

Step 1 - Create the controller and Form Request

```
php artisan make:controller OrderController
php artisan make:request StoreOrderRequest
```

Two new files now exist:

- **app/Http/Controllers/OrderController.php** - the controller.
- **app/Http/Requests/StoreOrderRequest.php** - the form request.

Step 2 - Write the Form Request

Open **app/Http/Requests/StoreOrderRequest.php** and replace the two methods that are already in there:

```
public function authorize(): bool
{
    return auth()->check();
}

public function rules(): array
{
    return [
        'pizza_id' => ['required', 'integer', 'exists:pizzas,id'],
        'quantity' => ['required', 'integer', 'min:1', 'max:10'],
    ];
}
```

Reading the rules

Each form field gets a list of rules. Laravel checks them all and only passes the request to the controller if every single rule is satisfied. Here are the rules we used:

Rule	What it checks
required	The field must be present and not empty.
integer	The value must be a whole number (or a string that looks like one).
exists:pizzas,id	The value must match an *id* that actually exists in the *pizzas* table. Stops people from submitting fake pizza IDs.
min:1	The value must be at least 1.
max:10	The value must be 10 or less.

More rules you will use later

Laravel ships with around 80 built-in validation rules. The most common ones for everyday forms:

- **email** - must be a valid email address.
- **url** - must be a valid URL.
- **unique:users,email** - no other row in users has this email (perfect for registration).
- **confirmed** - the field has a matching *field_confirmation* (perfect for password fields).
- **in:pending,approved,cancelled** - the value is one of these.
- **date**, **after:today** - date validations.
- **image**, **mimes:jpg,png**, **max:2048** - file uploads.

> **TIP**
>
> **exists:pizzas,id** stops a malicious user from typing a fake pizza ID into their browser DevTools and submitting it. Without this rule, your *orders* table would have rows pointing to non-existent pizzas. Always validate foreign keys.

Step 3 - Write the OrderController

The controller is now wonderfully simple. All of the input validation happens before this method even runs - by the time we are inside, we know the data is clean. Open **app/Http/Controllers/OrderController.php**:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Requests\StoreOrderRequest;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;

class OrderController extends Controller
{
    public function store(StoreOrderRequest $request): RedirectResponse
    {
        Order::create([
            'user_id'  => auth()->id(),
            'pizza_id' => $request->pizza_id,
            'quantity' => $request->quantity,
            'status'   => 'pending',
        ]);

        return redirect()
            ->route('dashboard.customer')
            ->with('success', 'Order placed! It is being prepared.');
    }
}
```

What this controller does, line by line

• **store(StoreOrderRequest $request)** - the type-hint is the magic. Laravel sees we want a StoreOrderRequest, automatically runs the validation rules, and only calls our method if everything passes. If validation fails, the user goes back to the menu page with error messages.

• **auth()->id()** - returns the ID of the currently logged-in user. The auth helper is available everywhere thanks to Laravel facades.

• **Order::create([...])** - inserts a new row into the orders table. Eloquent fills in *created_at* and *updated_at* automatically.

• **redirect()->route('dashboard.customer')** - sends the browser to the dashboard page. We use the route's name instead of hard-coding /dashboard.

• **->with('success', '...')** - attaches a one-time message called a *flash message*. It shows up on the next page and then disappears.

> **TIP**
>
> The redirect-with-flash pattern is the standard way to do a successful form submission in Laravel. Never re-render the form view directly after a successful POST - always redirect. This stops the dreaded "do you want to resubmit this form?" browser popup when someone refreshes.

Step 4 - Add the route

Open **routes/web.php** and add the orders route inside an auth group (we will refactor existing routes too):

```
use App\Http\Controllers\OrderController;
use App\Http\Controllers\CustomerDashboardController;

Route::middleware('auth')->group(function () {
    Route::get('/menu', [MenuController::class, 'index'])->name('menu.index');
    Route::post('/orders', [OrderController::class, 'store'])->name('orders.store');
    Route::get('/dashboard', [CustomerDashboardController::class, 'index'])
        ->name('dashboard.customer');
});
```

Step 5 - Add a flash message to the layout

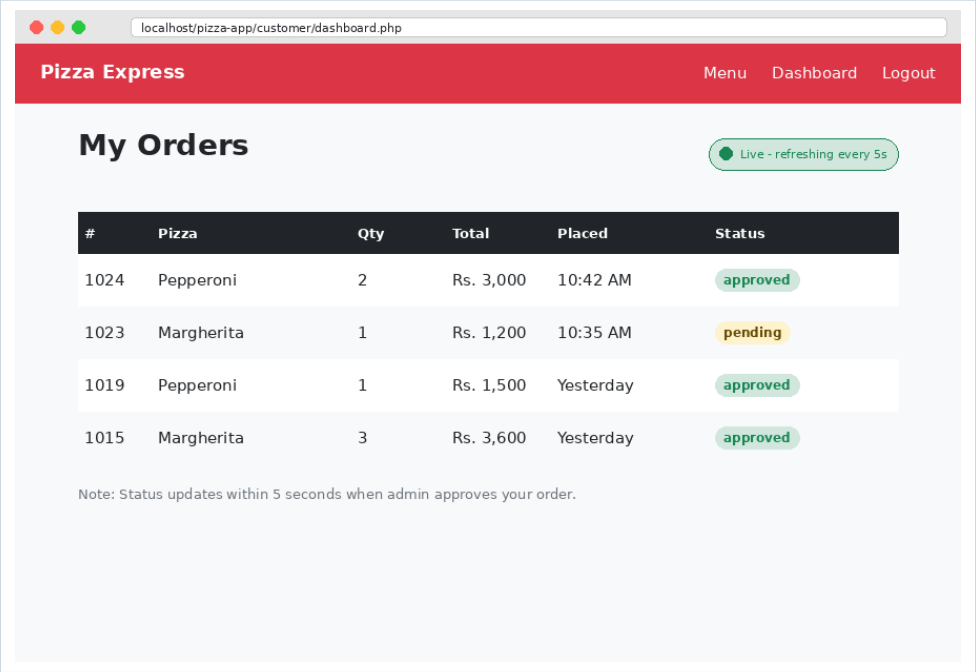
When we redirected, we attached **->with('success', ...)**. That's a flash message - it lives in the session for one request. To display it, add this snippet to **resources/views/layouts/app.blade.php** just inside the <body>:

```
@if (session('success'))
    <div class="container mt-3">
        <div class="alert alert-success">
            {{ session('success') }}
        </div>
    </div>
@endif
```

Try it: log in, place an order from /menu. You should be redirected to the dashboard with a green success banner. Check phpMyAdmin - there's a new row in the **orders** table with status 'pending'.

Chapter 10 - Customer Dashboard with Live Updates

The customer dashboard shows the user their orders. The clever part: the page refreshes itself every 5 seconds, so when an admin approves an order, the badge changes from yellow to green automatically. No page reload, no email needed. This feature feels magical to users but is shockingly simple to build.



Customer dashboard with live status updates.

How live updates work - the simple version

Real-time websites can be built with WebSockets, server-sent events, or fancy push notifications. We are going to use the simplest technique that fits the job: **polling**.

Polling means the browser keeps asking the server "any updates?" every few seconds. If there is something new, it updates the page. If not, nothing happens. It is like a child in the back of a car asking "are we there yet?" every minute - except it works.

The pattern in three pieces

1. **1.** A regular Blade page renders the dashboard frame (navbar, table headers, an empty <tbody>).
2. **2.** A separate URL like */dashboard/orders.json* returns just the raw data as JSON - no HTML.
3. **3.** JavaScript on the page calls that JSON URL every 5 seconds and rebuilds the table rows from the response.

We need two controller methods (the page and the JSON feed), one Blade view, and about 20 lines of JavaScript. That is the whole feature.

Step 1 - Create the controller

```
php artisan make:controller CustomerDashboardController
```

Open the new file and write two methods - one for the page itself, one for the JSON feed that JavaScript will poll:

```
<?php

namespace App\Http\Controllers;

use App\Models\Order;
use Illuminate\Http\JsonResponse;

class CustomerDashboardController extends Controller
{
    public function index()
    {
        return view('dashboard.customer');
    }

    public function ordersJson(): JsonResponse
    {
        $orders = Order::with('pizza')
            ->where('user_id', auth()->id())
            ->orderByDesc('created_at')
            ->get()
            ->map(fn ($o) => [
                'id'         => $o->id,
                'pizza'      => $o->pizza->name,
                'quantity'   => $o->quantity,
                'total'      => number_format($o->total(), 2),
                'placed_at'  => $o->created_at->format('h:i A'),
                'status'     => $o->status,
            ]);

        return response()->json($orders);
    }
}
```

> **NOTE**
>
> **Order::with('pizza')** is called *eager loading*. It fetches the related pizza for every order in ONE extra query, instead of one query per order (the dreaded N+1 problem). Laravel makes the right thing this easy to do.

Step 2 - Add the JSON route

In **routes/web.php**, inside the auth middleware group:

```
Route::get('/dashboard/orders.json',
    [CustomerDashboardController::class, 'ordersJson'])
    ->name('dashboard.customer.json');
```

Step 3 - Build the Blade view

Create **resources/views/dashboard/customer.blade.php**:

```
<x-app-layout>
    <x-slot name="header">
        <h2 class="font-bold text-xl">My Orders</h2>
    </x-slot>

    <div class="container py-4">
        <div class="d-flex justify-content-between align-items-center mb-3">
            <h3>My Orders</h3>
            <span class="badge bg-success-subtle text-success border border-success">
                <span class="dot"></span> Live - refreshing every 5s
            </span>
        </div>

        <div class="table-responsive">
            <table class="table table-striped align-middle">
                <thead class="table-dark">
                    <tr>
                        <th>#</th><th>Pizza</th><th>Qty</th>
                        <th>Total</th><th>Placed</th><th>Status</th>
                    </tr>
                </thead>
                <tbody id="ordersBody">
                    <tr><td colspan="6" class="text-center">Loading...</td></tr>
                </tbody>
            </table>
        </div>
    </div>

    @push('scripts')
    <script>
    async function loadOrders() {
        const res = await fetch('{{ route('dashboard.customer.json') }}');
        const data = await res.json();
        const body = document.getElementById('ordersBody');

        if (!data.length) {
            body.innerHTML =
              '<tr><td colspan="6" class="text-center">No orders yet.</td></tr>';
            return;
        }

        body.innerHTML = data.map(o => `
            <tr>
                <td>${o.id}</td>
                <td>${o.pizza}</td>
                <td>${o.quantity}</td>
                <td>Rs. ${o.total}</td>
                <td>${o.placed_at}</td>
                <td>
                  <span class="badge bg-${o.status === 'approved'
                    ? 'success' : 'warning text-dark'}">
```

```
                    ${o.status}
                  </span>
                </td>
            </tr>
        `).join('');
    }

    loadOrders();
    setInterval(loadOrders, 5000);
    </script>
    @endpush
</x-app-layout>
```

Step 4 - Make sure the layout supports @push

In **resources/views/layouts/app.blade.php**, just before the closing **</body>** tag, add:

```
@stack('scripts')
```

This is where Blade injects any scripts pushed from individual views. It is the cleanest way to add per-page JavaScript without bloating the layout.

> **TIP**
>
> Polling every 5 seconds is fine for a learning project. For a production app with thousands of users, look into **Laravel Reverb** (the official WebSocket server, included with Laravel 13) which pushes updates instead. We will stay with polling - it works on any shared host with zero configuration.

Chapter 11 - Admin Role and Middleware Protection

Right now any logged-in user could visit /admin/dashboard if we built it. We need a guard that says: "only users with role = admin may enter". Laravel's term for these guards is **middleware**.

What is middleware, really?

Middleware is a fancy word for a checkpoint. Imagine a hotel corridor leading to the executive lounge. Before you reach the lounge door, you walk past a security guard who checks your room key. If you have the right access level, the guard waves you through. If not, you are turned around.

In Laravel, every request travels through a chain of middleware before reaching the controller. Each one can either let the request continue down the chain, or short-circuit it with a redirect or an error.

Examples of middleware in everyday Laravel apps

Middleware	What it does
auth	Blocks the request if the user is not logged in. Sends them to the login page.
admin (we'll build this)	Blocks the request if the logged-in user is not an admin. Returns 403.
throttle:60,1	Allows only 60 requests per minute - rate limiting.
verified	Requires the user to have verified their email.
EnsureFeatureIsEnabled	Custom - blocks requests when a feature flag is off.

Middleware is the cleanest way to handle cross-cutting concerns. Without it, you would have to repeat the same if-statements at the top of every controller method. With it, you write the rule once and apply it to many routes.

Step 1 - Create the AdminMiddleware

```
php artisan make:middleware AdminMiddleware
```

Artisan creates **app/Http/Middleware/AdminMiddleware.php**. Open it and replace the *handle* method:

```
public function handle(Request $request, Closure $next): Response
{
    if (!auth()->check() || !auth()->user()->isAdmin()) {
        abort(403, 'Admins only.');
    }

    return $next($request);
}
```

How this method works

• **$request** - the incoming HTTP request (URL, headers, form data).

• **$next** - a callable that represents the rest of the chain. Calling *$next($request)* means "let the request continue".

• **auth()->check()** - true if a user is logged in.

• **auth()->user()->isAdmin()** - calls the helper we wrote on the User model in chapter 5.

• **abort(403)** - immediately returns a 403 Forbidden page. The request never reaches the controller.

> **NOTE**
>
> If you need a different page than 403, you can replace it with something like *return redirect()->route('login')* for non-logged-in users. The principle is the same - the middleware decides whether to pass the request along.

Step 2 - Register the middleware alias

Right now Laravel does not know our middleware exists. We need to give it a short alias so we can apply it to routes by name. In Laravel 13, this is done in **bootstrap/app.php**. Find the **->withMiddleware(...)** call and update it:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->alias([
        'admin' => \App\Http\Middleware\AdminMiddleware::class,
    ]);
})
```

Now we can use *'admin'* as a route middleware anywhere - just like Laravel's built-in 'auth' alias. We will use it in chapter 12 to protect the admin routes.

Step 3 - Redirect admins after login

When the admin signs in, we want them sent to /admin/dashboard, not the customer dashboard. We need to tweak Breeze's login controller to look at the role and redirect accordingly. Open **app/Http/Controllers/Auth/AuthenticatedSessionController.php** (created by Breeze) and find the *store* method. After the *$request->authenticate()* line, replace the *return redirect()->intended(...)* line with:

```
$request->session()->regenerate();

if (auth()->user()->isAdmin()) {
    return redirect()->intended(route('admin.dashboard'));
}

return redirect()->intended(route('dashboard.customer'));
```

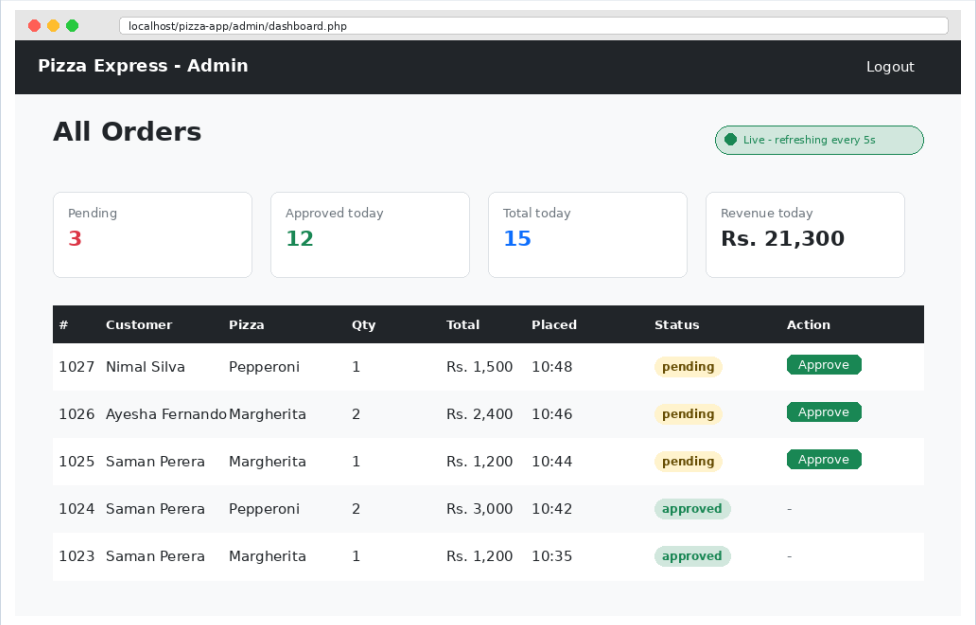
Why session regenerate?

$request->session()->regenerate() creates a fresh session ID after a successful login. This is a security best practice that prevents *session fixation attacks* - where an attacker tricks a victim into using a session ID the attacker already knows. Breeze does this for you, but it is good to know what the line is for.

> **TIP**
>
> The same login form serves both customers and admins. Behind the scenes, the redirect chooses where each role lands. This is much simpler than two separate login pages, and just as secure thanks to the role check.

Chapter 12 - Admin Dashboard and the Approve Action

The admin dashboard mirrors the customer dashboard but shows every order from every customer, and adds an Approve button to pending ones. The pattern is identical to what we built in chapter 10.



Admin dashboard - the command centre of the app.

Step 1 - Create the controller

```
php artisan make:controller Admin/DashboardController
```

Notice the **Admin/** prefix - it puts the file in **app/Http/Controllers/Admin/**. Grouping admin controllers in their own folder keeps the project tidy as it grows.

```
<?php

namespace App\Http\Controllers\Admin;

use App\Http\Controllers\Controller;
use App\Models\Order;
use Illuminate\Http\JsonResponse;
use Illuminate\Http\RedirectResponse;

class DashboardController extends Controller
{
    public function index()
    {
        return view('admin.dashboard');
    }

    public function ordersJson(): JsonResponse
    {
        $orders = Order::with(['pizza', 'user'])
            ->orderByDesc('created_at')
            ->get()
            ->map(fn ($o) => [
                'id'         => $o->id,
```

```
                'customer'  => $o->user->name,
                'pizza'     => $o->pizza->name,
                'quantity'  => $o->quantity,
                'total'     => number_format($o->total(), 2),
                'placed_at' => $o->created_at->format('h:i A'),
                'status'    => $o->status,
            ]);

        return response()->json($orders);
    }

    public function approve(Order $order): RedirectResponse
    {
        if ($order->status === 'pending') {
            $order->update(['status' => 'approved']);
        }

        return back()->with('success', "Order #{$order->id} approved.");
    }
}
```

> **NOTE**
>
> The **approve(Order $order)** method uses route model binding. Laravel sees the type-hint and automatically fetches the Order with that ID from the database. If no order matches, Laravel returns a 404 page automatically. One line of code does the work of ten.

Step 2 - Add the admin routes

In **routes/web.php**:

```
use App\Http\Controllers\Admin\DashboardController as AdminDashboardController;

Route::middleware(['auth', 'admin'])
    ->prefix('admin')
    ->name('admin.')
    ->group(function () {
        Route::get('/dashboard',
            [AdminDashboardController::class, 'index'])->name('dashboard');
        Route::get('/dashboard/orders.json',
            [AdminDashboardController::class, 'ordersJson'])->name('orders.json');
        Route::patch('/orders/{order}/approve',
            [AdminDashboardController::class, 'approve'])->name('orders.approve');
    });
```

All admin routes now require BOTH 'auth' (logged in) AND 'admin' (role = admin) middleware. The *prefix('admin')* means every URL starts with /admin. The *name('admin.')* prefixes route names with *admin.* - so the dashboard is **route('admin.dashboard')**.

Step 3 - The admin Blade view

Create **resources/views/admin/dashboard.blade.php**:

```
<x-app-layout>
    <x-slot name="header">
        <h2 class="font-bold text-xl">All Orders (Admin)</h2>
    </x-slot>

    <div class="container py-4">
        <div class="d-flex justify-content-between align-items-center mb-3">
            <h3>All Orders</h3>
            <span class="badge bg-success-subtle text-success border border-success">
                Live - refreshing every 5s
            </span>
        </div>

        <div class="table-responsive">
            <table class="table table-striped align-middle">
                <thead class="table-dark">
                    <tr>
                        <th>#</th><th>Customer</th><th>Pizza</th><th>Qty</th>
                        <th>Total</th><th>Placed</th><th>Status</th><th>Action</th>
                    </tr>
                </thead>
                <tbody id="ordersBody">
                    <tr><td colspan="8" class="text-center">Loading...</td></tr>
                </tbody>
            </table>
        </div>
    </div>

    @push('scripts')
    <script>
    const csrfToken = '{{ csrf_token() }}';

    async function loadOrders() {
        const res = await fetch('{{ route('admin.orders.json') }}');
        const data = await res.json();
        const body = document.getElementById('ordersBody');

        body.innerHTML = data.map(o => `
            <tr>
                <td>${o.id}</td>
                <td>${o.customer}</td>
                <td>${o.pizza}</td>
                <td>${o.quantity}</td>
                <td>Rs. ${o.total}</td>
                <td>${o.placed_at}</td>
                <td>
                  <span class="badge bg-${o.status === 'approved'
                    ? 'success' : 'warning text-dark'}">${o.status}</span>
                </td>
                <td>${
                    o.status === 'pending'
                    ? `<button class="btn btn-sm btn-success"
                         onclick="approve(${o.id})">Approve</button>`
                    : '-'
                }</td>
            </tr>
        `).join('');
    }

    async function approve(id) {
```

```
        await fetch(`/admin/orders/${id}/approve`, {
            method: 'PATCH',
            headers: {
                'X-CSRF-TOKEN': csrfToken,
                'Accept': 'application/json',
            },
        });
        loadOrders();
    }

    loadOrders();
    setInterval(loadOrders, 5000);
    </script>
    @endpush
</x-app-layout>
```

> **TIP**
>
> The **X-CSRF-TOKEN** header is Laravel's protection against Cross-Site Request Forgery. Without it, the PATCH request would be rejected with a 419 error. Every state-changing AJAX request to a Laravel app needs this header.

Step 4 - End-to-end test

1. Open two browsers (or one normal, one Incognito).
2. Browser 1: log in as your test customer. Visit /menu, place an order.
3. Browser 2: log in as **** with password **admin123**.
4. Browser 2 lands on /admin/dashboard. Within 5 seconds the new order appears.
5. Click **Approve** in browser 2.
6. Switch to browser 1 - within 5 seconds the badge turns green.
7. Loop completed. You just built a real-time approval system.

Chapter 13 - Blade Components and the Layout File

By now you have noticed every Blade view starts with **<x-app-layout>**. That's a Blade **component** - Laravel's way of letting you build reusable UI pieces. They are like lego blocks for HTML.

Why components matter

- Define your navbar once - every page uses it via <x-app-layout>.
- Define a button style once - reuse it everywhere as <x-button>.
- Change the look in one place, the whole site updates.
- Pass data to components like function arguments.
- Keep individual page views tiny and readable.

The layout component (already exists)

Breeze created **resources/views/components/app-layout.blade.php**. Open it - it imports the navigation and renders any content you pass between **<x-app-layout>** tags via **{{ $slot }}**.

```
<!-- Simplified version -->
<!DOCTYPE html>
<html>
<head>
    <title>{{ config('app.name') }}</title>
    <link href="https://cdn.jsdelivr.net/npm/[email protected]/dist/css/bootstrap.min.css"
          rel="stylesheet">
</head>
<body>
    @include('layouts.navigation')

    @if (isset($header))
        <header class="bg-light py-3 mb-4 border-bottom">
            <div class="container">{{ $header }}</div>
        </header>
    @endif

    <main>{{ $slot }}</main>

    @stack('scripts')
</body>
</html>
```

Step 1 - Make a custom button component

```
php artisan make:component PrimaryButton --view
```

Open **resources/views/components/primary-button.blade.php** and write:

```
@props(['type' => 'submit'])

<button type="{{ $type }}"
        {{ $attributes->merge(['class' => 'btn btn-danger fw-bold']) }}>
    {{ $slot }}
</button>
```

Now anywhere in your views you can write:

```
<x-primary-button>Order Now</x-primary-button>

<x-primary-button class="btn-lg w-100">
    Place Order
</x-primary-button>
```

> **NOTE**
>
> **@props** declares the variables a component accepts. **$attributes->merge([...])** combines default classes with any extra classes the caller passes. Components feel a lot like React or Vue components - except they render server-side, no JavaScript needed.

Step 2 - A pizza card component

The menu page repeats the same card markup twice. Let's extract it:

```
php artisan make:component PizzaCard --view
```

resources/views/components/pizza-card.blade.php:

```
@props(['pizza'])

<div class="card h-100 shadow-sm">
    <div class="card-body">
        <h3 class="card-title">{{ $pizza->name }}</h3>
        <p class="text-muted">{{ $pizza->description }}</p>
        <p class="h4 text-danger fw-bold">
            Rs. {{ number_format($pizza->price) }}
        </p>

        <form method="POST" action="{{ route('orders.store') }}"
              class="d-flex gap-2">
            @csrf
            <input type="hidden" name="pizza_id" value="{{ $pizza->id }}">
            <input type="number" name="quantity"
                   value="1" min="1" max="10"
                   class="form-control" style="width: 80px;">
            <x-primary-button class="flex-grow-1">Order</x-primary-button>
        </form>
    </div>
</div>
```

Now the menu view's loop becomes much shorter:

```
<div class="row g-4">
    @foreach ($pizzas as $pizza)
        <div class="col-md-6">
            <x-pizza-card :pizza="$pizza" />
        </div>
    @endforeach
</div>
```

> **TIP**
>
> The **:pizza="$pizza"** syntax (with the colon) means "pass this PHP variable". Without the colon, you would be passing the literal string '$pizza'. Small detail, big difference.

Chapter 14 - Git, GitHub, and Deploying to Production

Your project works locally. Now we save the code to GitHub (so you can show it on your CV) and deploy it to a real server (so the whole world can order pizza from you).

Step 1 - Set up Git

Laravel ships with a sensible **.gitignore** already - it excludes *vendor/*, *node_modules/*, *.env*, and other things that should never be committed. Just initialise the repo:

```
cd ~/code/pizza-app
git init
git add .
git commit -m "Initial commit - Laravel 13 pizza app"
```

Step 2 - Create a GitHub repository

1. Visit **github.com** and sign in.
2. Click **+** in the top right then **New repository**.
3. Name it **pizza-laravel**.
4. Public or Private - your choice.
5. Do NOT initialise with a README - we already have files.
6. Click **Create repository**.

Step 3 - Push your code

GitHub shows you the exact commands. They look like this:

```
git remote add origin https://github.com/YOUR_USERNAME/pizza-laravel.git
git branch -M main
git push -u origin main
```

> **NOTE**
>
> GitHub no longer accepts password login. You need a **Personal Access Token**: Settings -> Developer settings -> Personal access tokens -> Generate. Paste the token where it asks for a password.

Step 4 - Daily Git workflow

```
git status                  # what changed?
git add .                   # stage everything
git commit -m "Add admin reports page"
git push                    # send to GitHub
```

Step 5 - A README for your portfolio

Replace the auto-generated **README.md** with something employers will appreciate:

```
# Pizza Ordering System (Laravel 13)

A full-stack pizza ordering web app. Customers register, browse a menu,
and place orders. An admin dashboard shows incoming orders and approves
them with one click. Customer dashboards update live via AJAX polling.

Built as a learning project from a tutorial at egotechworld.com.

## Tech Stack
- Laravel 13 + PHP 8.3
- MySQL 8
- Eloquent ORM, Form Requests, Middleware, Blade Components
- Bootstrap 5 (via CDN) + a touch of Tailwind from Breeze
- Vanilla JS (fetch + setInterval) for live updates

## Features
- Customer registration & login (Laravel Breeze)
- Role-based admin area (custom AdminMiddleware)
- Two pizzas seeded from a database seeder
- Live status updates without page reload
- One-click order approval

## Setup
\`\`\`bash
composer install
npm install && npm run build
cp .env.example .env
php artisan key:generate
# update DB_DATABASE / DB_USERNAME / DB_PASSWORD in .env
php artisan migrate --seed
php artisan serve
\`\`\`

Then visit http://localhost:8000 and register a customer account.
Admin login: [email protected] / admin123

## Author
Your Name - github.com/yourusername
```

Step 6 - Deploying to a real server

Laravel apps need a few things shared hosts may not have. Your options, easiest to most powerful:

Host	Cost	Difficulty	Notes
LankaHost / Hostinger	Cheap	Easy	Cheap shared hosting works for small Laravel apps. PHP 8.3 required.
DigitalOcean droplet	$4/mo+	Medium	Full VPS. More control, you manage the server yourself.
Laravel Forge + DO	$12/mo+	Easy after setup	Forge handles server config. Industry standard.
Laravel Cloud	Pay-as-you-go	Easiest	Official platform. Push to deploy. Built for Laravel.

Generic shared-host steps

1. Upload the project via FTP or cPanel File Manager.
2. Create a MySQL database in cPanel; note credentials.
3. SSH in (or use cPanel terminal). Run **composer install --no-dev**.
4. Run **npm install && npm run build** to compile assets.
5. Copy **.env.example** to **.env**; fill in real credentials and APP_URL.
6. Run **php artisan key:generate**.
7. Run **php artisan migrate --seed**.
8. Point your domain's document root to the project's **public/** folder.
9. Run **php artisan config:cache && php artisan route:cache** for speed.

> **WARNING**
>
> Always set **APP_DEBUG=false** in production .env. With debug on, any error page shows your full code, env variables, and stack trace - a goldmine for attackers.

Chapter 15 - Testing, Common Bugs, and Where to Go Next

Manual testing checklist

Walk through this list before declaring the project done. Tick each with [v] or [x] - fix any [x] before moving on.

Area	What to test
Registration	New user can register. Duplicate email is rejected.
Login	Wrong password is refused. Right password redirects correctly.
Sessions	Closing the browser and re-opening keeps you logged in.
Menu access	Visiting /menu while logged out redirects to login.
Order form	Quantity 0 or 999 is rejected with a friendly error.
Customer dashboard	Status badge changes within 5 seconds of admin approval.
Admin guard	Visiting /admin/dashboard as a customer returns 403 Forbidden.
Admin approval	Approve button works and disappears after click.
CSRF	AJAX approve still works after token refresh.
Logout	Logout invalidates the session - you can't go back.

Automated tests with Pest

Laravel ships with PHPUnit, but most 2026 projects use **Pest** - a friendlier wrapper. Install it:

```
composer require pestphp/pest --dev --with-all-dependencies
php artisan pest:install
```

Then write a test in **tests/Feature/OrderTest.php**:

```
<?php

use App\Models\User;
use App\Models\Pizza;

it('lets a logged-in customer place an order', function () {
    $user = User::factory()->create();
    $pizza = Pizza::create([
        'name' => 'Test Pizza',
        'description' => 'A test pizza',
        'price' => 1000,
    ]);

    $response = $this->actingAs($user)->post('/orders', [
        'pizza_id' => $pizza->id,
        'quantity' => 2,
    ]);
```

```
    $response->assertRedirect();
    expect($user->orders()->count())->toBe(1);
});

it('rejects unauthenticated order requests', function () {
    $pizza = Pizza::create([
        'name' => 'Test',
        'description' => 'd',
        'price' => 1000,
    ]);

    $response = $this->post('/orders', [
        'pizza_id' => $pizza->id,
        'quantity' => 1,
    ]);

    $response->assertRedirect('/login');
});
```

Run the tests:

```
php artisan test
```

> **TIP**
>
> Tests are not just for big companies. Even one or two tests around critical features (like 'orders save correctly') catch a lot of regressions when you change code six months later. Write at least one test for every feature.

Common Laravel bugs and how to fix them

Symptom	Likely cause	Fix
419 Page Expired	Missing CSRF token in form or AJAX.	Add **@csrf** in forms; X-CSRF-TOKEN header in AJAX.
Class 'App\Models\X' not found	Wrong namespace or forgot to import.	Add **use App\Models\X;** at the top.
Database connection refused	MySQL not running, or wrong port in .env.	Start MySQL; check DB_HOST/PORT.
Method does not exist on null	Loaded an Eloquent relationship that's empty.	Use **?->** or check for null first.
Vite manifest error	Forgot to run npm run dev / build.	**npm install && npm run dev**.
403 even though I'm admin	role column not set in DB.	phpMyAdmin -> users -> set role = 'admin'.
Page just shows 'No application encryption key'	Missing APP_KEY in .env.	Run **php artisan key:generate**.

Where to go next - 10 ideas

1. **More pizzas** - admin form to create/edit/delete pizzas (full CRUD with policies).
2. **Image upload** - photo per pizza, stored in *storage/app/public*.
3. **Shopping cart** - session-stored cart, single 'Place Order' button at the end.
4. **Order cancellation** - customer cancels pending orders within 5 minutes.
5. **Notifications** - Laravel Notifications send email when admin approves.
6. **Search and filter** - Eloquent scopes for filtering the menu.
7. **Pagination** - *$orders->paginate(10)* on the admin dashboard.
8. **Reverb WebSockets** - replace polling with true push for instant updates.
9. **Reports** - Chart.js bar chart of revenue per day.
10. **Stripe payments** - Laravel Cashier integration to take real money.

Final words

You have built a real Laravel application from **composer create-project** all the way to a deployed live site. Along the way you learned MVC, Eloquent, migrations, seeders, Blade components, middleware, Form Requests, route model binding, and AJAX polling. That is most of a junior Laravel developer's daily toolkit.

Now **show it**. Share the GitHub link on your CV. Post a screenshot on LinkedIn. Write a blog post on egotechworld.com about what you learned (we welcome guest posts). The project is the credential; talking about it is what gets you hired.

Good luck, and welcome to the Laravel community.

Quick Reference Card - Laravel 13

Most-used Artisan commands

Command	Does
php artisan serve	Start dev server on http://localhost:8000.
php artisan make:model Foo -mfs	Create model + migration + factory + seeder.
php artisan make:controller FooController	Create a controller.
php artisan make:request StoreFooRequest	Create a Form Request for validation.
php artisan make:middleware FooMiddleware	Create middleware.
php artisan migrate	Run pending migrations.
php artisan migrate:fresh --seed	Drop all tables, re-migrate, re-seed.
php artisan db:seed	Run seeders only.
php artisan route:list	Show every route in the app.
php artisan tinker	Interactive PHP shell with your app loaded.
php artisan test	Run all tests.

Project structure (final)

```
pizza-app/
+-- app/
|   +-- Http/
|   |   +-- Controllers/
|   |   |   +-- Auth/                # Breeze
|   |   |   +-- Admin/
|   |   |   |   `-- DashboardController.php
|   |   |   +-- MenuController.php
|   |   |   +-- OrderController.php
|   |   |   `-- CustomerDashboardController.php
|   |   +-- Middleware/
|   |   |   `-- AdminMiddleware.php
|   |   `-- Requests/
|   |       `-- StoreOrderRequest.php
|   `-- Models/
|       +-- User.php
|       +-- Pizza.php
|       `-- Order.php
+-- database/
|   +-- migrations/
|   |   +-- ..._create_users_table.php
|   |   +-- ..._add_role_to_users_table.php
|   |   +-- ..._create_pizzas_table.php
|   |   `-- ..._create_orders_table.php
|   `-- seeders/
|       +-- DatabaseSeeder.php
|       `-- PizzaSeeder.php
+-- resources/views/
|   +-- layouts/app.blade.php
|   +-- components/
|   |   +-- pizza-card.blade.php
|   |   `-- primary-button.blade.php
|   +-- menu/index.blade.php
|   +-- dashboard/customer.blade.php
|   `-- admin/dashboard.blade.php
+-- routes/
|   +-- web.php
|   `-- auth.php
+-- .env
`-- README.md
```

Key URLs after running php artisan serve

Page	URL
Home	http://localhost:8000/
Register	http://localhost:8000/register
Login	http://localhost:8000/login
Menu	http://localhost:8000/menu
Customer dashboard	http://localhost:8000/dashboard
Admin dashboard	http://localhost:8000/admin/dashboard

Default admin credentials (from the seeder)

```
Email:    [email protected]
Password: admin123
```

> **WARNING**
>
> Change the admin password before going to production. Edit the seeder, or run *php artisan tinker* and update the user.

Where to learn more

- **laravel.com/docs/13.x** - the official docs. Surprisingly readable.
- **laracasts.com** - 1000+ video tutorials. The Laravel community lives here.
- **laravel-news.com** - weekly newsletter and articles.
- **egotechworld.com** - more Sinhala and English coding tutorials, free project source code downloads, and AI tools.

Closing note from egotechworld.com

If this tutorial helped you, share it with a friend who's learning to code. Bookmark **egotechworld.com** for more PHP, Python, Laravel, and Django tutorials. We post new content regularly and welcome guest articles from learners like you.

Happy coding!