Chapter 1. Why Laravel?

In the early days of the dynamic web, writing a web application looked a lot different than it does today. Developers then were responsible for writing the code for not just the unique business logic of our applications, but also each of the components that are so common across sites—user authentication, input validation, database access, templating, and more.

Today, programmers have dozens of application development frameworks and thousands of components and libraries easily accessible. It's a common refrain among programmers that, by the time you learn one framework, three newer (and purportedly better) frameworks have popped up intending to replace it.

"Just because it's there" might be a valid justification for climbing a mountain, but there are better reasons to choose to use a specific framework—or to use a framework at all. It's worth asking the question, why frameworks? More specifically, why Laravel?

Why Use a Framework?

It's easy to see why it's beneficial to use the individual components, or packages, that are available to PHP developers. With packages, someone else is responsible for developing and maintaining an isolated piece of code that has a well-defined job, and in theory that person has a deeper understanding of this single component than you have time to have.

Frameworks like Laravel—and Symfony, Lumen, and Slim—prepackage a collection of third-party components together with custom framework "glue" like configuration files, service providers, prescribed directory structures, and application bootstraps. So, the benefit of using a framework in general is that someone has made decisions not just about individual components for you, but also about *how those components should fit together*.

"I'll Just Build It Myself"

Let’s say you start a new web app without the benefit of a framework. Where do you begin? Well, it should probably route HTTP requests, so you now need to evaluate all of the HTTP request and response libraries available and pick one. Then you’ll have to pick a router. Oh, and you’ll probably need to set up some form of routes configuration file. What syntax should it use? Where should it go? What about controllers? Where do they live, and how are they loaded? Well, you probably need a dependency injection container to resolve the controllers and their dependencies. But which one?

Furthermore, if you do take the time to answer all those questions and successfully create your application, what’s the impact on the next developer? What about when you have four such custom framework–based applications, or a dozen, and you have to remember where the controllers live in each, or what the routing syntax is?

Consistency and Flexibility

Frameworks address this issue by providing a carefully considered answer to the question “Which component should we use here?” and ensuring that the particular components chosen work well together. Additionally, frameworks provide conventions that reduce the amount of code a developer new to the project has to understand—if you understand how routing works in one Laravel project, for example, you understand how it works in all Laravel projects.

When someone prescribes rolling your own framework for each new project, what they’re really advocating is the ability to *control* what does and doesn’t go into your application’s foundation. That means the best frameworks will not only provide you with a solid foundation, but also give you the freedom to customize to your heart’s content. And this, as I’ll show you in the rest of this book, is part of what makes Laravel so special.

A Short History of Web and PHP Frameworks

An important part of being able to answer the question “Why Laravel?” is understanding Laravel’s history—and understanding what came before it. Prior to Laravel’s rise in popularity, there were a variety of frameworks and other movements in PHP and other web development spaces.

Ruby on Rails

David Heinemeier Hansson released the first version of Ruby on Rails in 2004, and it’s been hard to find a web application framework since then that hasn’t been influenced by Rails in some way.

Rails popularized MVC, RESTful JSON APIs, convention over configuration, ActiveRecord, and many more tools and conventions that had a profound influence on the way web developers approached their applications—especially with regard to rapid application development.

The Influx of PHP Frameworks

It was clear to most developers that Rails and similar web application frameworks were the wave of the future, and PHP frameworks, including those admittedly imitating Rails, started popping up quickly.

CakePHP was the first in 2005, and it was soon followed by Symfony, CodeIgniter, Zend Framework, and Kohana (a CodeIgniter fork). Yii arrived in 2008, and Aura and Slim in 2010. 2011 brought FuelPHP and Laravel, both of which were not quite CodeIgniter offshoots, but instead proposed as alternatives.

Some of these frameworks were more Rails-y, focusing on database object-relational mappers (ORMs), MVC structures, and other tools targeting rapid development. Others, like Symfony and Zend, focused more on enterprise design patterns and ecommerce.

The Good and the Bad of CodeIgniter

CakePHP and CodeIgniter were the two early PHP frameworks that were most open about how much their inspiration was drawn from Rails. CodeIgniter quickly rose to fame and by 2010 was arguably the most popular of the independent PHP frameworks.

CodeIgniter was simple, easy to use, and boasted amazing documentation and a strong community. But its use of modern technology and patterns advanced slowly; and as the framework world grew and PHP’s tooling advanced, CodeIgniter started falling behind in terms of both technological advances and out-of-the-box features. Unlike many other frameworks, CodeIgniter was managed by a company, and it was slow to catch up with PHP 5.3’s newer features like namespaces and the moves to GitHub and later Composer. It was in 2010 that Taylor Otwell, Laravel’s creator, became dissatisfied enough with CodeIgniter that he set off to write his own framework.

Laravel 1, 2, and 3

The first beta of Laravel 1 was released in June 2011, and it was written completely from scratch. It featured a custom ORM (Eloquent); closure-based routing (inspired by Ruby Sinatra); a module system for extension; and helpers for forms, validation, authentication, and more.

Early Laravel development moved quickly, and Laravel 2 and 3 were released in November 2011 and February 2012, respectively. They introduced controllers, unit testing, a command-line tool, an inversion of control (IoC) container, Eloquent relationships, and migrations.

Laravel 4

With Laravel 4, Taylor rewrote the entire framework from the ground up. By this point Composer, PHP’s now-ubiquitous package manager, was showing signs of becoming an industry standard, and Taylor saw the value of rewriting the framework as a collection of components, distributed and bundled together by Composer.

Taylor developed a set of components under the code name *Illuminate* and, in May 2013, released Laravel 4 with an entirely new structure. Instead of bundling the majority of its code as a download, Laravel now pulled in the majority of its components from Symfony (another framework that released its components for use by others) and the Illuminate components through Composer.

Laravel 4 also introduced queues, a mail component, facades, and database seeding. And because Laravel was now relying on Symfony components, it was announced that Laravel would be mirroring (not exactly, but soon after) the six-monthly release schedule Symfony follows.

Laravel 5

Laravel 4.3 was scheduled to release in November 2014, but as development progressed it became clear that the significance of its changes merited a major release, and Laravel 5 was released in February 2015.

Laravel 5 featured a revamped directory structure, removal of the form and HTML helpers, the introduction of the contract interfaces, a spate of new views, Socialite for social media authentication, Elixir for asset compilation, Scheduler to simplify cron, dotenv for simplified environment management, form requests, and a brand new REPL (read–evaluate–print loop). Since then it’s grown in features and maturity, but there have been no major changes like in previous versions.

What’s So Special About Laravel?

So what is it that sets Laravel apart? Why is it worth having more than one PHP framework at any time? They all use components from Symfony anyway, right? Let’s talk a bit about what makes Laravel “tick.”

The Philosophy of Laravel

You only need to read through the Laravel marketing materials and READMEs to start seeing its values. Taylor uses light-related words like “Illuminate” and “Spark.” And then there are these: “Artisans.” “Elegant.” Also, these: “Breath of fresh air.” “Fresh start.” And finally: “Rapid.” “Warp speed.”

The two most strongly communicated values of the framework are to increase developer speed and developer happiness. Taylor has described the “Artisan” language as intentionally contrasting against more utilitarian values. You can see the genesis of this sort of thinking in his 2011 question on StackExchange in which he stated, “Sometimes I spend ridiculous amounts of time (hours) agonizing over making code ‘look pretty’”—just for the sake of a better experience of looking at the code itself. And he’s often talked about the value of making it easier and quicker for developers to take their ideas to fruition, getting rid of unnecessary barriers to creating great products.

Laravel is, at its core, about equipping and enabling developers. Its goal is to provide clear, simple, and beautiful code and features that help developers quickly learn, start, and develop, and write code that's simple, clear, and lasting.

The concept of targeting developers is clear across Laravel materials. "Happy developers make the best code" is written in the documentation. "Developer happiness from download to deploy" was the unofficial slogan for a while. Of course, any tool or framework will say it wants developers to be happy. But having developer happiness as a *primary* concern, rather than secondary, has had a huge impact on Laravel's style and decision-making progress. Where other frameworks may target architectural purity as their primary goal, or compatibility with the goals and values of enterprise development teams, Laravel's primary focus is on serving the individual developer. That doesn't mean you can't write architecturally pure or enterprise-ready applications in Laravel, but it won't have to be at the expense of the readability and comprehensibility of your codebase.

How Laravel Achieves Developer Happiness

Just saying you want to make developers happy is one thing. Doing it is another, and it requires you to question what in a framework is most likely to make developers unhappy and what is most likely to make them happy. There are a few ways Laravel tries to make developers' lives easier.

First, Laravel is a rapid application development framework. That means it focuses on a shallow (easy) learning curve and on minimizing the steps between starting a new app and publishing it. All of the most common tasks in building web applications, from database interactions to authentication to queues to email to caching, are made simpler by the components Laravel provides. But Laravel's components aren't just great on their own; they provide a consistent API and predictable structures across the entire framework. That means that, when you're trying something new in Laravel, you're more than likely going to end up saying, "… and it just works."

This doesn’t end with the framework itself, either. Laravel provides an entire ecosystem of tools for building and launching applications. You have Homestead and Valet for local development, Forge for server management, and Envoyer for advanced deployment. And there’s a suite of add-on packages: Cashier for payments and subscriptions, Echo for WebSockets, Scout for search, Passport for API authentication, Dusk for frontend testing, Socialite for social login, Horizon for monitoring queues, Nova for building admin panels, and Spark to bootstrap your SaaS. Laravel is trying to take the repetitive work out of developers’ jobs so they can do something unique.

Next, Laravel focuses on “convention over configuration”—meaning that if you’re willing to use Laravel’s defaults, you’ll have to do much less work than with other frameworks that require you to declare all of your settings even if you’re using the recommended configuration. Projects built on Laravel take less time than those built on most other PHP frameworks.

Laravel also focuses deeply on simplicity. It’s possible to use dependency injection and mocking and the Data Mapper pattern and repositories and Command Query Responsibility Segregation and all sorts of other more complex architectural patterns with Laravel, if you want. But while other frameworks might suggest using those tools and structures on every project, Laravel and its documentation and community lean toward starting with the simplest possible implementation—a global function here, a facade there, ActiveRecord over there. This allows developers to create the simplest possible application to solve for their needs, without limiting its usefulness in complex environments.

An interesting source of how Laravel is different from other PHP frameworks is that its creator and its community are more connected to and inspired by Ruby and Rails and functional programming languages than by Java. There’s a strong current in modern PHP to lean toward verbosity and complexity, embracing the more Java-esque aspects of PHP. But Laravel tends to be on the other side, embracing expressive, dynamic, and simple coding practices and language features.

The Laravel Community

If this book is your first exposure to the Laravel community, you have something special to look forward to. One of the distinguishing elements of Laravel, which has contributed to its growth and success, is the welcoming, teaching community that surrounds it. From Jeffrey Way's Laracasts video tutorials to Laravel News to Slack and IRC and Discord channels, from Twitter friends to bloggers to podcasts to the Laracon conferences, Laravel has a rich and vibrant community full of folks who've been around since day one and folks who are just starting their own "day one." And this isn't an accident:

From the very beginning of Laravel, I've had this idea that all people want to feel like they are part of something. It's a natural human instinct to want to belong and be accepted into a group of other like-minded people. So, by injecting personality into a web framework and being really active with the community, that type of feeling can grow in the community.

Taylor Otwell, *Product and Support interview*

Taylor understood from the early days of Laravel that a successful open source project needed two things: good documentation and a welcoming community. And those two things are now hallmarks of Laravel.

How It Works

Up until now, everything I've shared here has been entirely abstract. What about the code, you ask? Let's dig into a simple application (Example 1-1) so you can see what working with Laravel day to day is actually like.

Example 1-1. "Hello, World" in routes/web.php

```php
<?php

Route::get('/', function () {
    return 'Hello, World!';
});
```

The simplest possible action you can take in a Laravel application is to define a route and return a result any time someone visits that route. If you initialize a brand new Laravel application on your machine, define the route in Example 1-1, and then serve the site from the *public* directory, you'll have a fully functioning "Hello, World" example (see Figure 1-1).

Figure 1-1. Returning “Hello, World!” with Laravel

It looks very similar with controllers, as you can see in Example 1-2.

Example 1-2. “Hello, World” with controllers

```
// File: routes/web.php
<?php

Route::get('/', 'WelcomeController@index');
// File: app/Http/Controllers/WelcomeController.php
<?php

namespace App\Http\Controllers;

class WelcomeController extends Controller
{
    public function index()
    {
        return 'Hello, World!';
    }
}
```

And if you’re storing your greetings in a database, it’ll also look pretty similar (see Example 1-3).

Example 1-3. Multigreeting “Hello, World” with database access

```
// File: routes/web.php
<?php

use App\Greeting;

Route::get('create-greeting', function () {
    $greeting = new Greeting;
    $greeting->body = 'Hello, World!';
```

```
	$greeting->save();
});

Route::get('first-greeting', function () {
	return Greeting::first()->body;
});
// File: app/Greeting.php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Greeting extends Model
{
	//
}
// File: database/migrations/2015_07_19_010000_create_greetings_table.php
<?php

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

class CreateGreetingsTable extends Migration
{
	public function up()
	{
		Schema::create('greetings', function (Blueprint $table) {
			$table->bigIncrements('id');
			$table->string('body');
			$table->timestamps();
		});
	}

	public function down()
	{
		Schema::dropIfExists('greetings');
	}
}
```

Example 1-3 might be a bit overwhelming, and if so, just skip over it. You'll learn about everything that's happening here in later chapters, but you can already see that with just a few lines of code, you can set up database migrations and models and pull records out. It's just that simple.

Why Laravel?

So—why Laravel?

Because Laravel helps you bring your ideas to reality with no wasted code, using modern coding standards, surrounded by a vibrant community, with an empowering ecosystem of tools.

And because you, dear developer, deserve to be happy.