

Installation

- [Meet Laravel](#)
 - [Why Laravel?](#)
- [Creating a Laravel Application](#)
 - [Installing PHP and the Laravel Installer](#)
 - [Creating an Application](#)
- [Initial Configuration](#)
 - [Environment Based Configuration](#)
 - [Databases and Migrations](#)
 - [Directory Configuration](#)
- [Installation Using Herd](#)
 - [Herd on macOS](#)
 - [Herd on Windows](#)
- [IDE Support](#)
- [Next Steps](#)
 - [Laravel the Full Stack Framework](#)
 - [Laravel the API Backend](#)

Meet Laravel

Laravel is a web application framework with expressive, elegant syntax. A web framework provides a structure and starting point for creating your application, allowing you to focus on creating something amazing while we sweat the details.

Laravel strives to provide an amazing developer experience while providing powerful features such as thorough dependency injection, an expressive database abstraction layer, queues and scheduled jobs, unit and integration testing, and more.

Whether you are new to PHP web frameworks or have years of experience, Laravel is a framework that can grow with you. We'll help you take your first steps as a web developer or give you a boost as you take your expertise to the next level. We can't wait to see what you build.

Why Laravel?

There are a variety of tools and frameworks available to you when building a web application. However, we believe Laravel is the best choice for building modern, full-stack web applications.

A Progressive Framework

We like to call Laravel a "progressive" framework. By that, we mean that Laravel grows with you. If you're just taking your first steps into web development, Laravel's vast library of documentation, guides, and [video tutorials](#) will help you learn the ropes without becoming overwhelmed.

If you're a senior developer, Laravel gives you robust tools for [dependency injection](#), [unit testing](#), [queues](#), [real-time events](#), and more. Laravel is fine-tuned for building professional web applications and ready to handle enterprise work loads.

A Scalable Framework

Laravel is incredibly scalable. Thanks to the scaling-friendly nature of PHP and Laravel's built-in support for fast, distributed cache systems like Redis, horizontal scaling with Laravel is a breeze. In fact, Laravel applications have been easily scaled to handle hundreds of millions of requests per month.

Need extreme scaling? Platforms like [Laravel Cloud](#) allow you to run your Laravel application at nearly limitless scale.

A Community Framework

Laravel combines the best packages in the PHP ecosystem to offer the most robust and developer friendly framework available. In addition, thousands of talented developers from around the world have [contributed to the framework](#). Who knows, maybe you'll even become a Laravel contributor.

Creating a Laravel Application

Installing PHP and the Laravel Installer

Before creating your first Laravel application, make sure that your local machine has [PHP](#), [Composer](#), and [the Laravel installer](#) installed. In addition, you should install either [Node and NPM](#) or [Bun](#) so that you can compile your application's frontend assets.

If you don't have PHP and Composer installed on your local machine, the following commands will install PHP, Composer, and the Laravel installer on macOS, Windows, or Linux:

```
/bin/bash -c "$(curl -fsSL https://php.new/install/mac/8.4)"
```

```
# Run as administrator...
Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-Object
System.Net.WebClient).DownloadString('https://php.new/install/windows/8.4'))
```

```
/bin/bash -c "$(curl -fsSL https://php.new/install/linux/8.4)"
```

After running one of the commands above, you should restart your terminal session. To update PHP, Composer, and the Laravel installer after installing them via [php.new](#), you can re-run the command in your terminal.

If you already have PHP and Composer installed, you may install the Laravel installer via Composer:

```
composer global require laravel/installer
```

[!NOTE] For a fully-featured, graphical PHP installation and management experience, check out [Laravel Herd](#).

Creating an Application

After you have installed PHP, Composer, and the Laravel installer, you're ready to create a new Laravel application. The Laravel installer will prompt you to select your preferred testing framework, database, and starter kit:

```
laravel new example-app
```

Once the application has been created, you can start Laravel's local development server, queue worker, and Vite development server using the `dev` Composer script:

```
cd example-app
npm install && npm run build
composer run dev
```

Once you have started the development server, your application will be accessible in your web browser at <http://localhost:8000>. Next, you're ready to [start taking your next steps into the Laravel ecosystem](#). Of course, you may also want to [configure a database](#).

[!NOTE] If you would like a head start when developing your Laravel application, consider using one of our [starter kits](#). Laravel's starter kits provide backend and frontend authentication scaffolding for your new Laravel application.

Initial Configuration

All of the configuration files for the Laravel framework are stored in the `config` directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

Laravel needs almost no additional configuration out of the box. You are free to get started developing! However, you may wish to review the `config/app.php` file and its documentation. It contains several options such as `url` and `locale` that you may wish to change according to your application.

Environment Based Configuration

Since many of Laravel's configuration option values may vary depending on whether your application is running on your local machine or on a production web server, many important configuration values are defined using the `.env` file that exists at the root of your application.

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would be exposed.

[!NOTE] For more information about the `.env` file and environment based configuration, check out the full [configuration documentation](#).

Databases and Migrations

Now that you have created your Laravel application, you probably want to store some data in a database. By default, your application's `.env` configuration file specifies that Laravel will be interacting with an SQLite database.

During the creation of the application, Laravel created a `database/database.sqlite` file for you, and ran the necessary migrations to create the application's database tables.

If you prefer to use another database driver such as MySQL or PostgreSQL, you can update your `.env` configuration file to use the appropriate database. For example, if you wish to use MySQL, update your `.env` configuration file's `DB_*` variables like so:

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

If you choose to use a database other than SQLite, you will need to create the database and run your application's [database migrations](#):

```
php artisan migrate
```

[!NOTE] If you are developing on macOS or Windows and need to install MySQL, PostgreSQL, or Redis locally, consider using [Herd Pro](#) or [DBngin](#).

Directory Configuration

Laravel should always be served out of the root of the "web directory" configured for your web server. You should not attempt to serve a Laravel application out of a subdirectory of the "web directory". Attempting to do so could expose sensitive files present within your application.

Installation Using Herd

[Laravel Herd](#) is a blazing fast, native Laravel and PHP development environment for macOS and Windows. Herd includes everything you need to get started with Laravel development, including PHP and Nginx.

Once you install Herd, you're ready to start developing with Laravel. Herd includes command line tools for [php](#), [composer](#), [laravel](#), [expose](#), [node](#), [npm](#), and [nvm](#).

[!NOTE] [Herd Pro](#) augments Herd with additional powerful features, such as the ability to create and manage local MySQL, Postgres, and Redis databases, as well as local mail viewing and log monitoring.

Herd on macOS

If you develop on macOS, you can download the Herd installer from the [Herd website](#). The installer automatically downloads the latest version of PHP and configures your Mac to always run [Nginx](#) in the background.

Herd for macOS uses [dnsmasq](#) to support "parked" directories. Any Laravel application in a parked directory will automatically be served by Herd. By default, Herd creates a parked directory at `~/Herd` and you can access any Laravel application in this directory on the `.test` domain using its directory name.

After installing Herd, the fastest way to create a new Laravel application is using the Laravel CLI, which is bundled with Herd:

```
cd ~/Herd
laravel new my-app
cd my-app
herd open
```

Of course, you can always manage your parked directories and other PHP settings via Herd's UI, which can be opened from the Herd menu in your system tray.

You can learn more about Herd by checking out the [Herd documentation](#).

Herd on Windows

You can download the Windows installer for Herd on the [Herd website](#). After the installation finishes, you can start Herd to complete the onboarding process and access the Herd UI for the first time.

The Herd UI is accessible by left-clicking on Herd's system tray icon. A right-click opens the quick menu with access to all tools that you need on a daily basis.

During installation, Herd creates a "parked" directory in your home directory at `%USERPROFILE%\Herd`. Any Laravel application in a parked directory will automatically be served by Herd, and you can access any Laravel application in this directory on the `.test` domain using its directory name.

After installing Herd, the fastest way to create a new Laravel application is using the Laravel CLI, which is bundled with Herd. To get started, open Powershell and run the following commands:

```
cd ~\Herd
laravel new my-app
cd my-app
herd open
```

You can learn more about Herd by checking out the [Herd documentation for Windows](#).

IDE Support

You are free to use any code editor you wish when developing Laravel applications; however, [PhpStorm](#) offers extensive support for Laravel and its ecosystem, including [Laravel Pint](#).

In addition, the community maintained [Laravel Idea](#) PhpStorm plugin offers a variety of helpful IDE augmentations, including code generation, Eloquent syntax completion, validation rule completion, and more.

If you develop in [Visual Studio Code \(VS Code\)](#), the official [Laravel VS Code Extension](#) is now available. This extension brings Laravel-specific tools directly into your VS Code environment, enhancing productivity.

Next Steps

Now that you have created your Laravel application, you may be wondering what to learn next. First, we strongly recommend becoming familiar with how Laravel works by reading the following documentation:

- [Request Lifecycle](#)
- [Configuration](#)
- [Directory Structure](#)
- [Frontend](#)
- [Service Container](#)
- [Facades](#)

How you want to use Laravel will also dictate the next steps on your journey. There are a variety of ways to use Laravel, and we'll explore two primary use cases for the framework below.

Laravel the Full Stack Framework

Laravel may serve as a full stack framework. By "full stack" framework we mean that you are going to use Laravel to route requests to your application and render your frontend via [Blade templates](#) or a single-page application hybrid technology like [Inertia](#). This is the most common way to use the Laravel framework, and, in our opinion, the most productive way to use Laravel.

If this is how you plan to use Laravel, you may want to check out our documentation on [frontend development](#), [routing](#), [views](#), or the [Eloquent ORM](#). In addition, you might be interested in learning about community packages like [Livewire](#) and [Inertia](#). These packages allow you to use Laravel as a full-stack framework while enjoying many of the UI benefits provided by single-page JavaScript applications.

If you are using Laravel as a full stack framework, we also strongly encourage you to learn how to compile your application's CSS and JavaScript using [Vite](#).

[!NOTE] If you want to get a head start building your application, check out one of our official [application starter kits](#).

Laravel the API Backend

Laravel may also serve as an API backend to a JavaScript single-page application or mobile application. For example, you might use Laravel as an API backend for your [Next.js](#) application. In this context, you may use Laravel to provide [authentication](#) and data storage / retrieval for your application, while also taking advantage of Laravel's powerful services such as queues, emails, notifications, and more.

If this is how you plan to use Laravel, you may want to check out our documentation on [routing](#), [Laravel Sanctum](#), and the [Eloquent ORM](#).

Configuration

- [Introduction](#)
- [Environment Configuration](#)
 - [Environment Variable Types](#)
 - [Retrieving Environment Configuration](#)
 - [Determining the Current Environment](#)
 - [Encrypting Environment Files](#)
- [Accessing Configuration Values](#)
- [Configuration Caching](#)
- [Configuration Publishing](#)
- [Debug Mode](#)
- [Maintenance Mode](#)

Introduction

All of the configuration files for the Laravel framework are stored in the `config` directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

These configuration files allow you to configure things like your database connection information, your mail server information, as well as various other core configuration values such as your application URL and encryption key.

The `about` Command

Laravel can display an overview of your application's configuration, drivers, and environment via the `about` Artisan command.

```
php artisan about
```

If you're only interested in a particular section of the application overview output, you may filter for that section using the `--only` option:

```
php artisan about --only=environment
```

Or, to explore a specific configuration file's values in detail, you may use the `config:show` Artisan command:

```
php artisan config:show database
```

Environment Configuration

It is often helpful to have different configuration values based on the environment where the application is running. For example, you may wish to use a different cache driver locally than you do on your production server.

To make this a cinch, Laravel utilizes the [DotEnv](#) PHP library. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file that defines many common environment variables. During the Laravel installation process, this file will automatically be copied to `.env`.

Laravel's default `.env` file contains some common configuration values that may differ based on whether your application is running locally or on a production web server. These values are then read by the configuration files within the `config` directory using Laravel's `env` function.

If you are developing with a team, you may wish to continue including and updating the `.env.example` file with your application. By putting placeholder values in the example configuration file, other developers on your team can clearly see which environment variables are needed to run your application.

[!NOTE] Any variable in your `.env` file can be overridden by external environment variables such as server-level or system-level environment variables.

Environment File Security

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

However, it is possible to encrypt your environment file using Laravel's built-in [environment encryption](#). Encrypted environment files may be placed in source control safely.

Additional Environment Files

Before loading your application's environment variables, Laravel determines if an `APP_ENV` environment variable has been externally provided or if the `--env` CLI argument has been specified. If so, Laravel will attempt to load an `.env.[APP_ENV]` file if it exists. If it does not exist, the default `.env` file will be loaded.

Environment Variable Types

All variables in your `.env` files are typically parsed as strings, so some reserved values have been created to allow you to return a wider range of types from the `env()` function:

<code>.env</code> Value	<code>env()</code> Value
true	(bool) true
(true)	(bool) true
false	(bool) false
(false)	(bool) false
empty	(string) ""
(empty)	(string) ""
null	(null) null
(null)	(null) null

If you need to define an environment variable with a value that contains spaces, you may do so by enclosing the value in double quotes:

```
APP_NAME="My Application"
```

Retrieving Environment Configuration

All of the variables listed in the `.env` file will be loaded into the `$_ENV` PHP super-global when your application receives a request. However, you may use the `env` function to retrieve values from these variables in your configuration files. In fact, if you review the Laravel configuration files, you will notice many of the options are already using this function:

```
'debug' => env('APP_DEBUG', false),
```

The second value passed to the `env` function is the "default value". This value will be returned if no environment variable exists for the given key.

Determining the Current Environment

The current application environment is determined via the `APP_ENV` variable from your `.env` file. You may access this value via the `environment` method on the `App` facade:

```
use Illuminate\Support\Facades\App;  
$environment = App::environment();
```

You may also pass arguments to the `environment` method to determine if the environment matches a given value. The method will return `true` if the environment matches any of the given values:

```
if (App::environment('local')) {  
    // The environment is local  
}  
  
if (App::environment(['local', 'staging'])) {  
    // The environment is either local OR staging...  
}
```

[!NOTE] The current application environment detection can be overridden by defining a server-level `APP_ENV` environment variable.

Encrypting Environment Files

Unencrypted environment files should never be stored in source control. However, Laravel allows you to encrypt your environment files so that they may safely be added to source control with the rest of your application.

Encryption

To encrypt an environment file, you may use the `env:encrypt` command:

```
php artisan env:encrypt
```

Running the `env:encrypt` command will encrypt your `.env` file and place the encrypted contents in an `.env.encrypted` file. The decryption key is presented in the output of the command and should be stored in a secure password manager. If you would like to provide your own encryption key you may use the `--key` option when invoking the command:

```
php artisan env:encrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

[!NOTE] The length of the key provided should match the key length required by the encryption cipher being used. By default, Laravel will use the `AES-256-CBC` cipher which requires a 32 character key. You are free to use any cipher supported by Laravel's `encrypter` by passing the `--cipher` option when invoking the command.

If your application has multiple environment files, such as `.env` and `.env.staging`, you may specify the environment file that should be encrypted by providing the environment name via the `--env` option:

```
php artisan env:encrypt --env=staging
```

Decryption

To decrypt an environment file, you may use the `env:decrypt` command. This command requires a decryption key, which Laravel will retrieve from the `LARAVEL_ENV_ENCRYPTION_KEY` environment variable:

```
php artisan env:decrypt
```

Or, the key may be provided directly to the command via the `--key` option:

```
php artisan env:decrypt --key=3UVsEgGVK36XN82KKeyLFMhvosbZN1aF
```

When the `env:decrypt` command is invoked, Laravel will decrypt the contents of the `.env.encrypted` file and place the decrypted contents in the `.env` file.

The `--cipher` option may be provided to the `env:decrypt` command in order to use a custom encryption cipher:

```
php artisan env:decrypt --key=qUWuNRdfuImXcKxZ --cipher=AES-128-CBC
```

If your application has multiple environment files, such as `.env` and `.env.staging`, you may specify the environment file that should be decrypted by providing the environment name via the `--env` option:

```
php artisan env:decrypt --env=staging
```

In order to overwrite an existing environment file, you may provide the `--force` option to the `env:decrypt` command:

```
php artisan env:decrypt --force
```

Accessing Configuration Values

You may easily access your configuration values using the `Config` facade or global `config` function from anywhere in your application. The configuration values may be accessed using "dot" syntax, which includes the name of the file and option you wish to access. A default value may also be specified and will be returned if the configuration option does not exist:

```
use Illuminate\Support\Facades\Config;

$value = Config::get('app.timezone');

$value = config('app.timezone');

// Retrieve a default value if the configuration value does not exist...
$value = config('app.timezone', 'Asia/Seoul');
```

To set configuration values at runtime, you may invoke the `Config` facade's `set` method or pass an array to the `config` function:

```
Config::set('app.timezone', 'America/Chicago');

config(['app.timezone' => 'America/Chicago']);
```

To assist with static analysis, the `Config` facade also provides typed configuration retrieval methods. If the retrieved configuration value does not match the expected type, an exception will be thrown:

```
Config::string('config-key');
Config::integer('config-key');
Config::float('config-key');
Config::boolean('config-key');
Config::array('config-key');
```

Configuration Caching

To give your application a speed boost, you should cache all of your configuration files into a single file using the `config:cache` Artisan command. This will combine all of the configuration options for your application into a single file which can be quickly loaded by the framework.

You should typically run the `php artisan config:cache` command as part of your production deployment process. The command should not be run during local development as configuration options will frequently need to be changed during the course of your application's development.

Once the configuration has been cached, your application's `.env` file will not be loaded by the framework during requests or Artisan commands; therefore, the `env` function will only return external, system level environment variables.

For this reason, you should ensure you are only calling the `env` function from within your application's configuration (`config`) files. You can see many examples of this by examining Laravel's default configuration files. Configuration values may be accessed from anywhere in your application using the `config` function [described above](#).

The `config:clear` command may be used to purge the cached configuration:

```
php artisan config:clear
```

[!WARNING] If you execute the `config:cache` command during your deployment process, you should be sure that you are only calling the `env` function from within your configuration files. Once the configuration has been cached, the `.env` file will not be loaded; therefore, the `env` function will only return external, system level environment variables.

Configuration Publishing

Most of Laravel's configuration files are already published in your application's `config` directory; however, certain configuration files like `cors.php` and `view.php` are not published by default, as most applications will never need to modify them.

However, you may use the `config:publish` Artisan command to publish any configuration files that are not published by default:

```
php artisan config:publish  
  
php artisan config:publish --all
```

Debug Mode

The `debug` option in your `config/app.php` configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the `APP_DEBUG` environment variable, which is stored in your `.env` file.

[!WARNING] For local development, you should set the `APP_DEBUG` environment variable to `true`. **In your production environment, this value should always be `false`. If the variable is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.**

Maintenance Mode

When your application is in maintenance mode, a custom view will be displayed for all requests into your application. This makes it easy to "disable" your application while it is updating or when you are performing maintenance. A maintenance mode check is included in the default middleware stack for your application. If the application is in maintenance mode, a `Symfony\Component\HttpKernel\Exception\HttpException` instance will be thrown with a status code of 503.

To enable maintenance mode, execute the `down` Artisan command:

```
php artisan down
```

If you would like the `Refresh` HTTP header to be sent with all maintenance mode responses, you may provide the `refresh` option when invoking the `down` command. The `Refresh` header will instruct the browser to automatically refresh the page after the specified number of seconds:

```
php artisan down --refresh=15
```

You may also provide a `retry` option to the `down` command, which will be set as the `Retry-After` HTTP header's value, although browsers generally ignore this header:

```
php artisan down --retry=60
```


Bypassing Maintenance Mode

To allow maintenance mode to be bypassed using a secret token, you may use the `secret` option to specify a maintenance mode bypass token:

```
php artisan down --secret="1630542a-246b-4b66-afa1-dd72a4c43515"
```

After placing the application in maintenance mode, you may navigate to the application URL matching this token and Laravel will issue a maintenance mode bypass cookie to your browser:

```
https://example.com/1630542a-246b-4b66-afa1-dd72a4c43515
```

If you would like Laravel to generate the secret token for you, you may use the `with-secret` option. The secret will be displayed to you once the application is in maintenance mode:

```
php artisan down --with-secret
```

When accessing this hidden route, you will then be redirected to the `/` route of the application. Once the cookie has been issued to your browser, you will be able to browse the application normally as if it was not in maintenance mode.

[!NOTE] Your maintenance mode secret should typically consist of alpha-numeric characters and, optionally, dashes. You should avoid using characters that have special meaning in URLs such as `?` or `&`.

Maintenance Mode on Multiple Servers

By default, Laravel determines if your application is in maintenance mode using a file-based system. This means to activate maintenance mode, the `php artisan down` command has to be executed on each server hosting your application.

Alternatively, Laravel offers a cache-based method for handling maintenance mode. This method requires running the `php artisan down` command on just one server. To use this approach, modify the maintenance mode variables in your application's `.env` file. You should select a cache `store` that is accessible by all of your servers. This ensures the maintenance mode status is consistently maintained across every server:

```
APP_MAINTENANCE_DRIVER=cache  
APP_MAINTENANCE_STORE=database
```

Pre-Rendering the Maintenance Mode View

If you utilize the `php artisan down` command during deployment, your users may still occasionally encounter errors if they access the application while your Composer dependencies or other infrastructure components are updating. This occurs because a significant part of the Laravel framework must boot in order to determine your application is in maintenance mode and render the maintenance mode view using the templating engine.

For this reason, Laravel allows you to pre-render a maintenance mode view that will be returned at the very beginning of the request cycle. This view is rendered before any of your application's dependencies have loaded. You may pre-render a template of your choice using the `down` command's `render` option:

```
php artisan down --render="errors::503"
```

Redirecting Maintenance Mode Requests

While in maintenance mode, Laravel will display the maintenance mode view for all application URLs the user attempts to access. If you wish, you may instruct Laravel to redirect all requests to a specific URL. This may be accomplished using the `redirect` option. For example, you may wish to redirect all requests to the `/` URI:

```
php artisan down --redirect=/
```

Disabling Maintenance Mode

To disable maintenance mode, use the `up` command:

```
php artisan up
```

[!NOTE] You may customize the default maintenance mode template by defining your own template at `resources/views/errors/503.blade.php`.

Maintenance Mode and Queues

While your application is in maintenance mode, no [queued jobs](#) will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

Alternatives to Maintenance Mode

Since maintenance mode requires your application to have several seconds of downtime, consider running your applications on a fully-managed platform like [Laravel Cloud](#) to accomplish zero-downtime deployment with Laravel.

Directory Structure

- [Introduction](#)
- [The Root Directory](#)
 - [The app Directory](#)
 - [The bootstrap Directory](#)
 - [The config Directory](#)
 - [The database Directory](#)
 - [The public Directory](#)
 - [The resources Directory](#)
 - [The routes Directory](#)
 - [The storage Directory](#)
 - [The tests Directory](#)
 - [The vendor Directory](#)
- [The App Directory](#)
 - [The Broadcasting Directory](#)
 - [The Console Directory](#)
 - [The Events Directory](#)
 - [The Exceptions Directory](#)
 - [The Http Directory](#)
 - [The Jobs Directory](#)
 - [The Listeners Directory](#)
 - [The Mail Directory](#)
 - [The Models Directory](#)
 - [The Notifications Directory](#)
 - [The Policies Directory](#)
 - [The Providers Directory](#)
 - [The Rules Directory](#)

Introduction

The default Laravel application structure is intended to provide a great starting point for both large and small applications. But you are free to organize your application however you like. Laravel imposes almost no restrictions on where any given class is located - as long as Composer can autoload the class.

The Root Directory

The App Directory

The **app** directory contains the core code of your application. We'll explore this directory in more detail soon; however, almost all of the classes in your application will be in this directory.

The Bootstrap Directory

The **bootstrap** directory contains the **app.php** file which bootstraps the framework. This directory also houses a **cache** directory which contains framework generated files for performance optimization such as the route and services cache files.

The Config Directory

The **config** directory, as the name implies, contains all of your application's configuration files. It's a great idea to read through all of these files and familiarize yourself with all of the options available to you.

The Database Directory

The **database** directory contains your database migrations, model factories, and seeds. If you wish, you may also use this directory to hold an SQLite database.

The Public Directory

The **public** directory contains the **index.php** file, which is the entry point for all requests entering your application and configures autoloading. This directory also houses your assets such as images, JavaScript, and CSS.

The Resources Directory

The **resources** directory contains your **views** as well as your raw, un-compiled assets such as CSS or JavaScript.

The Routes Directory

The `routes` directory contains all of the route definitions for your application. By default, two route files are included with Laravel: `web.php` and `console.php`.

The `web.php` file contains routes that Laravel places in the `web` middleware group, which provides session state, CSRF protection, and cookie encryption. If your application does not offer a stateless, RESTful API then all your routes will most likely be defined in the `web.php` file.

The `console.php` file is where you may define all of your closure based console commands. Each closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application. You may also `schedule` tasks in the `console.php` file.

Optionally, you may install additional route files for API routes (`api.php`) and broadcasting channels (`channels.php`), via the `install:api` and `install:broadcasting` Artisan commands.

The `api.php` file contains routes that are intended to be stateless, so requests entering the application through these routes are intended to be authenticated [via tokens](#) and will not have access to session state.

The `channels.php` file is where you may register all of the [event broadcasting](#) channels that your application supports.

The Storage Directory

The `storage` directory contains your logs, compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into `app`, `framework`, and `logs` directories. The `app` directory may be used to store any files generated by your application. The `framework` directory is used to store framework generated files and caches. Finally, the `logs` directory contains your application's log files.

The `storage/app/public` directory may be used to store user-generated files, such as profile avatars, that should be publicly accessible. You should create a symbolic link at `public/storage` which points to this directory. You may create the link using the `php artisan storage:link` Artisan command.

The Tests Directory

The `tests` directory contains your automated tests. Example [Pest](#) or [PHPUnit](#) unit tests and feature tests are provided out of the box. Each test class should be suffixed with the word `Test`. You may run your tests using the `/vendor/bin/pest` or `/vendor/bin/phpunit` commands. Or, if you would like a more detailed and beautiful representation of your test results, you may run your tests using the `php artisan test` Artisan command.

The Vendor Directory

The `vendor` directory contains your [Composer](#) dependencies.

The App Directory

The majority of your application is housed in the `app` directory. By default, this directory is namespaced under `App` and is autoloaded by Composer using the [PSR-4 autoloading standard](#).

By default, the `app` directory contains the `Http`, `Models`, and `Providers` directories. However, over time, a variety of other directories will be generated inside the app directory as you use the make Artisan commands to generate classes. For example, the `app/Console` directory will not exist until you execute the `make:command` Artisan command to generate a command class.

Both the `Console` and `Http` directories are further explained in their respective sections below, but think of the `Console` and `Http` directories as providing an API into the core of your application. The HTTP protocol and CLI are both mechanisms to interact with your application, but do not actually contain application logic. In other words, they are two ways of issuing commands to your application. The `Console` directory contains all of your Artisan commands, while the `Http` directory contains your controllers, middleware, and requests.

[!NOTE] Many of the classes in the `app` directory can be generated by Artisan via commands. To review the available commands, run the `php artisan list make` command in your terminal.

The Broadcasting Directory

The `Broadcasting` directory contains all of the broadcast channel classes for your application. These classes are generated using the `make:channel` command. This directory does not exist by default, but will be created for you when you create your first channel. To learn more about channels, check out the documentation on [event broadcasting](#).

The Console Directory

The `Console` directory contains all of the custom Artisan commands for your application. These commands may be generated using the `make:command` command.

The Events Directory

This directory does not exist by default, but will be created for you by the `event:generate` and `make:event` Artisan commands. The `Events` directory houses [event classes](#). Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

The Exceptions Directory

The `Exceptions` directory contains all of the custom exceptions for your application. These exceptions may be generated using the `make:exception` command.

The Http Directory

The `Http` directory contains your controllers, middleware, and form requests. Almost all of the logic to handle requests entering your application will be placed in this directory.

The Jobs Directory

This directory does not exist by default, but will be created for you if you execute the `make:job` Artisan command. The `Jobs` directory houses the `queueable jobs` for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle. Jobs that run synchronously during the current request are sometimes referred to as "commands" since they are an implementation of the `command pattern`.

The Listeners Directory

This directory does not exist by default, but will be created for you if you execute the `event:generate` or `make:listener` Artisan commands. The `Listeners` directory contains the classes that handle your `events`. Event listeners receive an event instance and perform logic in response to the event being fired. For example, a `UserRegistered` event might be handled by a `SendWelcomeEmail` listener.

The Mail Directory

This directory does not exist by default, but will be created for you if you execute the `make:mail` Artisan command. The `Mail` directory contains all of your `classes that represent emails` sent by your application. Mail objects allow you to encapsulate all of the logic of building an email in a single, simple class that may be sent using the `Mail::send` method.

The Models Directory

The `Models` directory contains all of your `Eloquent model classes`. The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

The Notifications Directory

This directory does not exist by default, but will be created for you if you execute the `make:notification` Artisan command. The `Notifications` directory contains all of the "transactional" `notifications` that are sent by your application, such as simple notifications about events that happen within your application. Laravel's notification feature abstracts sending notifications over a variety of drivers such as email, Slack, SMS, or stored in a database.

The Policies Directory

This directory does not exist by default, but will be created for you if you execute the `make:policy` Artisan command. The `Policies` directory contains the `authorization policy classes` for your application. Policies are used to determine if a user can perform a given action against a resource.

The Providers Directory

The **Providers** directory contains all of the **service providers** for your application. Service providers bootstrap your application by binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.

In a fresh Laravel application, this directory will already contain the **AppServiceProvider**. You are free to add your own providers to this directory as needed.

The Rules Directory

This directory does not exist by default, but will be created for you if you execute the **make:rule** Artisan command. The **Rules** directory contains the custom validation rule objects for your application. Rules are used to encapsulate complicated validation logic in a simple object. For more information, check out the [validation documentation](#).

Frontend

- [Introduction](#)
- [Using PHP](#)
 - [PHP and Blade](#)
 - [Livewire](#)
 - [Starter Kits](#)
- [Using React or Vue](#)
 - [Inertia](#)
 - [Starter Kits](#)
- [Bundling Assets](#)

Introduction

Laravel is a backend framework that provides all of the features you need to build modern web applications, such as [routing](#), [validation](#), [caching](#), [queues](#), [file storage](#), and more. However, we believe it's important to offer developers a beautiful full-stack experience, including powerful approaches for building your application's frontend.

There are two primary ways to tackle frontend development when building an application with Laravel, and which approach you choose is determined by whether you would like to build your frontend by leveraging PHP or by using JavaScript frameworks such as Vue and React. We'll discuss both of these options below so that you can make an informed decision regarding the best approach to frontend development for your application.

Using PHP

PHP and Blade

In the past, most PHP applications rendered HTML to the browser using simple HTML templates interspersed with PHP `echo` statements which render data that was retrieved from a database during the request:

```
<div>
  <?php foreach ($users as $user): ?>
    Hello, <?php echo $user->name; ?> <br />
  <?php endforeach; ?>
</div>
```

In Laravel, this approach to rendering HTML can still be achieved using [views](#) and [Blade](#). Blade is an extremely light-weight templating language that provides convenient, short syntax for displaying data, iterating over data, and more:

```
<div>
  @foreach ($users as $user)
    Hello, {{ $user->name }} <br />
  @endforeach
</div>
```

When building applications in this fashion, form submissions and other page interactions typically receive an entirely new HTML document from the server and the entire page is re-rendered by the browser. Even today, many applications may be perfectly suited to having their frontends constructed in this way using simple Blade templates.

Growing Expectations

However, as user expectations regarding web applications have matured, many developers have found the need to build more dynamic frontends with interactions that feel more polished. In light of this, some developers choose to begin building their application's frontend using JavaScript frameworks such as Vue and React.

Others, preferring to stick with the backend language they are comfortable with, have developed solutions that allow the construction of modern web application UIs while still primarily utilizing their backend language of choice. For example, in the [Rails](#) ecosystem, this has spurred the creation of libraries such as [Turbo Hotwire](#), and [Stimulus](#).

Within the Laravel ecosystem, the need to create modern, dynamic frontends by primarily using PHP has led to the creation of [Laravel Livewire](#) and [Alpine.js](#).

Livewire

[Laravel Livewire](#) is a framework for building Laravel powered frontends that feel dynamic, modern, and alive just like frontends built with modern JavaScript frameworks like Vue and React.

When using Livewire, you will create Livewire "components" that render a discrete portion of your UI and expose methods and data that can be invoked and interacted with from your application's frontend. For example, a simple "Counter" component might look like the following:

```
namespace App\Http\Livewire;

use Livewire\Component;

class Counter extends Component
{
    public $count = 0;

    public function increment()
    {
        $this->count++;
    }

    public function render()
    {
        return view('livewire.counter');
    }
}
```

And, the corresponding template for the counter would be written like so:

```
<div>
    <button wire:click="increment">+</button>
    <h1>{{ $count }}</h1>
</div>
```

As you can see, Livewire enables you to write new HTML attributes such as `wire:click` that connect your Laravel application's frontend and backend. In addition, you can render your component's current state using simple Blade expressions.

For many, Livewire has revolutionized frontend development with Laravel, allowing them to stay within the comfort of Laravel while constructing modern, dynamic web applications. Typically, developers using Livewire will also utilize [Alpine.js](#) to "sprinkle" JavaScript onto their frontend only where it is needed, such as in order to render a dialog window.

If you're new to Laravel, we recommend getting familiar with the basic usage of [views](#) and [Blade](#). Then, consult the official [Laravel Livewire documentation](#) to learn how to take your application to the next level with interactive Livewire components.

Starter Kits

If you would like to build your frontend using PHP and Livewire, you can leverage our [Livewire starter kit](#) to jump-start your application's development.

Using React or Vue

Although it's possible to build modern frontends using Laravel and Livewire, many developers still prefer to leverage the power of a JavaScript framework like React or Vue. This allows developers to take advantage of the rich ecosystem of JavaScript packages and tools available via NPM.

However, without additional tooling, pairing Laravel with React or Vue would leave us needing to solve a variety of complicated problems such as client-side routing, data hydration, and authentication. Client-side routing is often simplified by using opinionated React / Vue frameworks such as [Next](#) and [Nuxt](#); however, data hydration and authentication remain complicated and cumbersome problems to solve when pairing a backend framework like Laravel with these frontend frameworks.

In addition, developers are left maintaining two separate code repositories, often needing to coordinate maintenance, releases, and deployments across both repositories. While these problems are not insurmountable, we don't believe it's a productive or enjoyable way to develop applications.

Inertia

Thankfully, Laravel offers the best of both worlds. [Inertia](#) bridges the gap between your Laravel application and your modern React or Vue frontend, allowing you to build full-fledged, modern frontends using React or Vue while leveraging Laravel routes and controllers for routing, data hydration, and authentication — all within a single code repository. With this approach, you can enjoy the full power of both Laravel and React / Vue without crippling the capabilities of either tool.

After installing Inertia into your Laravel application, you will write routes and controllers like normal. However, instead of returning a Blade template from your controller, you will return an Inertia page:

```

<?php

namespace App\Http\Controllers;

use App\Models\User;
use Inertia\Inertia;
use Inertia\Response;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     */
    public function show(string $id): Response
    {
        return Inertia::render('users/show', [
            'user' => User::findOrFail($id)
        ]);
    }
}

```

An Inertia page corresponds to a React or Vue component, typically stored within the `resources/js/pages` directory of your application. The data given to the page via the `Inertia::render` method will be used to hydrate the "props" of the page component:

```

import Layout from '@layouts/authenticated';
import { Head } from '@inertiajs/react';

export default function Show({ user }) {
    return (
        <Layout>
            <Head title="Welcome" />
            <h1>Welcome</h1>
            <p>Hello {user.name}, welcome to Inertia.</p>
        </Layout>
    )
}

```

As you can see, Inertia allows you to leverage the full power of React or Vue when building your frontend, while providing a light-weight bridge between your Laravel powered backend and your JavaScript powered frontend.

Server-Side Rendering

If you're concerned about diving into Inertia because your application requires server-side rendering, don't worry. Inertia offers [server-side rendering support](#). And, when deploying your application via [Laravel Cloud](#) or [Laravel Forge](#), it's a breeze to ensure that Inertia's server-side rendering process is always running.

Starter Kits

If you would like to build your frontend using Inertia and Vue / React, you can leverage our [React or Vue application starter kits](#) to jump-start your application's development. Both of these starter kits scaffold your application's backend and frontend authentication flow using Inertia, Vue / React, [Tailwind](#), and [Vite](#) so that you can start building your next big idea.

Bundling Assets

Regardless of whether you choose to develop your frontend using Blade and Livewire or Vue / React and Inertia, you will likely need to bundle your application's CSS into production ready assets. Of course, if you choose to build your application's frontend with Vue or React, you will also need to bundle your components into browser ready JavaScript assets.

By default, Laravel utilizes [Vite](#) to bundle your assets. Vite provides lightning-fast build times and near instantaneous Hot Module Replacement (HMR) during local development. In all new Laravel applications, including those using our [starter kits](#), you will find a `vite.config.js` file that loads our light-weight Laravel Vite plugin that makes Vite a joy to use with Laravel applications.

The fastest way to get started with Laravel and Vite is by beginning your application's development using [our application starter kits](#), which jump-starts your application by providing frontend and backend authentication scaffolding.

[!NOTE] For more detailed documentation on utilizing Vite with Laravel, please see our [dedicated documentation on bundling and compiling your assets](#).

Starter Kits

- [Introduction](#)
- [Creating an Application Using a Starter Kit](#)
- [Available Starter Kits](#)
 - [React](#)
 - [Vue](#)
 - [Livewire](#)
- [Starter Kit Customization](#)
 - [React](#)
 - [Vue](#)
 - [Livewire](#)
- [WorkOS AuthKit Authentication](#)
- [Inertia SSR](#)
- [Community Maintained Starter Kits](#)
- [Frequently Asked Questions](#)

Introduction

To give you a head start building your new Laravel application, we are happy to offer [application starter kits](#). These starter kits give you a head start on building your next Laravel application, and include the routes, controllers, and views you need to register and authenticate your application's users.

While you are welcome to use these starter kits, they are not required. You are free to build your own application from the ground up by simply installing a fresh copy of Laravel. Either way, we know you will build something great!

Creating an Application Using a Starter Kit

To create a new Laravel application using one of our starter kits, you should first [install PHP and the Laravel CLI tool](#). If you already have PHP and Composer installed, you may install the Laravel installer CLI tool via Composer:

```
composer global require laravel/installer
```

Then, create a new Laravel application using the Laravel installer CLI. The Laravel installer will prompt you to select your preferred starter kit:

```
laravel new my-app
```

After creating your Laravel application, you only need to install its frontend dependencies via NPM and start the Laravel development server:

```
cd my-app
npm install && npm run build
composer run dev
```

Once you have started the Laravel development server, your application will be accessible in your web browser at <http://localhost:8000>.

Available Starter Kits

React

Our React starter kit provides a robust, modern starting point for building Laravel applications with a React frontend using [Inertia](#).

Inertia allows you to build modern, single-page React applications using classic server-side routing and controllers. This lets you enjoy the frontend power of React combined with the incredible backend productivity of Laravel and lightning-fast Vite compilation.

The React starter kit utilizes React 19, TypeScript, Tailwind, and the [shadcn/ui](#) component library.

Vue

Our Vue starter kit provides a great starting point for building Laravel applications with a Vue frontend using [Inertia](#).

Inertia allows you to build modern, single-page Vue applications using classic server-side routing and controllers. This lets you enjoy the frontend power of Vue combined with the incredible backend productivity of Laravel and lightning-fast Vite compilation.

The Vue starter kit utilizes the Vue Composition API, TypeScript, Tailwind, and the [shadcn-vue](#) component library.

Livewire

Our Livewire starter kit provides the perfect starting point for building Laravel applications with a [Laravel Livewire](#) frontend.

Livewire is a powerful way of building dynamic, reactive, frontend UIs using just PHP. It's a great fit for teams that primarily use Blade templates and are looking for a simpler alternative to JavaScript-driven SPA frameworks like React and Vue.

The Livewire starter kit utilizes Livewire, Tailwind, and the [Flux UI](#) component library.

Starter Kit Customization

React

Our React starter kit is built with Inertia 2, React 19, Tailwind 4, and [shadcn/ui](#). As with all of our starter kits, all of the backend and frontend code exists within your application to allow for full customization.

The majority of the frontend code is located in the `resources/js` directory. You are free to modify any of the code to customize the appearance and behavior of your application:

```
resources/js/  
├── components/    # Reusable React components  
├── hooks/         # React hooks  
├── layouts/       # Application layouts  
├── lib/           # Utility functions and configuration  
├── pages/         # Page components  
└── types/        # TypeScript definitions
```

To publish additional shadcn components, first [find the component you want to publish](#). Then, publish the component using `npx`:

```
npx shadcn@latest add switch
```

In this example, the command will publish the Switch component to `resources/js/components/ui/switch.tsx`. Once the component has been published, you can use it in any of your pages:

```
import { Switch } from "@/components/ui/switch"  
  
const MyPage = () => {  
  return (  
    <div>  
      <Switch />  
    </div>  
  );  
};  
  
export default MyPage;
```

Available Layouts

The React starter kit includes two different primary layouts for you to choose from: a "sidebar" layout and a "header" layout. The sidebar layout is the default, but you can switch to the header layout by modifying the layout that is imported at the top of your application's `resources/js/layouts/app-layout.tsx` file:

```
import AppLayoutTemplate from '@layouts/app/app-sidebar-layout'; // [t1! remove]
import AppLayoutTemplate from '@layouts/app/app-header-layout'; // [t1! add]
```

Sidebar Variants

The sidebar layout includes three different variants: the default sidebar variant, the "inset" variant, and the "floating" variant. You may choose the variant you like best by modifying the `resources/js/components/app-sidebar.tsx` component:

```
<Sidebar collapsible="icon" variant="sidebar"> [t1! remove]
<Sidebar collapsible="icon" variant="inset"> [t1! add]
```

Authentication Page Layout Variants

The authentication pages included with the React starter kit, such as the login page and registration page, also offer three different layout variants: "simple", "card", and "split".

To change your authentication layout, modify the layout that is imported at the top of your application's `resources/js/layouts/auth-layout.tsx` file:

```
import AuthLayoutTemplate from '@layouts/auth/auth-simple-layout'; // [t1!
remove]
import AuthLayoutTemplate from '@layouts/auth/auth-split-layout'; // [t1! add]
```

Vue

Our Vue starter kit is built with Inertia 2, Vue 3 Composition API, Tailwind, and [shadcn-vue](#). As with all of our starter kits, all of the backend and frontend code exists within your application to allow for full customization.

The majority of the frontend code is located in the [resources/js](#) directory. You are free to modify any of the code to customize the appearance and behavior of your application:

```
resources/js/
├── components/    # Reusable Vue components
├── composables/  # Vue composables / hooks
├── layouts/       # Application layouts
├── lib/           # Utility functions and configuration
├── pages/         # Page components
└── types/        # TypeScript definitions
```

To publish additional shadcn-vue components, first [find the component you want to publish](#). Then, publish the component using [npx](#):

```
npx shadcn-vue@latest add switch
```

In this example, the command will publish the Switch component to [resources/js/components/ui/Switch.vue](#). Once the component has been published, you can use it in any of your pages:

```
<script setup lang="ts">
import { Switch } from '@Components/ui/switch'
</script>

<template>
  <div>
    <Switch />
  </div>
</template>
```

Available Layouts

The Vue starter kit includes two different primary layouts for you to choose from: a "sidebar" layout and a "header" layout. The sidebar layout is the default, but you can switch to the header layout by modifying the layout that is imported at the top of your application's [resources/js/layouts/AppLayout.vue](#) file:

```
import AppLayout from '@layouts/app/AppSidebarLayout.vue'; // [t1! remove]
import AppLayout from '@layouts/app/AppHeaderLayout.vue'; // [t1! add]
```

Sidebar Variants

The sidebar layout includes three different variants: the default sidebar variant, the "inset" variant, and the "floating" variant. You may choose the variant you like best by modifying the `resources/js/components/AppSidebar.vue` component:

```
<Sidebar collapsible="icon" variant="sidebar"> [t1! remove]
<Sidebar collapsible="icon" variant="inset"> [t1! add]
```

Authentication Page Layout Variants

The authentication pages included with the Vue starter kit, such as the login page and registration page, also offer three different layout variants: "simple", "card", and "split".

To change your authentication layout, modify the layout that is imported at the top of your application's `resources/js/layouts/AuthLayout.vue` file:

```
import AuthLayout from '@layouts/auth/AuthSimpleLayout.vue'; // [t1! remove]
import AuthLayout from '@layouts/auth/AuthSplitLayout.vue'; // [t1! add]
```

Livewire

Our Livewire starter kit is built with Livewire 3, Tailwind, and [Flux UI](#). As with all of our starter kits, all of the backend and frontend code exists within your application to allow for full customization.

Livewire and Volt

The majority of the frontend code is located in the `resources/views` directory. You are free to modify any of the code to customize the appearance and behavior of your application:

```
resources/views
├── components      # Reusable Livewire components
├── flux            # Customized Flux components
├── livewire        # Livewire pages
├── partials        # Reusable Blade partials
├── dashboard.blade.php # Authenticated user dashboard
└── welcome.blade.php  # Guest user welcome page
```

Traditional Livewire Components

The frontend code is located in the `resources/views` directory, while the `app/Livewire` directory contains the corresponding backend logic for the Livewire components.

Available Layouts

The Livewire starter kit includes two different primary layouts for you to choose from: a "sidebar" layout and a "header" layout. The sidebar layout is the default, but you can switch to the header layout by modifying the layout that is used by your application's `resources/views/components/layouts/app.blade.php` file. In addition, you should add the `container` attribute to the main Flux component:

```
<x-layouts.app.header>
  <flux:main container>
    {{ $slot }}
  </flux:main>
</x-layouts.app.header>
```

Authentication Page Layout Variants

The authentication pages included with the Livewire starter kit, such as the login page and registration page, also offer three different layout variants: "simple", "card", and "split".

To change your authentication layout, modify the layout that is used by your application's `resources/views/components/layouts/auth.blade.php` file:

```
<x-layouts.auth.split>
  {{ $slot }}
</x-layouts.auth.split>
```

WorkOS AuthKit Authentication

By default, the React, Vue, and Livewire starter kits all utilize Laravel's built-in authentication system to offer login, registration, password reset, email verification, and more. In addition, we also offer a [WorkOS AuthKit](#) powered variant of each starter kit that offers:

- Social authentication (Google, Microsoft, GitHub, and Apple)
- Passkey authentication
- Email based "Magic Auth"
- SSO

Using WorkOS as your authentication provider [requires a WorkOS account](#). WorkOS offers free authentication for applications up to 1 million monthly active users.

To use WorkOS AuthKit as your application's authentication provider, select the WorkOS option when creating your new starter kit powered application via `laravel new`.

Configuring Your WorkOS Starter Kit

After creating a new application using a WorkOS powered starter kit, you should set the `WORKOS_CLIENT_ID`, `WORKOS_API_KEY`, and `WORKOS_REDIRECT_URL` environment variables in your application's `.env` file. These variables should match the values provided to you in the WorkOS dashboard for your application:

```
WORKOS_CLIENT_ID=your-client-id
WORKOS_API_KEY=your-api-key
WORKOS_REDIRECT_URL="${APP_URL}/authenticate"
```

Additionally, you should configure the application homepage URL in your WorkOS dashboard. This URL is where users will be redirected after they log out of your application.

Configuring AuthKit Authentication Methods

When using a WorkOS powered starter kit, we recommend that you disable "Email + Password" authentication within your application's WorkOS AuthKit configuration settings, allowing users to only authenticate via social authentication providers, passkeys, "Magic Auth", and SSO. This allows your application to totally avoid handling user passwords.

Configuring AuthKit Session Timeouts

In addition, we recommend that you configure your WorkOS AuthKit session inactivity timeout to match your Laravel application's configured session timeout threshold, which is typically two hours.

Inertia SSR

The React and Vue starter kits are compatible with Inertia's [server-side rendering](#) capabilities. To build an Inertia SSR compatible bundle for your application, run the `build:ssr` command:

```
npm run build:ssr
```

For convenience, a `composer dev:ssr` command is also available. This command will start the Laravel development server and Inertia SSR server after building an SSR compatible bundle for your application, allowing you to test your application locally using Inertia's server-side rendering engine:

```
composer dev:ssr
```

Community Maintained Starter Kits

When creating a new Laravel application using the Laravel installer, you may provide any community maintained starter kit available on Packagist to the `--using` flag:

```
laravel new my-app --using=example/starter-kit
```

Creating Starter Kits

To ensure your starter kit is available to others, you will need to publish it to [Packagist](#). Your starter kit should define its required environment variables in its `.env.example` file, and any necessary post-installation commands should be listed in the `post-create-project-cmd` array of the starter kit's `composer.json` file.

Frequently Asked Questions

How do I upgrade?

Every starter kit gives you a solid starting point for your next application. With full ownership of the code, you can tweak, customize, and build your application exactly as you envision. However, there is no need to update the starter kit itself.

How do I enable email verification?

Email verification can be added by uncommenting the `MustVerifyEmail` import in your `App/Models/User.php` model and ensuring the model implements the `MustVerifyEmail` interface:

```
namespace App\Models;

use Illuminate\Contracts\Auth\MustVerifyEmail;
// ...

class User extends Authenticatable implements MustVerifyEmail
{
    // ...
}
```

After registration, users will receive a verification email. To restrict access to certain routes until the user's email address is verified, add the `verified` middleware to the routes:

```
Route::middleware(['auth', 'verified'])->group(function () {
    Route::get('dashboard', function () {
        return Inertia::render('dashboard');
    }->name('dashboard'));
});
```

[!NOTE] Email verification is not required when using the [WorkOS](#) variant of the starter kits.

How do I modify the default email template?

You may want to customize the default email template to better align with your application's branding. To modify this template, you should publish the email views to your application with the following command:

```
php artisan vendor:publish --tag=laravel-mail
```

This will generate several files in `resources/views/vendor/mail`. You can modify any of these files as well as the `resources/views/vendor/mail/themes/default.css` file to change the look and appearance of the default email template.

Deployment

- [Introduction](#)
- [Server Requirements](#)
- [Server Configuration](#)
 - [Nginx](#)
 - [FrankenPHP](#)
 - [Directory Permissions](#)
- [Optimization](#)
 - [Caching Configuration](#)
 - [Caching Events](#)
 - [Caching Routes](#)
 - [Caching Views](#)
- [Debug Mode](#)
- [The Health Route](#)
- [Deploying With Laravel Cloud or Forge](#)

Introduction

When you're ready to deploy your Laravel application to production, there are some important things you can do to make sure your application is running as efficiently as possible. In this document, we'll cover some great starting points for making sure your Laravel application is deployed properly.

Server Requirements

The Laravel framework has a few system requirements. You should ensure that your web server has the following minimum PHP version and extensions:

- PHP \geq 8.2
- ctype PHP Extension
- cURL PHP Extension
- DOM PHP Extension
- Fileinfo PHP Extension
- Filter PHP Extension
- Hash PHP Extension
- Mbstring PHP Extension
- OpenSSL PHP Extension
- PCRE PHP Extension
- PDO PHP Extension
- Session PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Server Configuration

Nginx

If you are deploying your application to a server that is running Nginx, you may use the following configuration file as a starting point for configuring your web server. Most likely, this file will need to be customized depending on your server's configuration. **If you would like assistance in managing your server, consider using a fully-managed Laravel platform like [Laravel Cloud](#).**

Please ensure, like the configuration below, your web server directs all requests to your application's `public/index.php` file. You should never attempt to move the `index.php` file to your project's root, as serving the application from the project root will expose many sensitive configuration files to the public Internet:

```
server {
    listen 80;
    listen [::]:80;
    server_name example.com;
    root /srv/example.com/public;

    add_header X-Frame-Options "SAMEORIGIN";
    add_header X-Content-Type-Options "nosniff";

    index index.php;

    charset utf-8;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    error_page 404 /index.php;

    location ~ ^/index\.php(/|$) {
        fastcgi_pass unix:/var/run/php/php8.2-fpm.sock;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
        fastcgi_hide_header X-Powered-By;
    }

    location ~ /\.(!well-known).* {
        deny all;
    }
}
```

FrankenPHP

[FrankenPHP](#) may also be used to serve your Laravel applications. FrankenPHP is a modern PHP application server written in Go. To serve a Laravel PHP application using FrankenPHP, you may simply invoke its `php-server` command:

```
frankenphp php-server -r public/
```

To take advantage of more powerful features supported by FrankenPHP, such as its [Laravel Octane](#) integration, HTTP/3, modern compression, or the ability to package Laravel applications as standalone binaries, please consult FrankenPHP's [Laravel documentation](#).

Directory Permissions

Laravel will need to write to the `bootstrap/cache` and `storage` directories, so you should ensure the web server process owner has permission to write to these directories.

Optimization

When deploying your application to production, there are a variety of files that should be cached, including your configuration, events, routes, and views. Laravel provides a single, convenient `optimize` Artisan command that will cache all of these files. This command should typically be invoked as part of your application's deployment process:

```
php artisan optimize
```

The `optimize:clear` method may be used to remove all of the cache files generated by the `optimize` command as well as all keys in the default cache driver:

```
php artisan optimize:clear
```

In the following documentation, we will discuss each of the granular optimization commands that are executed by the `optimize` command.

Caching Configuration

When deploying your application to production, you should make sure that you run the `config:cache` Artisan command during your deployment process:

```
php artisan config:cache
```

This command will combine all of Laravel's configuration files into a single, cached file, which greatly reduces the number of trips the framework must make to the filesystem when loading your configuration values.

[!WARNING] If you execute the `config:cache` command during your deployment process, you should be sure that you are only calling the `env` function from within your configuration files. Once the configuration has been cached, the `.env` file will not be loaded and all calls to the `env` function for `.env` variables will return `null`.

Caching Events

You should cache your application's auto-discovered event to listener mappings during your deployment process. This can be accomplished by invoking the `event:cache` Artisan command during deployment:

```
php artisan event:cache
```

Caching Routes

If you are building a large application with many routes, you should make sure that you are running the `route:cache` Artisan command during your deployment process:

```
php artisan route:cache
```

This command reduces all of your route registrations into a single method call within a cached file, improving the performance of route registration when registering hundreds of routes.

Caching Views

When deploying your application to production, you should make sure that you run the `view:cache` Artisan command during your deployment process:

```
php artisan view:cache
```

This command precompiles all your Blade views so they are not compiled on demand, improving the performance of each request that returns a view.

Debug Mode

The debug option in your `config/app.php` configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the `APP_DEBUG` environment variable, which is stored in your application's `.env` file.

[!WARNING] In your production environment, this value should always be `false`. If the `APP_DEBUG` variable is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.

The Health Route

Laravel includes a built-in health check route that can be used to monitor the status of your application. In production, this route may be used to report the status of your application to an uptime monitor, load balancer, or orchestration system such as Kubernetes.

By default, the health check route is served at `/up` and will return a 200 HTTP response if the application has booted without exceptions. Otherwise, a 500 HTTP response will be returned. You may configure the URI for this route in your application's `bootstrap/app` file:

```
->withRouting(  
    web: __DIR__.'/../routes/web.php',  
    commands: __DIR__.'/../routes/console.php',  
    health: '/up', // [tl! remove]  
    health: '/status', // [tl! add]  
)
```

When HTTP requests are made to this route, Laravel will also dispatch a `Illuminate\Foundation\Events\DiagnosingHealth` event, allowing you to perform additional health checks relevant to your application. Within a `listener` for this event, you may check your application's database or cache status. If you detect a problem with your application, you may simply throw an exception from the listener.

Deploying With Laravel Cloud or Forge

Laravel Cloud

If you would like a fully-managed, auto-scaling deployment platform tuned for Laravel, check out [Laravel Cloud](#). Laravel Cloud is a robust deployment platform for Laravel, offering managed compute, databases, caches, and object storage.

Launch your Laravel application on Cloud and fall in love with the scalable simplicity. Laravel Cloud is fine-tuned by Laravel's creators to work seamlessly with the framework so you can keep writing your Laravel applications exactly like you're used to.

Laravel Forge

If you prefer to manage your own servers but aren't comfortable configuring all of the various services needed to run a robust Laravel application, [Laravel Forge](#) is a VPS server management platform for Laravel applications.

Laravel Forge can create servers on various infrastructure providers such as DigitalOcean, Linode, AWS, and more. In addition, Forge installs and manages all of the tools needed to build robust Laravel applications, such as Nginx, MySQL, Redis, Memcached, Beanstalk, and more.