|  |  |  |
| --- | --- | --- |
| Laravel 10 | | |
|  | | |
| Author: | Harpreet Singh |  |
| Date: | Aug 27, 2023 |  |
| Time: | **4:06 PM** |  |
| Source: | Laravel.com |  |

# Laravel commands ( - )

|  |  |
| --- | --- |
| Title | **Description/Commands** |
| What is Laravel? | After you Laravel is an open-source web application framework. It is a very well documented, expressive, and easy to learn framework. Laravel is very developer friendly as the framework can help beginners as well as advanced users. As you grow as a developer you can go deeper into Laravel functionalities and give more robust and enterprise solutions.  Additionally, the framework is very scalable as you can use packages like Vapor to handle hundreds of thousands of requests using AWS serverless technology. |
| Install Laravel | After you have installed PHP and Composer, you may create a new Laravel project via the Composer create-project command.   * composer create-project Laravel/Laravel example-project |
| What is artisan? | Artisan is the command line interface included with Laravel. Artisan exists at the root of your application as the artisan script and provides a number of helpful commands that can assist you while you build your application. |
| Artisan CLI | After the project has been created, start Laravel's local development server using the Laravel's Artisan CLI serve command   * cd example-project * php artisan serve |
| Breeze package | Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. In addition, Breeze includes a simple "profile" page where the user may update their name, email address, and password.  Laravel Breeze's default view layer is made up of simple Blade templates styled with Tailwind CSS. Or, Breeze can scaffold your application using Vue or React and Inertia.  <https://laravel.com/docs/10.x/starter-kits#laravel-breeze> |
| Breeze install | First, you should create a new Laravel application, configure your database, and run your database migrations. Once you have created a new Laravel application, you may install Laravel Breeze using Composer:   * composer require Laravel/breeze –dev * php artisan breeze:install * php artisan migrate * npm install * npm run dev |
| Migration | Migrations are like version control for our database, allowing developers to define and share the application’s database schema definition. If we have ever had to tell a teammate to manually add a column to their local database schema after pulling in your changes from source control, you have faced the problem that database migration solve.  The Laravel Schema façade provides database agnostic support for creating and manipulating tables across all of Laravel’s supported database systems. Typically, migrations will use this façade to create and modify database tables and columns.   * php artisan migrate -help |
| Run migration | Run the migrations and create database schema   * php artisan migrate |
| Create migration | Create a migration to rename a column from the table   * php artisan make:migration update\_user\_table\_name\_to\_username –table=users * php artisan migrate |
|  | Create a migration to add new column to users table   * php artisan make:migration update\_user\_table\_add\_avatar\_field --table=users |
|  | make:controller Create a new controller class  php artisan make:controller Profile/AvatarController |
| Database: Getting Started | Almost every modern web application interacts with a database. Laravel makes interacting with databases extremely simple across a variety of supported databases using raw SQL, a fluent query builder, and the Eloquent ORM. Currently, Laravel provides first-party support for five databases:  MariaDB 10.10+ (Version Policy)  MySQL 5.7+ (Version Policy)  PostgreSQL 11.0+ (Version Policy)  SQLite 3.8.8+  SQL Server 2017+ (Version Policy) |
| DB Facade | Once we have configured our database connection, you may run queries using the DB facade. The DB facade provides methods for each type of query: select, update, insert, delete, and statement.  <https://laravel.com/docs/10.x/database#introduction>  It is facility in Laravel which provides different method to interact with database like select, insert, update, delete.  To run a basic SELECT query, you may use the select method on the DB facade:  $users = DB::select('select \* from users');  $users = DB::select('select \* from users where active = ?', [True]);  $pendingpayments = DB::select('select \* from payment where status = ?', [unpaid]);  **Selecting Scalar Values**  Sometimes your database query may result in a single, scalar value. Instead of being required to retrieve the query's scalar result from a record object, Laravel allows you to retrieve this value directly using the scalar method:  $burgers = DB::scalar(  "select count(case when food = 'burger' then 1 end) as burgers from menu"  );  **Using Named Bindings**  Instead of using ? to represent your parameter bindings, you may execute a query using named bindings:  $results = DB::select('select \* from users where id = :id', ['id' => 1]);  DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);  $affected = DB::update(  'update users set votes = 100 where name = ?',  ['Anita']  );  $deleted = DB::delete('delete from users');  DB::statement('drop table users'); |
| Query Builder | Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works perfectly with all of Laravel's supported database systems.  <https://laravel.com/docs/10.x/queries#introduction>  The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean or sanitize strings passed to the query builder as query bindings.  use Illuminate\Support\Facades\DB;    $users = DB::table('users')->get();    foreach ($users as $user) {  echo $user->name;  }  $user = DB::table('users')->where('name', 'John')->first();  return $user->email;  $email = DB::table('users')->where('name', 'John')->value('email');  $user = DB::table('users')->find(3);  $titles = DB::table('users')->pluck('title'); |
| Eloquent ORM | Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well. |
| Helper functions  :Aug 30, 2023 | Laravel includes a variety of global "helper" PHP functions. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.  <https://laravel.com/docs/10.x/helpers#introduction> |
| Str helper functions | Str::random()  The Str::random method generates a random string of the specified length. This function uses PHP's random\_bytes function:  use Illuminate\Support\Str;  $random = Str::random(40);  Str::upper()  The Str::upper method converts the given string to uppercase:  use Illuminate\Support\Str;  $string = Str::upper('laravel');  // LARAVEL |
| bcrypt() | The bcrypt function hashes the given value using Bcrypt. You may use this function as an alternative to the Hash facade:  $password = bcrypt('my-secret-password'); |
| Accessor | An accessor transforms an Eloquent attribute value when it is accessed. To define an accessor, create a protected method on your model to represent the accessible attribute. This method name should correspond to the "camel case" representation of the true underlying model attribute / database column when applicable.  To do so, we supply the **get** argument to the Attribute class constructor.  In this example, we'll define an accessor for the first\_name attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of the first\_name attribute. All attribute accessor / mutator methods must declare a return type-hint of Illuminate\Database\Eloquent\Casts\Attribute:  <https://laravel.com/docs/10.x/eloquent-mutators#defining-an-accessor>  <?php    namespace App\Models;    use Illuminate\Database\Eloquent\Casts\Attribute;  use Illuminate\Database\Eloquent\Model;    class User extends Model  {  /\*\*  \* Get the user's first name.  \*/  protected function firstName(): Attribute  {  return Attribute::make(  get: fn (string $value) => ucfirst($value),  );  }  } |
| Mutator | A mutator transforms an Eloquent attribute value when it is set. To define a mutator, you may provide the **set** argument when defining your attribute. Let's define a mutator for the user login password attribute. This mutator will be automatically called when we attempt to set the value of the user login password attribute on the model:  <https://laravel.com/docs/10.x/eloquent-mutators#defining-a-mutator>  <?php    namespace App\Models;    use Illuminate\Database\Eloquent\Casts\Attribute;  use Illuminate\Database\Eloquent\Model;    class User extends Model  {  /\*\*  \* Interact with the user's first name.  \*/  protected function password(): Attribute  {  return Attribute::make(  set: fn (string $value) => bcrypt($value),  );  }  } |
| Config cache | To create cache for config folder to a single file which is location in /bootstrap/cache/config.php  php artisan config:cache  php artisan config:show  php artisan config:clear  config cache will help to load faster configuration files & reduce load time. |
| Tinker  :31 Aug, 2023 | Interact with your application  <https://laravel.com/docs/10.x/artisan#tinker>  Laravel Tinker is a powerful REPL for the Laravel framework, powered by the PsySH package.  All Laravel applications include Tinker by default. However, you may install Tinker using Composer if you have previously removed it from your application:   * composer require laravel/tinker   Tinker allows you to interact with your entire Laravel application on the command line, including your Eloquent models, jobs, events, and more. To enter the Tinker environment, run the tinker Artisan command:   * php artisan tinker |
| Mass Assignment | <https://laravel.com/docs/10.x/eloquent#mass-assignment> |
| Form Method Spoofing | HTML forms do not support PUT, PATCH, or DELETE actions. So, when defining PUT, PATCH, or DELETE routes that are called from an HTML form, you will need to add a hidden \_method field to the form. The value sent with the \_method field will be used as the HTTP request method:  <form action="/example" method="POST">  <input type="hidden" name="\_method" value="PUT">  <input type="hidden" name="\_token" value="{{ csrf\_token() }}">  </form>  For convenience, you may use the @method Blade directive to generate the \_method input field:  <form action="/example" method="POST">  @method('PUT')  @csrf  </form> |
| CSRF Protection | <https://laravel.com/docs/10.x/csrf#main-content>  Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user. Thankfully, Laravel makes it easy to protect your application from cross-site request forgery (CSRF) attacks.  Laravel automatically generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the person actually making the requests to the application. Since this token is stored in the user's session and changes each time the session is regenerated, a malicious application is unable to access it.  The current session's CSRF token can be accessed via the request's session or via the csrf\_token helper function:  use Illuminate\Http\Request;    Route::get('/token', function (Request $request) {  $token = $request->session()->token();    $token = csrf\_token();    // ...  });  Anytime you define a "POST", "PUT", "PATCH", or "DELETE" HTML form in your application, you should include a hidden CSRF \_token field in the form so that the CSRF protection middleware can validate the request. For convenience, you may use the @csrf Blade directive to generate the hidden token input field:  <form method="POST" action="/profile">  @csrf    <!-- Equivalent to... -->  <input type="hidden" name="\_token" value="{{ csrf\_token() }}" />  </form>  The App\Http\Middleware\VerifyCsrfToken middleware, which is included in the web middleware group by default, will automatically verify that the token in the request input matches the token stored in the session. When these two tokens match, we know that the authenticated user is the one initiating the request. |
| HTTP Responses  Redirects | All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is returning a string from a route or controller. The framework will automatically convert the string into a full HTTP response:  <https://laravel.com/docs/10.x/responses#redirects>  Redirect responses are instances of the Illuminate\Http\RedirectResponse class, and contain the proper headers needed to redirect the user to another URL. There are several ways to generate a RedirectResponse instance. The simplest method is to use the global redirect helper:  Route::get('/dashboard', function () {  return redirect('home/dashboard');  });  return redirect()->route('login');  return redirect()->route('profile', ['id' => 1]); |
| Form Validation | <https://laravel.com/docs/10.x/validation#rule-image>  Laravel provides several different approaches to validate your application's incoming data. It is most common to use the validate method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.  Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.  /\*\*  \* Store a new blog post.  \*/  public function store(Request $request): RedirectResponse  {  $validated = $request->validate([  'title' => 'required|unique:posts|max:255',  'body' => 'required',  ]);    // The blog post is valid...    return redirect('/posts');  } |
| Form Request Validation | Creating Form Requests  For more complex validation scenarios, you may wish to create a "form request". Form requests are custom request classes that encapsulate their own validation and authorization logic. To create a form request class, you may use the make:request Artisan CLI command:  php artisan make:request StorePostRequest  The generated form request class will be placed in the app/Http/Requests directory. If this directory does not exist, it will be created when you run the make:request command. Each form request generated by Laravel has two methods: authorize and rules.  As you might have guessed, the authorize method is responsible for determining if the currently authenticated user can perform the action represented by the request, while the rules method returns the validation rules that should apply to the request's data:  /\*\*  \* Get the validation rules that apply to the request.  \*  \* @return array<string, \Illuminate\Contracts\Validation\Rule|array|string>  \*/  public function rules(): array  {  return [  'title' => 'required|unique:posts|max:255',  'body' => 'required',  ];  } |
| File Storage | File Uploads  In web applications, one of the most common use-cases for storing files is storing user uploaded files such as photos and documents. Laravel makes it very easy to store uploaded files using the store method on an uploaded file instance. Call the store method with the path at which you wish to store the uploaded file:  $path = $request->file('avatar')->store('avatars'); |
|  | **The Public Disk**  [**https://laravel.com/docs/10.x/filesystem#the-public-disk**](https://laravel.com/docs/10.x/filesystem#the-public-disk)  The public disk included in your application's filesystems configuration file is intended for files that are going to be publicly accessible. By default, the public disk uses the local driver and stores its files in storage/app/public.  To make these files accessible from the web, you should create a symbolic link from public/storage to storage/app/public. Utilizing this folder convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like Envoyer.  To create the symbolic link, you may use the storage:link Artisan command:   * php artisan storage:link   Once a file has been stored and the symbolic link has been created, you can create a URL to the files using the asset helper:   * echo asset('storage/file.txt'); |
|  | **Deleting Files**  [**https://laravel.com/docs/10.x/filesystem#deleting-files**](https://laravel.com/docs/10.x/filesystem#deleting-files)  The delete method accepts a single filename or an array of files to delete:  use Illuminate\Support\Facades\Storage;    Storage::delete('file.jpg');    Storage::delete(['file.jpg', 'file2.jpg']);  If necessary, you may specify the disk that the file should be deleted from:  use Illuminate\Support\Facades\Storage;    Storage::disk('public')->delete('path/file.jpg'); |
| Laravel Socialite | In addition to typical, form based authentication, Laravel also provides a simple, convenient way to authenticate with OAuth providers using Laravel Socialite. Socialite currently supports authentication via Facebook, Twitter, LinkedIn, Google, GitHub, GitLab, Bitbucket, and Slack.  <https://laravel.com/docs/10.x/socialite>  To get started with Socialite, use the Composer package manager to add the package to your project's dependencies:   * composer require laravel/socialite |
| Model | Create a new Eloquent model class   * php artisan make:model -h   Along with controller, migration and request run a combined make model command   * php artisan make:model Ticket -m -r -R   It will create Ticket Model, migration, store & update ticket request, Ticket Controller |
| App\Enums | namespace App\Enums;  enum TicketStatus:string  {  case OPEN = 'open';  case RESOLVED = 'resolved';  case REJECTED = 'rejected';  }  In migration:  $table->string('status')->default(TicketStatus::OPEN->value); |
| Route List | * php artisan route:list |
| Resource Controllers | https://laravel.com/docs/10.x/controllers#resource-controllers   * php artisan make:controller PhotoController –resource   use App\Http\Controllers\PhotoController;    Route::resource('photos', PhotoController::class); |
| Blade Components | <https://laravel.com/docs/10.x/blade#components>  There are two approaches to writing components: class based components and anonymous components.  The make:component command will also create a view template for the component. The view will be placed in the resources/views/components directory. When writing components for your own application, components are automatically discovered within the app/View/Components directory and resources/views/components directory, so no further component registration is typically required.   * php artisan make:component Textarea |
| Validation In array | <https://laravel.com/docs/10.x/validation#rule-in-array>  use Illuminate\Validation\Rule;    Validator::make($data, [  'toppings' => [  'required',  Rule::In(['garlic', pineapple']),  Rule::notIn(['sprinkles', 'cherries']),  ],  ]); |
| Notifications | <https://laravel.com/docs/10.x/notifications#main-content>  Generating Notifications  In Laravel, each notification is represented by a single class that is typically stored in the app/Notifications directory. Don't worry if you don't see this directory in your application - it will be created for you when you run the make:notification Artisan command:   * php artisan make:notification InvoicePaid   This command will place a fresh notification class in your app/Notifications directory. Each notification class contains a via method and a variable number of message building methods, such as toMail or toDatabase, that convert the notification to a message tailored for that particular channel. |
| Eloquent: Relationships | <https://laravel.com/docs/10.x/eloquent-relationships#one-to-many> |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
| Garbage Collection |  |

: Added date, :: Updated date