Artisan Console

Introduction

Artisan is the command-line interface included with Laravel. It provides a number of helpful commands that can assist you while you build your application. To view a list of all available Artisan commands, you may use the `list` command:

```
php artisan list
```

Every command also includes a "help" screen which displays and describes the command's available arguments and options. To view a help screen, precede the name of the command with `help`:

```
php artisan help migrate
```

Tinker (REPL)

All Laravel applications include Tinker, a REPL powered by the PsySH package. Tinker allows you to interact with your entire Laravel application on the command line, including the Eloquent ORM, jobs, events, and more. To enter the Tinker environment, run the `tinker` Artisan command:

```
php artisan tinker
```

You can publish Tinker's configuration file using the `vendor:publish` command:

```
php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"
```

Command Whitelist

Tinker utilizes a white-list to determine which Artisan commands are allowed to be run within its shell. By default, you may run the `clear-compiled`, `down`, `env`, `inspire`, `migrate`, `optimize`, and `up` commands. If you would like to white-list more commands you may add them to the `commands` array in your `tinker.php` configuration file:

```
'commands' => [
    // App\Console\Commands\ExampleCommand::class,
],
```

Alias Blacklist

Typically, Tinker automatically aliases classes as you require them in Tinker. However, you may wish to never alias some classes. You may accomplish this by listing the classes in the `dont_alias` array of your `tinker.php` configuration file:

```
'dont_alias' => [
    App\User::class,
],
```

Writing Commands

In addition to the commands provided with Artisan, you may also build your own custom commands. Commands are typically stored in the `app/Console/Commands` directory; however, you are free to choose your own storage location as long as your commands can be loaded by Composer.

Generating Commands

To create a new command, use the `make:command` Artisan command. This command will create a new command class in the `app/Console/Commands` directory. Don't worry if this directory does not exist in your application, since it will be created the first time you run the `make:command` Artisan command. The generated command will include the default set of properties and methods that are present on all commands:

```
php artisan make:command SendEmails
```

Command Structure

After generating your command, you should fill in the `signature` and `description` properties of the class, which will be used when displaying your command on the `list` screen. The `handle` method will be called when your command is executed. You may place your command logic in this method.

For greater code reuse, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks. In the example below, note that we inject a service class to do the "heavy lifting" of sending the e-mails.

Let's take a look at an example command. Note that we are able to inject any dependencies we need into the command's `handle` method. The Laravel service container will automatically inject all dependencies that are type-hinted in this method's signature:

```
<?php

namespace App\Console\Commands;

use App\DripEmailer;
use App\User;
use Illuminate\Console\Command;

class SendEmails extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'email:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send drip e-mails to a user';

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Execute the console command.
     *
     * @param  \App\DripEmailer  $drip
     * @return mixed
     */
    public function handle(DripEmailer $drip)
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```

Closure Commands

Closure based commands provide an alternative to defining console commands as classes. In the same way that route Closures are an alternative to controllers, think of command Closures as an alternative to command classes. Within the `commands` method of your `app/Console/Kernel.php` file, Laravel loads the `routes/console.php` file:

```
/**
 * Register the Closure based commands for the application.
 *
 * @return void
 */
protected function commands()
{
    require base_path('routes/console.php');
}
```

Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application. Within this file, you may define all of your Closure based routes using the `Artisan::command` method. The `command` method accepts two arguments: the command signature and a Closure which receives the commands arguments and options:

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
});
```

The Closure is bound to the underlying command instance, so you have full access to all of the helper methods you would typically be able to access on a full command class.

Type-Hinting Dependencies

In addition to receiving your command's arguments and options, command Closures may also type-hint additional dependencies that you would like resolved out of the service container:

```
use App\DripEmailer;
use App\User;

Artisan::command('email:send {user}', function (DripEmailer $drip, $user) {
    $drip->send(User::find($user));
});
```

Closure Command Descriptions

When defining a Closure based command, you may use the `describe` method to add a description to the command. This description will be displayed when you run the `php artisan list` or `php artisan help` commands:

```
Artisan::command('build {project}', function ($project) {
    $this->info("Building {$project}!");
})->describe('Build the project');
```

Defining Input Expectations

When writing console commands, it is common to gather input from the user through arguments or options. Laravel makes it very convenient to define the input you expect from the user using the `signature` property on your commands. The `signature` property allows you to define the name, arguments, and options for the command in a single, expressive, route-like syntax.

Arguments

All user supplied arguments and options are wrapped in curly braces. In the following example, the command defines one **required** argument: `user`:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user}';
```

You may also make arguments optional and define default values for arguments:

```
// Optional argument...
email:send {user?}
```

```
// Optional argument with default value...
email:send {user=foo}
```

Options

Options, like arguments, are another form of user input. Options are prefixed by two hyphens (`--`) when they are specified on the command line. There are two types of options: those that receive a value and those that don't. Options that don't receive a value serve as a boolean "switch". Let's take a look at an example of this type of option:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue}';
```

In this example, the `--queue` switch may be specified when calling the Artisan command. If the `--queue` switch is passed, the value of the option will be `true`. Otherwise, the value will be `false`:

```
php artisan email:send 1 --queue
```

Options With Values

Next, let's take a look at an option that expects a value. If the user must specify a value for an option, suffix the option name with a `=` sign:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send {user} {--queue=}';
```

In this example, the user may pass a value for the option like so:

```
php artisan email:send 1 --queue=default
```

You may assign default values to options by specifying the default value after the option name. If no option value is passed by the user, the default value will be used:

```
email:send {user} {--queue=default}
```

Option Shortcuts

To assign a shortcut when defining an option, you may specify it before the option name and use a | delimiter to separate the shortcut from the full option name:

```
email:send {user} {--Q|queue}
```

Input Arrays

If you would like to define arguments or options to expect array inputs, you may use the `*` character. First, let's take a look at an example that specifies an array argument:

```
email:send {user*}
```

When calling this method, the `user` arguments may be passed in order to the command line. For example, the following command will set the value of `user` to `['foo', 'bar']`:

```
php artisan email:send foo bar
```

When defining an option that expects an array input, each option value passed to the command should be prefixed with the option name:

```
email:send {user} {--id=*}

php artisan email:send --id=1 --id=2
```

Input Descriptions

You may assign descriptions to input arguments and options by separating the parameter from the description using a colon. If you need a little extra room to define your command, feel free to spread the definition across multiple lines:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'email:send
                        {user : The ID of the user}
                        {--queue= : Whether the job should be queued}';
```

Command I/O

Retrieving Input

While your command is executing, you will obviously need to access the values for the arguments and options accepted by your command. To do so, you may use the `argument` and `option` methods:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $userId = $this->argument('user');

    //
}
```

If you need to retrieve all of the arguments as an `array`, call the `arguments` method:

```
$arguments = $this->arguments();
```

Options may be retrieved just as easily as arguments using the `option` method. To retrieve all of the options as an array, call the `options` method:

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options...
$options = $this->options();
```

If the argument or option does not exist, `null` will be returned.

Prompting For Input

In addition to displaying output, you may also ask the user to provide input during the execution of your command. The `ask` method will prompt the user with the given question, accept their input, and then return the user's input back to your command:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $name = $this->ask('What is your name?');
}
```

The `secret` method is similar to `ask`, but the user's input will not be visible to them as they type in the console. This method is useful when asking for sensitive information such as a password:

```
$password = $this->secret('What is the password?');
```

Asking For Confirmation

If you need to ask the user for a simple confirmation, you may use the `confirm` method. By default, this method will return `false`. However, if the user enters `y` or `yes` in response to the prompt, the method will return `true`.

```
if ($this->confirm('Do you wish to continue?')) {
    //
}
```

Auto-Completion

The `anticipate` method can be used to provide auto-completion for possible choices. The user can still choose any answer, regardless of the auto-completion hints:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Multiple Choice Questions

If you need to give the user a predefined set of choices, you may use the `choice` method. You may set the array index of the default value to be returned if no option is chosen:

```
$name = $this->choice('What is your name?', ['Taylor', 'Dayle'], $defaultIndex);
```

Writing Output

To send output to the console, use the `line`, `info`, `comment`, `question` and `error` methods. Each of these methods will use appropriate ANSI colors for their purpose. For example, let's display some general information to the user. Typically, the `info` method will display in the console as green text:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->info('Display this on the screen');
}
```

To display an error message, use the `error` method. Error message text is typically displayed in red:

```
$this->error('Something went wrong!');
```

If you would like to display plain, uncolored console output, use the `line` method:

```
$this->line('Display this on the screen');
```

Table Layouts

The `table` method makes it easy to correctly format multiple rows / columns of data. Just pass in the headers and rows to the method. The width and height will be dynamically calculated based on the given data:

```
$headers = ['Name', 'Email'];

$users = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $users);
```

Progress Bars

For long running tasks, it could be helpful to show a progress indicator. Using the output object, we can start, advance and stop the Progress Bar. First, define the total number of steps the process will iterate through. Then, advance the Progress Bar after processing each item:

```
$users = App\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
    $this->performTask($user);

    $bar->advance();
}
```

```
$bar->finish();
```

For more advanced options, check out the Symfony Progress Bar component documentation.

Registering Commands

Because of the `load` method call in your console kernel's `commands` method, all commands within the `app/Console/Commands` directory will automatically be registered with Artisan. In fact, you are free to make additional calls to the `load` method to scan other directories for Artisan commands:

```
/**
 * Register the commands for the application.
 *
 * @return void
 */
protected function commands()
{
    $this->load(__DIR__.'/Commands');
    $this->load(__DIR__.'/MoreCommands');

    // ...
}
```

You may also manually register commands by adding its class name to the `$commands` property of your `app/Console/Kernel.php` file. When Artisan boots, all the commands listed in this property will be resolved by the service container and registered with Artisan:

```
protected $commands = [
    Commands\SendEmails::class
];
```

Programmatically Executing Commands

Sometimes you may wish to execute an Artisan command outside of the CLI. For example, you may wish to fire an Artisan command from a route or controller. You may use the `call` method on the `Artisan` facade to accomplish this. The `call` method accepts either the command's name or class as the first argument, and an array of command parameters as the second argument. The exit code will be returned:

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});
```

Alternatively, you may pass the entire Artisan command to the `call` method as a string:

```
Artisan::call('email:send 1 --queue=default');
```

Using the `queue` method on the `Artisan` facade, you may even queue Artisan commands so they are processed in the background by your queue workers. Before using this method, make sure you have configured your queue and are running a queue listener:

```
Route::get('/foo', function () {
    Artisan::queue('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
});
```

You may also specify the connection or queue the Artisan command should be dispatched to:

```
Artisan::queue('email:send', [
    'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

Passing Array Values

If your command defines an option that accepts an array, you may pass an array of values to that option:

```
Route::get('/foo', function () {
    $exitCode = Artisan::call('email:send', [
        'user' => 1, '--id' => [5, 13]
    ]);
});
```

Passing Boolean Values

If you need to specify the value of an option that does not accept string values, such as the `--force` flag on the `migrate:refresh` command, you should pass `true` or `false`:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

Calling Commands From Other Commands

Sometimes you may wish to call other commands from an existing Artisan command. You may do so using the `call` method. This `call` method accepts the command name and an array of command parameters:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
    $this->call('email:send', [
        'user' => 1, '--queue' => 'default'
    ]);

    //
}
```

If you would like to call another console command and suppress all of its output, you may use the `callSilent` method. The `callSilent` method has the same signature as the `call` method:

```
$this->callSilent('email:send', [
    'user' => 1, '--queue' => 'default'
]);
```

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.