Artisan Console

#Introduction

Artisan is the name of the command-line interface included with Laravel. It provides a number of helpful commands for your use while developing your application. It is driven by the powerful Symfony Console component. 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, simply precede the name of the command with `help`:

```
php artisan help migrate
```

#Writing Commands

In addition to the commands provided with Artisan, you may also build your own custom commands for working with your application. You may store your custom commands in the `app/Console/Commands` directory; however, you are free to choose your own storage location as long as your commands can be autoloaded based on your `composer.json` settings.

To create a new command, you may use the `make:console` Artisan command, which will generate a command stub to help you get started:

```
php artisan make:console SendEmails
```

The command above would generate a class at `app/Console/Commands/SendEmails.php`. When creating the command, the `--command` option may be used to assign the terminal command name:

```
php artisan make:console SendEmails --command=emails:send
```

Command Structure

Once your command is generated, you should fill out 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 any command logic in this method. Let's take a look at an example command.

Note that we are able to inject any dependencies we need into the command's constructor. The Laravel service container will automatically inject all dependencies type-hinted in the constructor. For greater code reusability, it is good practice to keep your console commands light and let them defer to application services to accomplish their tasks.

```
<?php

namespace App\Console\Commands;

use App\User;
use App\DripEmailer;
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';

    /**
     * The drip e-mail service.
     *
     * @var DripEmailer
     */
    protected $drip;

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

        $this->drip = $drip;
    }

    /**
     * Execute the console command.
     *
     * @return mixed
     */
```

```
    public function handle()
    {
        $this->drip->send(User::find($this->argument('user')));
    }
}
```

#Command I/O

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.

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 optional arguments:

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

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

Options, like arguments, are also a form of user input. However, they are prefixed by two hyphens (`--`) when they are specified on the command line. We can define options in the signature like so:

```
/**
 * 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
```

You may also specify that the option should be assigned a value by the user by suffixing the option name with a `=` sign, indicating that a value should be provided:

```
/**
 * 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 also assign default values to options:

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

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 Descriptions

You may assign descriptions to input arguments and options by separating the parameter from the description using a colon:

```
/**
 * 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}';
```

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:

To retrieve the value of an argument, use the `argument` method:

```
/**
 * 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 `argument` with no parameters:

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

Options may be retrieved just as easily as arguments using the `option` method. Like the `argument` method, you may call `option` without any parameters in order to retrieve all of the options as an `array`:

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

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

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` in response to the prompt, the method will return `true`.

```
if ($this->confirm('Do you wish to continue? [y|N]')) {
    //
}
```

Giving The User A Choice

The `anticipate` method can be used to provided autocompletion for possible choices. The user can still choose any answer, regardless of the choices.

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

If you need to give the user a predefined set of choices, you may use the `choice` method. The user chooses the index of the answer, but the value of the answer will be returned to you. You may set the default value to be returned if nothing is chosen:

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

Writing Output

To send output to the console, use the `line`, `info`, `comment`, `question` and `error` methods. Each of these methods will use the appropriate ANSI colors for their purpose.

To display an information message to the user, use the `info` method. Typically, this 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 want to display plain console output, use the `line` method. The `line` method does not receive any unique coloration:

```
$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. You have to define the number of steps when you start the progress, then advance the Progress Bar after each step:

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

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

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

Once your command is finished, you need to register it with Artisan so it will be available for use. This is done within the `app/Console/Kernel.php` file.

Within this file, you will find a list of commands in the `commands` property. To register your command, simply add the class name to the list. 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
];
```

#Calling Commands Via Code

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 the name of the command 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'
    ]);

    //
});
```

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:

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

```
    //
});
```

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 may pass a boolean `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'
]);
```