Artisan Development

#Introduction

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.

#Building A Command

Generating The Class

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

Generate A New Command Class

```
php artisan make:console FooCommand
```

The command above would generate a class at `app/Console/Commands/FooCommand.php`.

When creating the command, the `--command` option may be used to assign the terminal command name:

```
php artisan make:console AssignUsers --command=users:assign
```

Writing The Command

Once your command is generated, you should fill out the `name` and `description` properties of the class, which will be used when displaying your command on the `list` screen.

The `fire` method will be called when your command is executed. You may place any command logic in this method.

Arguments & Options

The `getArguments` and `getOptions` methods are where you may define any arguments or options your command receives. Both of these methods return an array of commands, which are described by a list of array options.

When defining `arguments`, the array definition values represent the following:

```
[$name, $mode, $description, $defaultValue]
```

The argument `mode` may be any of the following: `InputArgument::REQUIRED` or `InputArgument::OPTIONAL`.

When defining `options`, the array definition values represent the following:

```
[$name, $shortcut, $mode, $description, $defaultValue]
```

For options, the argument `mode` may be: `InputOption::VALUE_REQUIRED`, `InputOption::VALUE_OPTIONAL`, `InputOption::VALUE_IS_ARRAY`, `InputOption::VALUE_NONE`.

The `VALUE_IS_ARRAY` mode indicates that the switch may be used multiple times when calling the command:

```
InputOption::VALUE_REQUIRED | InputOption::VALUE_IS_ARRAY
```

Would then allow for this command:

```
php artisan foo --option=bar --option=baz
```

The `VALUE_NONE` option indicates that the option is simply used as a "switch":

```
php artisan foo --option
```

Retrieving Input

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

Retrieving The Value Of A Command Argument

```
$value = $this->argument('name');
```

Retrieving All Arguments

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

Retrieving The Value Of A Command Option

```
$value = $this->option('name');
```

Retrieving All Options

```
$options = $this->option();
```

Writing Output

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

Sending Information To The Console

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

Sending An Error Message To The Console

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

Asking Questions

You may also use the `ask` and `confirm` methods to prompt the user for input:

Asking The User For Input

```
$name = $this->ask('What is your name?');
```

Asking The User For Secret Input

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

Asking The User For Confirmation

```
if ($this->confirm('Do you wish to continue? [yes|no]'))
{
    //
}
```

You may also specify a default value to the `confirm` method, which should be `true` or `false`:

```
$this->confirm($question, true);
```

Calling Other Commands

Sometimes you may wish to call other commands from your command. You may do so using the `call` method:

```
$this->call('command:name', ['argument' => 'foo', '--option' => 'bar']);
```

#Registering Commands

Registering An Artisan Command

Once your command is finished, you need to register it with Artisan so it will be available for use. This is typically done in 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 it to this list.

```
protected $commands = [
    'App\Console\Commands\FooCommand'
];
```

When Artisan boots, all the commands listed in this property will be resolved by the service container and registered with Artisan.