Stillat
(https://stillat.com/)

The Laravel Framework Console Kernel

December 1, 2016 Laravel (/explore/categories/laravel)

JOHN KOSTER

The console kernel exposes many different public methods. This article will not cover *all* of the public methods available, but only the most useful ones.

`all`

The `all` method is used to get all of the commands that have been registered with the console application. The commands will be returned as an array with the command name as the key and the command's class instance as the value. This method will bootstrap the application as well as force the loading of deferred service providers.

The following example demonstrates one way to call this method:

```
<?php

use Illuminate\Contracts\Console\Kernel;
use Illuminate\Support\Facades\Artisan;

// Get a console instance.
$console = app(Kernel::class);

// Get the registered console commands.
$registeredCommands = $console->all();


// The following facade method would be equivalent
$registeredCommands = Artisan::all();
```

After the above code has executed, the `$registeredCommands` variable would be an array containing all of the console commands that have been registered.

`registerCommand($command)`

The `registerCommand` method is used to register a command with the console application. In older versions of Laravel, it was required to use this method to register any custom commands with the console kernel. The `registerCommand` method expects a `$command` argument to be supplied; the supplied `$command` must ultimately be an instance of `Symfony\Component\Console\Command\Command` (this means that any Symfony command or Laravel derived command would be acceptable).

The following example demonstrates the `registerCommand` method usage:

```php
<?php

use Illuminate\Support\Facades\Artisan;
use Illuminate\Contracts\Console\Kernel;
use App\Console\Commands\Inspire;

// Get a console instance.
$console = app(Kernel::class);

// Register the default `Inspire`
// command with the application:
$console->registerCommand(app(Inspire::class));

// The following facade method would be equivalent:
Artisan::registerCommand(app(Inspire::class));
```

After the above code has executed, the `Inspire` command would be available to the console application. We could check to ensure it's existence by using the `all` method:

```php
<?php

// Determine if the `Inspire` command was registered or not.
$inspireRegistered = array_key_exists(
    'inspire',
    $console->all()
);
```

The above coder would check to see if a particular console command has been registered with the application. It does this by checking to see if the name of the command is included in the array returned by the kernel's `all` method. Since the `all` method returns an array where the registered command names are the keys, this would be sufficient for an existence test.

call($command, array $parameters = [])

The `call` method is used to execute an Artisan command from somewhere else in your applications code. It accepts the name of the command via an argument supplied for the `$command` parameter and an array of `$parameters` that should be supplied to the command. The exit code returned by the command will be the return value of the `call` method.

The following simple example calls the `inspire` (assuming it has been registered) command from some application code:

```php
<?php

use Illuminate\Contracts\Console\Kernel;
use Illuminate\Support\Facades\Artisan;

// Get a console instance.
$console = app(Kernel::class);

// Execute the inspire command:
$console->call('inspire');

// The following facade method would be equivalent:
Artisan::call('inspire');
```

The `inspire` command outputs a random inspiration quote. The outputted quote would *not* be returned from the `call` method. The `output` method would be used to get the output from the last ran Artisan command:

```php
<?php

// Get the output from the last command.
$output = $console->output();

// The following facade method would be equivalent:
$output = Artisan::output();
```

Assuming the last ran command was the `inspire` command, the `$output` variable would contain one of the randomly chosen inspiration quotes.

The following example demonstrate how to execute Artisan commands while also supplying arguments and options. Arguments are supplied as an array to the `$parameters` parameter. The supplied array should contain the name of the argument or option as the key and the associated value as the value for the given key.

We can take the following Artisan command that would normally be executed at the command line:

```
php artisan make:migration create_drinks_table --path=database/setup_migrations
```

And call it directly form our application like so:

```php
<?php

use Illuminate\Contracts\Console\Kernel;
use Illuminate\Support\Facades\Artisan;

// Get a console instance.
$console = app(Kernel::class);

// Generate a new migration from within our
// application using the `make:migration`
// Artisan command.
$console->call('make:migration', [
    'name'   => 'create_drinks_table',
    '--path' => 'database/setup_migrations'
]);
```

It is important to note that the name of the argument or option must be supplied as the key if they are used, even if they are not required when executing the command from the terminal. The names of options must also start with the `--` prefix.

output

The `output` method is used to retrieve the generated output from the Artisan console command that was executed last using the `call` method. The following example will assume that the `inspire` command has been registered.

First we need to `call` the `inspire` command:

```php
<?php

use Illuminate\Contracts\Console\Kernel;
use Illuminate\Support\Facades\Artisan;

// Get a console instance.
$console = app(Kernel::class);

// Execute the inspire command:
$console->call('inspire');

// The following facade method would be equivalent:
Artisan::call('inspire');
```

To retrieve the output from the `inspire` command, we can use the `output` method:

```php
<?php

// Get the output from the last command.
$output = $console->output();

// The following facade method would be equivalent:
$output = Artisan::output();
```

After the above code has executed, the `$output` variable would contain the output from the `inspire` command, which should be a randomly selected inspirational message.

> The `output` method does not remove any newline characters or special characters from the returned output. It is important to keep this mind when presenting command output to users.

`queue($command, array $parameters = [])`

The `queue` method is called in exactly the same way as the `call` method. It accepts the name of the command via an argument supplied for the `$command` parameter and an array of `$parameters` that should be supplied to the command. The exit code returned by the command will be the return value of the `call` method. Just like with the `call` method, the name of arguments and options must be supplied as the key if they are used, even if they are not required when executing the command from the terminal. The names of options must also start with the `--` prefix.

The major difference between the `queue` and `call` methods is that the `queue` method will cause the Artisan command to be processed in the background by the configured queue workers. The following example demonstrates how to call the `queue` method (the same example will be used from the `call` method section; generally more intensive tasks would be queued instead of generating migrations, such as the sending of emails):

```php
<?php

use Illuminate\Contracts\Console\Kernel;
use Illuminate\Support\Facades\Artisan;

// Get a console instance.
$console = app(Kernel::class);

// Generate a new migration from within our
// application using the `make:migration`
// Artisan command.
$console->queue('make:migration', [
    'name'   => 'create_drinks_table',
    '--path' => 'database/setup_migrations'
]);

// The following facade method would be equivalent:
Artisan::queue('make:migration', [
    'name'   => 'create_drinks_table',
    '--path' => 'database/setup_migrations'
]);
```


Up Next

Writing Custom Laravel Arti...
It is often very useful to create custom Artisan commands specifically for...
(/blog/2016/12/01/writing-custom-laravel-artisan-commands-an-introduction)

An Introduction to Laravel'...
The Artisan Command Line Environment (CLI) is a terminal based application...
(/blog/2016/11/30/an-introduction-to-laravels-artisan-console)

Creating a Hashing Manager ...

To make it easier to easily work with all the different hashing...
(/blog/2016/11/30/creating-a-hashing-manager-for-our-custom-laravel-hashing-implementations)

Creating a Service Provider...
In this article, we will create a service provider for all all of the...
(/blog/2016/11/30/creating-a-service-provider-our-custom-hashing-implementations)