LARAVEL ARTISAN

In depth coverage of Laravel features.

JOHNATHON KOSTER

Laravel Artisan

An in depth coverage of Laravel 5.3 features

Johnathon Koster

This book is for sale at http://leanpub.com/laravelartisan

This version was published on 2016-08-24

Tweet This Book!

Please help Johnathon Koster by spreading the word about this book on Twitter!

The suggested hashtag for this book is #laravelphp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#laravelphp

Contents

About the Book

The book started as a collection of personal notes that have been growing while working on both professional and personal projects using the Laravel PHP framework. These notes mainly focus on the nuances in the framework's features as well as extensive coverage of the helper functions, classes and features. Many of these notes can be found in fragmented versions on my personal website (http://www.stillat.com/[1]). This book is a cohesive collection and expansion of said notes (not all the content in the book is available on the website).

Why?

When this project (the collection of notes, and eventually this book) started back in the Laravel 4.x days, the documentation was not as great as it is today (a lot of hard work and effort has gone into the recent Laravel documentation by many talented and dedicated people). The following collection of notes were created as a sort of supplemental documentation to reinforce ideas and concepts learned about the framework while developing projects and ideas.

Who is This Book For?

This book is for anyone who uses the Laravel framework to create things, or anyone who is interested in learning more about the framework. A basic knowledge of PHP is recommended as well as some experience with frameworks like Laravel. There are some sections that dig deep into some underlying framework implementation details; you have been warned. Here be dragons.

How Is The Book Structured?

Each chapter was designed to hopefully stand out on its own. This was done to reduce the need to flip back and forth between pages repeatedly. However, the chapters and sections are organized so that the sections flow in a logical order (for example, `set` and `get` method sections usually appear next to one another).

The table of contents is also very detailed, often including function and method signatures as part of the listing. This book does not include an index, and structuring the table of contents in this way allows readers to quickly find a method or function and the corresponding page number to look up.

The book also includes appendices that provide supplemental information to the contents of the book. For example, completed appendices (at the time this introduction was written) include a list of available hashing functions available for use with PHP's `crypt` function as well as a list of common HTTP status codes.

[1] http://www.stillat.com/

Symbols Used In This Book

This icon will be used when defining a term. This icon is not used often, and usually only applied when there may be confusion about or term, or when sufficient ambiguity already exists around a term.

This icon is used to present the thoughts or opinions of the author(s) and acts as an invitation for conversation. These sections also feature a title styling that is more subtle.

Look for this icon in the text to learn about any appendices that may additional reference material for a given section.

This icon indicates information that is important or the cause of general errors.

This icon is used to point out or explain the cause of specific errors or problems that may arise during development.

This icon is used to point out additional information that might be useful in the context of a given chapter, but is not required.

This icon is used to provide additional contextually-relevant information to help clarify ideas or to point out multiple ways of doing something.

Is This Book Done?

In a word, no. The level of detail most of the sections get into can take a very long time to research, test and write. I believe that the book, as it sits, contains a wealth of information as is and should be released. This will allow me to gauge interest in the book and receive feedback. The received interest and feedback will hopefully provide motivation and justification for the continued work on this book (these things take a *massive* amount of time).

The short answer is that, yes, there are plans to cover all the major features of the Laravel framework at later dates. Which features covered will largely depend on reader interest and requests. Each section will attempt to cover each topic completely as well as point out any inconsistencies, nuances or hidden features related to a given feature set.

The following are the chapters and sections that I am currently working next:

- Artisan Console: Interacting With Commands (chapter)
- Artisan Console: Scheduling Commands (chapter)
- Artisan Console: Tinker (chapter)

After these sections have been written I intend to write the “beginning” of the book. This will cover an introduction to Laravel, Homestead, Valet, etc.

What Was Used to Write This Book?

The book publication and rendering was done using Leanpub (https://leanpub.com/[2]). The book was written using a custom variant of LeanPub’s markdown that is more suited for writing large programming books (with specific tooling for writing about Laravel). The actual writing took place in Adobe Brackets (http://brackets.io/[3]) using custom markdown and spell checking plugins (the spell checking plugin is available for free on GitHub[4]).

[2]https://leanpub.com/
[3]http://brackets.io/
[4]https://github.com/JohnathonKoster/brackets-spellcheck

Here be dragons!

```
                                                 .~))>>
                                                .~)>>
                                              .~))))>>>
                                            .~))>>             ___
                                          .~))>>)))>>      .-~))>>
                                        .~)))))>>       .-~))>>)>
                                      .~)))>>))))>>  .-~)>>)>
                  )                 .~))>>))))>>  .-~)))))>>)>
               ( )@@*)             //)>))))))  .-~))))>>)>
             ).@(@@               //))>>))) .-~))>>)))))>>)>
           (( @.@).              //))))) .-~)>>)))))>>)>
         ))  )@@*.@@ )          //)>))) //))))))>>))))>>)>
      ((  ((@@@.@@             |/))))) //)))))>>)))>>)>
     )) @@*. )@@ )   (\_(\-\b  |))>)) //)))>>)))))))>>)>
   (( @@@(.@(@ .    _/`-`  ~|b |>))) //)>>)))))))>>)>
    )* @@@ )@*     (@)  (@) /\b|))) //))))))>>))))>>
  (( @. )@( @ .   _/  /    /  \b)) //))>>)))))>>>_._
   )@@ (@@*)@@.  (6///6)- / ^  \b)//))))))>>)))>>   ~~-.
( @jgs@@. @@@.*@_ VvvvvV//  ^  \b/)>>))))>>      _.     `bb
 ((@@ @@@*.(@@ . - | o |' \ (  ^   \b)))>>        .'       b`,
  ((@@).*@@ )@ )   \^^^/  ((   ^  ~)_        \  /           b `,
    (@@. (@@ ).     `-'   (((   ^    `\ \ \ \ \|             b  `.
      (*.@*              / ((((        \| | |  \       .       b `.
                        / / (((((  \    \ /  _.-~\     Y,      b  ;
                       / / / (((((( \    \.-~   _.`" _.-~`,    b  ;
                      /   /   `(((((()    )    (((((~      `,  b  ;
                    _/  _/      `"""/   /'                  ; b   ;
                _.-~_.-~           /  /'                _.'~bb _.'
              ((((~~              / /'              _.'~bb.--~
                                 ((((          __.-~bb.-~
                                             .'  b .~~
                                             :bb ,'
                                             ~~~~
```

Laravel: The Basics

"I learned very early the difference between knowing the name of something and knowing something."

– Richard P. Feynman (1918 - 1988), Theoretical Physicist

1. Introduction

In this chapter we will begin the process of learning about the Laravel framework, and how it operates on a higher level before diving into various framework features in detail. So what exactly is Laravel? Doing a quick search for “What is Laravel” yields results that repeat some variation of the phrase “a framework that follows the model-view-controller (MVC) architectural pattern”. While this is true, it assumes that the reader has a rather large knowledge base of software design; this may include things such as the design, organization and architecture of web software systems. While this knowledge is definitely a benefit when approaching frameworks like Laravel, it is best to attempt to describe things as simply as possible when introducing any technology. And more often than not, it can be found that the reader does in fact *know* more than they thought, it’s just that they could not apply a specific name to a thing. To this end, Laravel can be described as a well thought out, opinionated collection of individual pieces of software that all elegantly work together to help make the process of building web applications easier.

Thoughts on Opinionated Software Development

It is common to find criticisms of opinionated software development throughout online discussion channels; some of the arguments against it are valid. However, it is important to understand that it is very difficult, if not impossible, to develop a cohesive and consistent software framework without some opinions on how the components of the system should interact. While there may be many opinions in how the various components of the Laravel framework work and interact, there are no artificial limits put into place that prevents you from pursuing another way of solving a problem. In fact, it can be argued that Laravel excels in getting out of the way when it has to.

The take away: there is no “Laravel way” of accomplishing tasks; it is all mostly PHP after all.

To help achieve it’s goal of making web development easier and more elegant, Laravel often exposes many of its own, simpler APIs and systems to accomplish common web development tasks. Most of these APIs are designed to be easy to read and understandable, even if the naming of functions and methods sacrifice brevity. Under the guidance of this philosophy, Laravel often provides a very elegant method syntax. For example, the following might be used to retrieve all the users with an active account from some database system (the specific details on implementing such a code example will follow in later chapters):

```php
<?php

use App\User;

// Retrieve all users with an active status of `true`.
$user = User::where('active', '=', true);
```

The style of syntax used in the previous example can be found throughout many of the consumable framework APIs (here consumable framework API is referring to any framework API that is intended to be used by developers). The system used in that code example is the Eloquent object-relational mapping (ORM) system, a simple ActiveRecord implementation for interacting with database systems, or other data storage systems. It is just one of the many default systems that Laravel provides.

The following sections briefly highlight some of the offerings of the Laravel framework:

Configuration Systems
: One thing that becomes quickly apparent when working with the Laravel framework is how easily customizable it is; one of the major features for accomplishing this is the configuration systems that are available as part of the core system. Laravel provides a simple, streamlined API for accessing and modifying configuration values throughout your application. This configuration system also supports multiple application environments.

Error and Logging Systems
: When developing applications, it is very convenient to be able to easily report and log errors and events as the application is running. Laravel makes it easy to log to a single file, multiple log files, configure rotating log files as well as writing error information to the system log.

Routing
: Routes are, fundamentally, a way to provide controlled and explicit access to your application; routes map a URI to some action within your application. Laravel provides a very powerful routing system that is capable of handling complex database situations, URI grouping and ways to easily create REST APIs, amongst many other powerful features.

HTTP Abstractions
: Laravel, building on top of the Symfony framework's battle-tested feature-set, provides convenient APIs for accessing and managing cookie, session and request data.

Template Engines and Language Localization
: Templates, or "views", are the primary way in which users interact will all web applications and are primarily composed of HTML, CSS and JavaScript files and other additional assets. Laravel ships with a powerful template language, Blade, as well as provides mechanisms to write templates in PHP or to even create a custom template engine. In addition, there are services and features in place to help ease the process of localizing application content.

Database Management and Access Abstractions
Interacting with database systems can be a complex task, and Laravel makes it incredibly easy with its database abstraction layer as well as its Eloquent ORM. Additionally, there are database management features available for building the actual structure of the database as well as populating database tables with testing data.

Authentication and Authorization
Authentication and authorization, while related, are two separate problems in their own right. Laravel provides features for both: authentication drivers and APIs handle allowing or restricting access to your application to only certain users. On the other hand, the authorization features make it easy to restrict access to specific content of features within the application, once authenticated.

Hashing and Encryption
Correctly creating and managing hashes of text can be a complicated task; Laravel makes it incredibly easy to accomplish both. It provides consistent hashing and encryption features that are used throughout the framework to implement features such as cross-site request forgery (CSRF) protection, cookie and session signing as well as provide the hashing mechanisms to authenticate users.

While the previous list of features is impressive in its own right, it is only the beginning of what Laravel offers developers to create elegant and feature-rich web applications. The remaining sections in this chapter will cover topics that are helpful when learning the Laravel framework, such as namespaces, autoloading, the Composer autoloader as well as other various topics.

If you are comfortable with the topics presented in each of these sections, feel free to skip them. On the other hand, if any of the subjects presented prove difficult, it may be beneficial to learn a little more about each of the subjects before embarking on the journey of learning the Laravel framework in detail.

This book is an attempt to cover all major framework features in detail. Throughout the journey there will be many different ideas introduced, new terminology used and the occasional rabbit trail will be explored. Each section of the book has been designed to stand apart on its own, separate from the others, but great efforts have been made to ensure the continuous flow of thought throughout the entire work.

1.1 What You Should Know

The vast majority of this book will assume that the you has a foundational understanding of the following topics; by no means does this mean that you must be an expert in any of this areas. If any of the following topics are difficult, I would recommend doing a little research on each of these topics to at least understand them conceptually; the exact specifics in each area will come over time through discipline and practice.

1.1.1 PHP and Object-Oriented Programming (OOP) Concepts

Before learning Laravel, or any PHP framework, it is recommended to have at least a basic understanding of the PHP programming language, or extensive experience with another object-oriented programming (OOP) language. Many of the features that Laravel provides are often abstractions built on top of libraries and features that are available in the core PHP framework (here framework is referring to PHP's extensive function library as well as PHP's Standard PHP Library (SPL) offerings).

1.1.2 A Little Linux

Every developer should know how to, at the very absolute least, navigate through and make changes to files on Linux-based operating systems. The book Servers for Hackers[1] by Chris Fidao is a great resource for anyone looking to learn more about managing servers, especially those coming from a programming background.

The following commands can be used to navigate the file system when working with the Linux command line; these commands will be able to get you through most of the situations when getting started with Linux-based operating systems (remember: pretty much everything in Linux-based operating systems are files, so learning how to work with the file system, at the very minimum, is an absolute requirement). The more advanced commands, and all the options that are available for these commands can be learned over time as the need for them arises; as Jeffery Way once said in his podcast The Laracasts Snippet[2] episode 38 "You Are Who You Say You Are[3]": "fake it 'till you make it".

The examples in this section (and throughout most of the book) are going to assume the use of the Bourne Again Shell (Bash).

1.1.2.1 The `man` Command

While not a file system management command, the `man` command is one of the most useful commands to know when working with Linux-based operating systems. This command will display the help information for other commands, which can be supplied as an argument to the first parameter:

```
# View the help information for the `ls` command.
man ls
```

1.1.2.2 The `cd` Command

The `cd` (change directory) command is what is used to actually move around Linux-based operating systems. It is the command that you use to essentially tell the operating system where to go.

For example, the following could be used to change the current directory to a `Homestead` directory:

[1] https://leanpub.com/serversforhackers
[2] https://laracasts.simplecast.fm/
[3] https://laracasts.simplecast.fm/38

```
# Change the current directory to `Homestead`
cd Homestead
```

1.1.2.3 The `ls` Command

The `ls` command is used to list the contents of a directory. The following examples demonstrate how to use the `ls` command to view directory contents:

```
# Change the current directory to `Homestead`
cd Homestead

# List the contents of a `Homestead` directory:
ls
```

After the above command has been issued, depending on the exact directory structure, the following output might appear (another thing to keep in mind is that most commands will not return any output if they were executed successfully):

```
LICENSE.txt  Vagrantfile     composer.json   composer.lock
homestead    init.bat    init.sh     readme.md   scripts     src
```

Additional information can be retrieved about the file system by passing in options. The `l` option can be used to show extra information about each file (or directory), such as the file mode, number of links, owner name, group name, etc (the full list of what is displayed can be learned by using the `man` command):

```
John:Homestead john$ ls -l
total 72
-rw-r--r--   1 john  staff  1077 Mar 20 20:11 LICENSE.txt
-rw-r--r--   1 john  staff   921 Mar 20 20:11 Vagrantfile
-rw-r--r--   1 john  staff   603 Mar 20 20:11 composer.json
-rw-r--r--   1 john  staff  6245 Mar 20 20:11 composer.lock
-rwxr-xr-x   1 john  staff   317 Mar 20 20:11 homestead
-rw-r--r--   1 john  staff   311 Mar 20 20:11 init.bat
-rw-r--r--   1 john  staff   270 Mar 20 20:11 init.sh
-rw-r--r--   1 john  staff   151 Mar 20 20:11 readme.md
drwxr-xr-x  14 john  staff   476 Mar 20 20:11 scripts
drwxr-xr-x   4 john  staff   136 Mar 20 20:11 src
```

Options can be combined, just like when calling Artisan commands. The a option can be added to show hidden files (or dot files):

```
John:Homestead john$ ls -la
total 88
drwxr-xr-x  16 john  staff   544 Mar 20 20:21 .
drwxr-xr-x+ 46 john  staff  1564 Aug 23 11:13 ..
drwxr-xr-x  13 john  staff   442 Mar 20 20:11 .git
-rw-r--r--   1 john  staff    14 Mar 20 20:11 .gitattributes
-rw-r--r--   1 john  staff    18 Mar 20 20:11 .gitignore
drwxr-xr-x   3 john  staff   102 Mar 21 21:25 .vagrant
-rw-r--r--   1 john  staff  1077 Mar 20 20:11 LICENSE.txt
-rw-r--r--   1 john  staff   921 Mar 20 20:11 Vagrantfile
-rw-r--r--   1 john  staff   603 Mar 20 20:11 composer.json
-rw-r--r--   1 john  staff  6245 Mar 20 20:11 composer.lock
-rwxr-xr-x   1 john  staff   317 Mar 20 20:11 homestead
-rw-r--r--   1 john  staff   311 Mar 20 20:11 init.bat
-rw-r--r--   1 john  staff   270 Mar 20 20:11 init.sh
-rw-r--r--   1 john  staff   151 Mar 20 20:11 readme.md
drwxr-xr-x  14 john  staff   476 Mar 20 20:11 scripts
drwxr-xr-x   4 john  staff   136 Mar 20 20:11 src
```

1.1.2.4 The `mkdir` Command

The `mkdir` command is used to create (or make) directories on the file system. It also has a very useful `p` option that will create directories recursively. The following examples demonstrate its usage:

```
# Create a new `app` directory
mkdir app

# Recursively create many directories
mkdir -p app/Http/Controllers
```

Typically, in my experience, the command that is most often called after the `mkdir` command is the `cd` command (usually combined using the && boolean operator):

```
mkdir app && cd app
```

This will create a new directory and then change into it. This is manageable for short directory names, but can quickly become inconvenient when creating larger directory structures. We can use the `$_` bash parameter, which will return the arguments supplied to the previous command to make this easier:

We surrounded the `$_` special parameter in quotes when issuing the `cd` command to account for spaces in the directory path.

```
# Create a nested directory structure and then change into it.
mkdir -p very/nested/directory/structure && cd "$_"
```

1.1.2.5 The `touch` Command

The `touch` command can be used to create a new file, if it does not exist. If a file does exist, it will update the files modification and access times.

```
# Create a new file.
touch filename.txt

# Since the file exists, update its modification and access times.
touch filename.txt

# Update only the modification time.
touch -m filename.txt

# Update only the access times.
touch -a filename.txt
```

1.1.2.6 The `rm` Command

The `rm` (or `unlink` command, if you prefer longer command names) is used to remove file entries from a directory (there is also `rmdir` command that is used specifically to remove directories, but the `rm` command can do both). The following examples demonstrate the usage of the `rm` command:

```
# Remove a file
rm filename.txt

# Remove a directory recursively
rm -r directory/path
```

1.1.2.7 Interesting Names

A PHP developer who has any experience interaction with the file system within their applications will notice that a lot of PHP's functions are named similarly to the command names presented in this section, and that they do most of the same things. This is not an accident, and this pattern can be seen throughout many of the functions present in PHP's framework. A little knowledge of Linux-based operating systems will go a long way when learning PHP in general, not just when working with Laravel.

1.1.3 Composer

Composer[4] is a dependency manager for PHP. It allows for the seamless reuse of existing, primarily open-source, software packages. Most of these software packages are hosted on services like GitHub[5]. Composer is primarily used to use the dependencies (things like libraries or frameworks) that any particular project requires (this is in contrast to package managers such as PEAR, which installs packages on a system-wide basis).

At the end of the day, Composer just makes it easier to use various software frameworks and libraries inside your project without too much thinking and tinkering involved. Take some time to read through the documentation located at https://getcomposer.org/doc/[6]. It is well written, and includes instructions on how to install it many different platforms. The rest of the book will assume a working installation of Composer as well as some basic knowledge Composer.

1.1.4 Namespaces and Auto-Loading

Namespaces are tool available to programmers to encapsulate items within their project. The encapsulation of items is useful for organizing a project, enforcing branding and to avoid excruciatingly long class and function names. A quick search of the Laravel code-base for `Manager` returns the following list of similarly named files:

- `src/Illuminate/Support/Manager.php`
- `src/Illuminate/Queue/Capsule/Manager.php`
- `src/Illuminate/Database/Capsule/Manager.php`

Each of these files are stored within the `src/` directory, neatly within their own *structured* subdirectory. If we were to open the `src/Illuminate/Support/Manager.php` file, as an example, we would see the following code at the beginning of the file:

```php
<?php

namespace Illuminate\Support;

use Closure;
use InvalidArgumentException;

abstract class Manager
{
    /**
     * The application instance.
     *
     * @var \Illuminate\Foundation\Application
```

[4]https://getcomposer.org/
[5]https://github.com/
[6]https://getcomposer.org/doc/

```
14      */
15     protected $app;
16     /**
17      * The registered custom driver creators.
18      *
19      * @var array
20      */
21     protected $customCreators = [];
22 
23     // ...
24 
25 }
```

On line three, we see the expression `namespace Illuminate\Support;`. This is stating that any classes, interface, function, etc definitions that appear within this file belong to the `Illuminate\Support` namespace. When looking at the `Illuminate\Support` namespace, we can see that there are two different strings separated by the \ sigil. This is because namespaces can be nested; that is, namespaces can belong to other namespaces, just like a file system can have nested directories, and those nested directories can have their own subdirectories.

In fact, when looking at the directory structure for the source code, we can see that the directory structure almost exactly mirrors the namespace structure. This is not an accident; there are a few very good reasons for this. The first, and probably most important, reason you would want to have some symmetry between the namespace structure and the project's directory structure is that it makes it very easy to locate files both within the file system and when interacting with them through application code.

Another excellent reason to have a structured and organized relationship between namespaces and directory structures is it allows for the creation and utilization of autoloading systems (the PSR-4: Autoloader[7] standard is what Laravel implements with its namespaces and directory structure). The autoloading of classes in PHP allows us developers to not have to worry about when classes are available to use within our code. If a class, or any other item, can be *found* by an autoloader system, it can be used within our code. Code is made available when it is used, without having to explicitly `include` or `require` the file that contains the classes that we wish to use.

On lines five and six, there are two expressions that begin with the `use` operator. The `use` operator is used to import some class, interface, function, etc that exists in some other namespace into the current namespace. It can be thought of as "I am going to use this other thing inside the file I am currently working in" or "I am going to make class X visible to some other class Y". That is what is occurring on lines five and six in the previous example. On line five, the `use Closure` statement will make the `Closure` class available within the current file, and line six will make the `InvalidArgumentException` class available within the current file. If we had not done this, an error stating that a class with name `Illuminate\Support\Closure` or `Illuminate\Support\InvalidArgumentException` could not be found, since PHP will look in the current namespace to resolve any class names (both the `Closure` and `InvalidArgumentException` classes are defined in the *root* or *global* namespace). On the surface, this may

[7]http://www.php-fig.org/psr/psr-4/

seem similar to using PHP's `require` or `import` functions, but there are some additional features that make the `use` operator very powerful (as well as being understood by autoloading systems).

One of the most powerful features of namespaces is the ability to alias imported class, function or namespaces. This is useful when importing classes, functions or namespaces whose names might conflict with the current namespace or entity name (here entity will refer to the class, function or namespace). There will be times when we are required to import an entity with the same name as the class or interface that exists in the current file (there are only so many good names for classes and interfaces after all). While this may not seem like something that would happen very often, it comes up fairly often, especially when coding to an interface (as most of the Laravel core does) or when utilizing code from third parties (such as when adding project dependencies using dependency manager tools, such as Composer). To get around these naming conflicts, PHP allows us to *alias* the classes, functions and interfaces we are importing; i.e, we have the ability to rename classes, functions and interfaces to whatever we want without modifying the source code of those things.

The following examples demonstrate the different aliasing features and scenarios:

```php
<?php

namespace App;

// Import the `Auth` facade.
use Illuminate\Support\Facades\Auth;

// Import the `Auth` facade; give it an alias.
use Illuminate\Support\Facades\Auth as AuthFacade;

// Import a global class (from the global or root namespace).
use Closure;

// Import a function (example only).
use function App\Support\functionName;

// Alias a function.
use function App\Support\functionName as betterFunctionName;

// Import a constant (declared using the const keyword).
use const App\Support\someConstant;
```

Fully Qualified Names

There will be times, in conversation, or when consulting learning materials that the term *fully qualified name* or FQN will come up. A FQN refers to the complete name of a class, function, or interface starting from the global namespace. Often times, the FQN for something will begin with the \ sigil; e.g, the FQN of the `Closure` class is `\Closure`; alternatively, the FQN of the abstract base Eloquent model is `\Illuminate\Database\Eloquent\Model`.

1.1.5 Traits

PHP is an object-oriented language, and as such, it supports the concept of inheritance. Inheritance allows us developers to design abstract or base classes that can be other objects can *inherit* properties and methods from. One of the most widely used base classes from the Laravel framework is the base Eloquent model (`Illuminate\Database\Eloquent\Model`). Using inheritance, we gain the ability to reuse many, or all, of the features provided by the base (referred to as a parent or superclass) in our own classes (referred to as a child or subclass); the main goal of inheritance is to provide a mechanism for code reuse. However, many languages, including PHP, do not allow any given subclass to inherit from more than one parent class; there are many good reasons for this, many of which are out of scope for this book. The paper CZ: Multimethods and Multiple Inheritance Without Diamonds[8] by Donna Malayeri and Jonathan Aldrich provides an excellent overview of the problems associated with multiple inheritance. However, there are times when it is beneficial to reuse code without inheriting from a base class; in PHP the mechanism to accomplish this are Traits[9].

Traits are functionally a language assisted code copy and paste feature, with benefits. Traits are similar to classes, in that they can define methods and properties, but they cannot be instantiated on their own; they are defined with the `trait` keyword rather than the `class` keyword. A great way to mentally think about the difference is that classes define objects while traits *add* new abilities to that class. Common use cases for traits are to provide a common implementation of a given interface (again, without the need for inheritance), or even to "mark" a given class as having some extra ability to treat it differently (this could include things such as marking a class as `Queueable` or `Serializable`, for instance); Laravel provides many helper utilities for interacting and working with traits. Classes that use traits are often referred to as the *exhibitor* class, a class that exhibits some property or ability that the trait provides.

The following is an example trait from the Laravel code-base:

```
1  <?php
2
3  namespace Illuminate\Notifications;
4
5  trait Notifiable
6  {
7      use HasDatabaseNotifications, RoutesNotifications;
8  }
```

Examining the code, we see that on line three we have the namespace declaration (traits can also be namespaced), followed by the trait definition on line five. On line seven is where things get interesting since we are reusing the `use` operator in a different context. When the `use` operator is used within a class, PHP interprets this as referencing a trait. This example also shows that traits can also reference other traits (traits use other traits, rather than extend them; traits have no concept of inheritance). The following is the code for the `HasDatabaseNotifications` (which is referenced on line seven) trait:

[8]http://www.cs.cmu.edu/~donna/public/malayeri.TR09-153.pdf

[9]http://php.net/manual/en/language.oop5.traits.php

```
 1 <?php
 2 
 3 namespace Illuminate\Notifications;
 4 
 5 trait HasDatabaseNotifications
 6 {
 7     /**
 8      * Get the entity's notifications.
 9      */
10     public function notifications()
11     {
12         return $this->morphMany(DatabaseNotification::class, 'notifiable')
13                             ->orderBy('created_at', 'desc');
14     }
15     /**
16      * Get the entity's unread notifications.
17      */
18     public function unreadNotifications()
19     {
20         return $this->morphMany(DatabaseNotification::class, 'notifiable')
21                             ->whereNull('read_at')
22                             ->orderBy('created_at', 'desc');
23     }
24 }
```

Here we can see a trait that is defining its own methods. When a class uses this trait, it will also gain access to all the methods and properties defined in this class. Within the method bodies, we can see that the trait is referencing `$this` as if it were a class. This is because when the code methods and properties within a trait are evaluated and executed, they will be doing so within the context of the exhibitor class. On lines 12 and 21, we can see that the trait is making a call to a `morphMany` method that it does not define itself. This method must be supplied be either another trait, or the exhibitor class itself. The take away here is that traits can be written with the assumption of a particular type of exhibitor class (in the case of the `HasDatabaseNotifications` trait, the intended exhibitor class would be any instance of the `Illuminate\Database\Eloquent\Model` class).

While PHP does not allow multiple inheritance it does allow for classes to use more than one trait. Because of this, there can be times when multiple traits are used that define methods with the same name. When this occurs, a fatal error is generated because PHP cannot decide which method should be used (in essence, there is some ambiguity that cannot be explicitly resolved). To overcome this, we can indicate which method implementation to use, or even use both by aliasing a method on the trait itself.

The following examples demonstrate how to resolve method name ambiguity.

```php
<?php

trait FirstTrait
{

    public function conflictingMethod()
    {
        return 'Hello';
    }

}

trait SecondTrait
{

    public function conflictingMethod()
    {
        return 'Goodbye';
    }

}

class HarmoniousResolution
{
    use FirstTrait, SecondTrait {
        // Use the conflictingMethod() from FirstTrait instead
        FirsTrait::conflictingMethod insteadof SecondTrait;

        // Use the conflictingMethod() from SecondTrait instead.
        SecondTrait::conflictingMethod insteadof FirstTrait;

        // Use both of them, but one must be renamed.
        FirstTrait::conflictingMethod as nonConflictingMethod;
    }
}
```

In the previous example, the `insteadof` operator is used to indicate which which method should take precedence. The method on the left side of the operator will be used over (or instead of) any method with the same name that exists within the trait on the right hand side of the operator.

In some more advanced scenarios, it may be required to actually change the visibility of the methods exhibited from a trait. This is done supplying a method visibility keyword after the `as` operator:

```php
<?php

// ...


class HarmoniousResolution
{
    use FirstTrait {
        // Change the visibility of conflictingMethod to protected.
        conflictingMethod as protected;

        // Change the visibility of conflictingMethod to private.
        conflictingMethod as private;

        // Change the visibility of conflictingMethod to public.
        conflictingMethod as public;
    }
}
```

There is much more that can be accomplished with traits that what has been discussed here. However, the ideas presented in this section should be enough to cover most of the scenarios and use-cases encountered when working with the Laravel framework.

Laravel provides localization features to help retrieve strings in various languages. These features will collectively be referred to as the "translation component" throughout the remainder of this section. The translation component has the following sub-components:

- `Illuminate\Translation\Translator`
- `Illuminate\Translation\FileLoader`

2. `Illuminate\Translation\FileLoader`

The `FileLoader` is the class that is responsible for loading the translation files from the file system. The language files are loaded from the `/resources/lang/` directory by default. The `lang/` directory contains the `en` (English from the ISO 639-1[1] standard) sub-directory by default. Each sub-directory within the `lang/` directory corresponds to a particular locale.

Each locale directory may contain various PHP files returning arrays. These arrays contain a key/value pair, indicating the key of the translation and the value of the language translation. For example, the following is the contents of the `/resources/lang/en/passwords.php` file:

```php
<?php

return [

    /*
    |--------------------------------------------------------------------------
    | Password Reminder Language Lines
    |--------------------------------------------------------------------------
    |
    | The following language lines are the default lines which match reasons
    | that are given by the password broker for a password update attempt
    | has failed, such as for an invalid token or invalid new password.
    |
    */

    'password' => 'Passwords must be at least six characters and match the
                   confirmation.',
    'user' => "We can't find a user with that e-mail address.",
    'token' => 'This password reset token is invalid.',
    'sent' => 'We have e-mailed your password reset link!',
    'reset' => 'Your password has been reset!',

];
```

The individual PHP files containing the translation lines are referred to as "groups". So for a fresh Laravel installation, the following groups exist:

[1] http://www.loc.gov/standards/iso639-2/php/English_list.php

Default Translation Groups

Group Name	Path
Pagination	`resources/lang/en/pagination.php`
Passwords	`resources/lang/en/passwords.php`
Validation	`resources/lang/en/validation.php`

2.1 `FileLoader` Public API

The `FileLoader` implements the `Illuminate\Translation\LoaderInterface` contract. The `LoaderInterface` requires two public methods `load` and `addNamespace`. `FileLoader` does not add any extra public methods to it's implementation.

2.1.1 `load($locale, $group, $namespace = null)`

The `load` method is used to load the messages for the provided `$locale`. The `load` method also requires a `$group` to be specified. The `$group` corresponds to the actual file of key/value pairs that should be loaded (such as `pagination.php` or `passwords.php`).

2.1.2 `addNamespace($namespace, $hint)`

The `addNamespace` method is responsible for adding a given `$namespace` to the loader. Each namespace requires a `$hint`. The `$namespace` parameter is used to give a name to the namespace, which typically would correspond to a package name, or some other logical organization. The `$hint` tells the framework *where* the locale and group files are located for the given namespace. For example, if a directory '/storage/lang_custom/' existed for storing custom language files, the directory path would become the `$hint`.

2.2 Translation Namespaces

To prevent conflicts between multiple vendors using the same group names for a particular locale, the translator component allows the usage of "namespaces". Namespaces are a way to refer to a specific collection of locales and groups, where collection here refers to a different location on the file system. There is no limit on the number of namespaces that can be created. The translator component always has a * namespace available.

The * namespace refers to any groups that exist within the default `/resources/lang` directory. Namespaces are separated from the group and language line with the `::` character sequence. Because of this, both function calls in the following example will return the same result:

```
// The following function calls are assuming the 'en' locale, using
// the default Laravel translation groups.

// Retrieving a language value without a namespace.
$passwordReset = trans('passwords.reset');

// Retrieving a language value with the default namespace.
$passwordResetTwo = trans('*::passwords.reset');
```

After the above code is executed, both `$passwordReset` and `$passwordResetTwo` would contain the same `Your password has been reset!` value.

Adding custom namespaces is simple. Assuming the directory `/storage/lang_custom/` exists, it could be added to the file loader like so:

```
$loader = app('translation.loader');
$loader->addNamespace('custom', storage_path('lang_custom'));
```

The `Illuminate\Translation\LoaderInterface` can be resolved from the application container using the `translation.loader` name.

Assuming there was a locale directory en with the following `passwords.php` group file:

```
<?php

return [

    'reset' => 'This is a custom reset message.',

];
```

The value for `reset` can be retrieved from the custom namespace like so:

```
$defaultReset = trans('password.reset');
$customReset  = trans('custom::password.reset');
```

After the above code is executed, the `$defaultReset` variable would have the value `Your password has been reset!` and the value of `$customReset` would be `This is a custom reset message.`.

3. Hashing: One Way Encryption

Hashing data is a common task in software development. Hashing is similar to encryption, with the difference that hashing is a one-way process. The goal is that the original message cannot be retrieved from the resulting hash. Hashing is accomplished through the use of a hash function. A hash function generally accepts some input, called a message and transforms the message to produce a given output, a digest. Laravel, as it does with many other things, defines a `Illuminate\Contracts\Hashing\Hasher` interface, which can be implemented to create new hashing providers that Laravel can use.

Any type that implements the `Hasher` interface must be able to generate a hash (by implementing the `make($value, array $options = [])` method), check that a value matches a hashed value (by implemented the `check($value, $hashedValue, array $options = [])` method) and a hasher must be able to determine if a given hashed value needs to be rehashed (by implementing the `needsRehash($hashedValue, array $options = [])` method).

Hashing Confusion

A lot of newcomers to Laravel, and programming in general, seem to be confused about the differences between hashing and encryption. Hashing is one way; the point is that the final result is not recoverable. Encryption, conversely, is two-way; and the final result is reversible given enough information. Another point of confusion is where Laravel stores the salts for password hashes in the database. In the case of newer PHP password hashing API's, the salt, along with any other information required to compute the hash, is stored with the hash in the database.

Laravel internally utilizes a `Hasher` implementation in a few scenarios, mostly having to deal with users and their passwords. At the time of writing, Laravel uses hashing in the following classes/services:

Class	Purpose
`Illumiante\Auth\EloquentUserProvider`	Validates a user against credentials that are passed into the `validateCredentials` method.
`Illumiante\Auth\DatabaseUserProvider`	Validates a user against credentials that are passed into the `validateCredentials` method.

Laravel provides one `Hasher` implementation right out of the box: the `Illuminate\Hashing\Bcrypt` hasher, which provides bcrypt[1] password hashing, a hashing function based on the Blowfish[2] cipher. The `Bcrypt` hasher implements all methods defined in the `Hasher` interface, and also provides one extra: `setRounds($rounds)`, which is used to set work factor for the bcrypt

[1]https://en.wikipedia.org/wiki/Bcrypt

[2]https://en.wikipedia.org/wiki/Blowfish_(cipher)

hashing function.

3.1 Laravel's `Bcrypt` Hasher

The `Illuminate\Hashing\Bcrypt` hasher is registered with the service container in the `Illuminate\Hashing\HashServiceProvider` service provider, and is bound to the name `hash` (the `hash` key in the service container is itself an alias for the `Illuminate\Contracts\Hashing\Hasher` interface). The following code sample would return an instance of the `Bcrypt` hasher by default:

```php
// Get a new Bcrypt instance.
$hasher = app('hash');
```

The `Bcrypt` hasher is also used by the `bcrypt` application helper function. It actually uses the above method to resolve the `Bcrypt` hasher from the service container. This means that any other service provider that modifies the `hash` service container entry will also affect the `bcrypt` helper function. The `bcrypt` helper function is also used internally in the `Illuminate\Foundation\Auth\ResetPasswords` trait; specifically, it is used in the implementation of the `resetPassword($user, $password)` method.

The following sections will explain the `Bcrypt` hasher's various methods.

3.1.1 `setRounds($rounds)`

The `setRounds` method is simple way to control how many rounds, or iterations the `make` method will use when calculating the final hash. An argument passed to `$rounds` must be a positive integer between `4` and `31` (including both `4` and `31`). If an integer outside the stated range is supplied an instance of `ErrorException` will be thrown. The internal default value for `$rounds` is `10`.

3.1.2 `make($value, array $options = [])`

The `make` method is used to hash a given `$value` with the provided `$options`. At the time of writing, the `rounds` value is the only option that is checked for in the `$options` array. See the information on the `setRounds` method for more information on the `rounds` option. The internal default value for `rounds` is `10`.

If a user supplies a value for `rounds` using the `$options` array, that value will be used when calculating the hash. In all other scenarios, the internal value for `rounds` will be used (this value can be modified using the `setRounds($rounds)` method).

The `make` method internally makes a call to PHP's `password_hash`[3] function with `PASSWORD_-BCRYPT` as the argument for the `$algo` parameter.

The following code example will get an implementation of `Illuminate\Contracts\Hashing\Hasher` from the service container and generate a few hashes. The output will appear below the code example. The output will be different each time the code is ran:

[3]http://php.net/manual/en/function.password-hash.php

```php
/**
 * Get a new instance of Hahser from the service container.
 *
 * @var \Illuminate\Contracts\Hashing\Hasher $hasher
 */
$hasher = app('Illuminate\Contracts\Hashing\Hasher');

// Generate hashes for integers 0 - 9
// using the default work factor.
for ($i = 0; $i < 10; $i++) {
    echo $hasher->make($i),PHP_EOL;
}
```

The above would produce results similar to the following output:

```
$2y$10$jVlNmKI5Gg0j.3nER7tevuUaesWYIuwoxzghpIKfb2LvNMoTGaac6
$2y$10$DZzbYPf88wWo6nFW4LOAje4oJsWxZK.vg.k6vqxouXmb/6lsx045y
$2y$10$7CKZ0rmSOVxjvc2XG3diLOpalxjMkjcdw1.zKeqZPiSa7dD8K7GdS
$2y$10$qpiLO4hCLgYWmK4WJp0tTuv7klAM6RO5QrAnEqn1ULN92S.V5U4B6
$2y$10$no8cpUgVPiLw9rYOBknKWeiO2fC45.dzzhxE.rW8qn9ZojixXvoq.
$2y$10$31BmbMpivMGz1bsf5PAwTeEFhnS0GMS4GOkjNp.TWh9PZsZ0jLVWC
$2y$10$76T.L071J.8ewnnI3oocoukJtC8QljZiIesebdgewyMKygsl20QU2
$2y$10$nCE.5KTwPR8g6mrm4M9jlecoBpITCavThwRT0IZVQuMRg8qHQXaea
$2y$10$8uQ/0uc6wB1OQ7220wA2Ze54WqMzKTxWPQyi./bODTiBL/I7QgwgW
$2y$10$2ywQnQZKXAHMRqmj4iCr5O9Lr67Gv9u1BjXEiVspSRO.gWV1ROF7a
```

3.1.3 `check($value, $hashedValue, array $options = [])`

The `check` method will validate a given `$value` against a previously generated `$hashedValue`. The `$options` parameter is not utilized in Laravel's `Bcrypt` implementation. The `$value` must be the plain text value to check, and the `$hashedValue` should be the previously generated hash. The `check` method internally makes a call to PHP's `password_verify`[4] function.

The following code example will generate a hash for the string `Ave Maria` and perform some check if arbitrary strings are valid against the hashed value.

[4]http://php.net/manual/en/function.password-verify.php

```php
/**
 * Get a new instance of Hahser from the service container.
 *
 * @var \Illuminate\Contracts\Hashing\Hasher $hasher
 */
$hasher = app('Illuminate\Contracts\Hashing\Hasher');

// Generate a hash for "Ave Maria"
$hash = $hasher->make('Ave Maria');

// true
$isValid = $hasher->check('Ave Maria', $hash);

// false
$isValid = $hasher->check('Ave maria', $hash);

// false
$isValid = $hasher->check('password', $hash);
```

The following example will generate twenty hashes for the string `Ave Maria` and check the original string against all hashes. This demonstrates that even though the hashes are different each time a string is hashed, enough information is stored with the hash that a password can be checked successfully;

```php
/**
 * Get a new instance of Hahser from the service container.
 *
 * @var \Illuminate\Contracts\Hashing\Hasher $hasher
 */
$hasher = app('Illuminate\Contracts\Hashing\Hasher');

// Generate 20 hashes for "Ave Maria"
for ($i = 0; $i < 20; $i++) {
    $validHashes[] = $hasher->make('Ave Maria');
}

// Check "Ave Maria" against all 20 differnet hashes.
// They should all be valid.
foreach ($validHashes as $hash) {
    $isValid = $hasher->check('Ave Maria', $hash);

    // Improve the readability of the output.
    $isValid = ($isValid) ? 'true' : 'false';

    echo $hash, ' valid: ', $isValid, PHP_EOL;
}
```

The above code would generate something similar to the following output (the exact hashes will change each time it is ran):

```
$2y$10$0wgNNEz6N6avHmomWEPQI.GVZS32c8Kk.E1jAdUp0n4SuILQNohQC valid: true
$2y$10$g81.C5MDdFfYXLMM6iRPJOW8frm6N6jAHYBzQT9TddbYFX2pDgt2W valid: true
$2y$10$lWIUivhbnr8W3ILhquCqVuVl2E2esGNFSd4pueuVp/I7BbCNPI48. valid: true
$2y$10$WIHxrlsgCpIqUZKIzi5AEe4cMWBCvFliwBas7YGMuJS4yDxI4f/Je valid: true
$2y$10$XBq/FNfv87T0SXGBUd0UvO/azpuECPDYM7cLSKhszWBwu3I2m5aO2 valid: true
$2y$10$El8v3pu2Ko4IQ82g4Xacd.NpJw3Pj4VLLoIJMIVmPDQqur9tSzG/i valid: true
$2y$10$pW85CptXh9cC6XQ6HqKwTeBOrBWBMsogFRpNTHxvHc4U8bBUS/Wte valid: true
$2y$10$5j6hK9uYpc3Vbp4ATsJh5O8MxTIzB8JMmgxLlULA6HjOd4mFnW13u valid: true
$2y$10$W3rNwIfEfBMaiub1kxhW5u2b3lrwpLNzPajOc2CYy224s4WUEdpGW valid: true
$2y$10$h6gEqC8IfQRmBflGD93BD.hNwNBYxQaMPqkKLqAySAfQziL2deYvS valid: true
$2y$10$8QQ7RpXOfCIl0dZi0XcL1uuAFui6xCqbWLyIZxRWmwwYizK4ckOO2 valid: true
$2y$10$RHhyoygqBS3Cb.kwXEihW.jLh.wOkIO6wjSZIqcK3HeStR.Jy/IFC valid: true
$2y$10$jHd4HrR.2f.Vf7XIsVdPVuoIxUNMWpErwRZtiyffenG7adJd7AOJS valid: true
$2y$10$CwK0aN9BtVYfb2yIIfbpduPJ1CFtvA028jK1EDZmH1PdpGwLO.b06 valid: true
$2y$10$lsfaDaZqrt.n.UByvC90e.JXCKUa48uiUekue5zKDIJEFKEblkNQK valid: true
$2y$10$.QoDYocdqXltvr4Q4E67V.tFpsXtuq3IshwCpMlanFES/sLYIUx4q valid: true
$2y$10$MlfY7z9aKSudnkd00r6h5O0ZSt6B7qBel8jXEYGEaL7.zugM.lfbq valid: true
$2y$10$4jvJYkkpTjlj1nwW8epIb.QLYK0nrDEnKmj/BfcGyTlIjaQO/2Dh6 valid: true
$2y$10$XFEfJeDM.rTc3Kj4cMEYtegiRejRynLbn7aubaPzd0G3iHbf5VgVS valid: true
$2y$10$.NDd6lPPSaJ0uRttNhB8K.kmk/i.1ntNI4/Apj.RtXkyDHncwAdNm valid: true
```

3.1.4 `needsRehash($hashedValue, array $options = [])`

The `needsRehash` method is used to determine if the provided `$hashedValue` has been hashed with the supplied `$options`. This is useful for when the work factor of a hashing function has been updated and all passwords need to be updated to use the new hashing options. The following code example will demonstrate the usage of the `needsRehash` method. It also shows that two hashes generated with different `$options` will still be valid, but only one of them does not need to be rehashed.

```php
/**
* Get a new instance of Hahser from the service container.
*
* @var \Illuminate\Contracts\Hashing\Hasher $hasher
*/
$hasher = app('Illuminate\Contracts\Hashing\Hasher');

// Generate a hash with fewer rounds than the default.
$lessRounds = $hasher->make('Ave Maria', ['rounds' => 4]);

// Generate a hash with the default number of rounds.
$defaultRounds = $hasher->make('Ave Maria');
```

```

// Generate a hash with more rounds than the default.
$moreRounds = $hasher->make('Ave Maria', ['rounds' => 11]);

// All of these are valid.
$isValid = $hasher->check('Ave Maria', $lessRounds);
$isValid = $hasher->check('Ave Maria', $defaultRounds);
$isValid = $hasher->check('Ave Maria', $moreRounds);

// This hash needs to be rehashed because it was generated with
// fewer rounds than the current number of rounds (default).
$needsRehash = $hasher->needsRehash($lessRounds);

// This hash does not need to be rehashed because it was generate
// with the same number of rounds as the current number of rounds.
$needsRehash = $hasher->needsRehash($defaultRounds);

// The $moreRounds hash also needs to be rehashed because it has more rounds
// than the current number of rounds (the default number), even though it
// has a higher work factor.
$needsRehash = $hasher->needsRehash($moreRounds);
```

3.2 The Hash Facade

Laravel provides many facades to make it easier to quickly build application and test ideas. One such facade is the `Illuminate\Support\Facades\Hash` facade, which provides access to whatever implementation is bound to the `Illuminate\Contracts\Hashing\Hasher` interface within the service container. Since the `Hash` facade resolves an implementation of `Hasher`, all `Hasher` methods can be used on the facade:

```
use Illuminate\Support\Facades\Hash;

// All of the following calls are functionally
// equivalent. The output will differ only
// because bcrypt hashes will differ.
$hash = Hash::make('test');
$hash = app('hash')->make('test');
$hash = app('Illuminate\Contracts\Hashing\Hasher')->make('test');
```

3.3 Available Hashing Functions

Even though Laravel only provides a `Illuminate\Contracts\Hashing\Hasher` implementation for the bcrypt function, there are many more hashing functions that are available to developers. The following functions are available through the use of PHP's `crypt`[5] function:

[5]http://php.net/manual/en/function.crypt.php

Function	Notes	Secure
`CRYPT_STD_DES`	Based on Standard DES, requires a two character salt from the alphabet `./0-9A-Za-z`. See alphabet table below for more details.	Insecure*
`CRYPT_EXT_DES`	Based on the Extended DES, allows for variable iterations (or rounds) ranging from `1` to `16 777 215`, inclusive. Rounds are expressed as an integer in base64. Also allows for a salt four bytes long, following the iteration count. Salts must be derived from the alphabet `./0-9A-Za-z`. See alphabet table below for more details.	Insecure*
`CRYPT_MD5`	MD5 hashing. Allows for a twelve character salt, starting with the string `$1$`. Because of the required characters in the salt, we are left with only eight usable characters for our salt.	Insecure*
`CRYPT_BLOWFISH`	Blowfish hashing. Salt begins with `$2y$`, a cost parameter, and another `$`. Salts end with twenty-two characters from the alphabet `./0-9A-Za-z`. See table below for more details. Default hashing function used by Laravel.	Secure*
`CRYPT_SHA256`	A SHA-256 hash. Salts are sixteen characters and prefixed by the string `$5$`. Salts can be begin with `rounds=<N>$`, where `<N>` is the value is the number of rounds (with the rounds ranging from `1`, to `999 999 999`, inclusive). The default number of rounds is `5 000`.	Secure**
`CRYPT_SHA512`	A SHA-512 hash. Salts are sixteen characters and prefixed by the string `$6$`. Salts can be begin with `rounds=<N>$`, where `<N>` is the value is the number of rounds (with the rounds ranging from `1`, to `999 999 999`, inclusive[6]). The default number of rounds is `5 000`.	Secure**

* The insecure/secure designation is determined based on whether or not the function is a general purpose function, and whether or not is an option for PHP's `password_hash`[7] function.

** The SHA-2 family of hashing functions have had no publicly disclosed successful attacks against all rounds of the function. NIST is currently working on the SHA-3 family[8] of hashing functions.

Appendix A: Available Hash Functions contains a listing of all the hash functions that are available for use with PHP's `hash` function.

The following table contains the characters from the alphabet `./0-9A-Za-z`, which is used by many of the hashing functions in PHP. It has been arranged so that the characters appear in the order if you were to create a function to convert an integer to base64. It is also important to note that the table includes spaces to improve readability and are **not** part of the alphabet.

[6]Regarding `CRYPT_SHA256` and `CRYPT_SHA512`, a value supplied for the number of rounds that falls out of the range `1` to `999 999 999`, inclusive, will cause the supplied value to be truncated to the nearest limit.

[7]http://php.net/manual/en/function.password-hash.php

[8]http://csrc.nist.gov/groups/ST/hash/index.html

`./0-9A-Za-z` Alphabet Characters

Symbols	`. /`
Numeric	`0 1 2 3 4 5 6 7 8 9`
Lowercase Alpha	`a b c d e f g h i j k l m n o p q r s t u v w x y z`
Uppercase Alpha	`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

The following sections will create implementations of `Illuminate\Contracts\Hashing\Hasher` for the `CRYPT_STD_DES`, `CRYPT_EXT_DES`, `CRYPT_MD5`, `CRYPT_SHA256` and `CRYPT_SHA512` functions. For an implementation of the `CRYPT_BLOWFISH` function, see Laravel's default `Illuminate\Hashing\BcryptHasher` implementation. Afterwards, we will create a new `Hasher` class that will allow us to interact with all the various hashing functions and a new service provider class to register everything with the service container.

The following implementations will be created in some directory that can be accessed with the namespace `Laravel\Artisan\Hashing`. This namespace and directory is completely arbitrary and can be changed to match the structure of any project so long as the namespace references are updated accordingly.

Artisan Console Overview

"I see no reason to believe that a creator of protoplasm or primeval matter, if such there be, has any reason to be interested in our insignificant race in a tiny corner of the universe, and still less in us, as still more insignificant individuals. Again, I see no reason why the belief that we are insignificant or fortuitous should lessen our faith."

– Rosalind Franklin (1920 - 1958), Physical Chemist, X-Ray Crystallographer

The Artisan Command Line Environment (CLI) is a terminal based application that can be used to ease development with Laravel. Artisan ships with many commands by default as well as exposes APIs to create your own custom commands.

Many of the commands available are commands to either generate database tables, manage existing tables or clear various application caches. The commands are organized by command namespaces; for example, the `clear` command under the `view` namespace would clear the previously cached views and is written as `view:clear`.

4. Database Management Commands

This section will provide a reference for the commands that are used to create and manage database tables. Each command will be discussed in greater detail in later sections.

Command	Signature	Description
`migrate`	`--database --force --path --pretend --seed --step`	Runs the database migrations.
`migrate:install`	`--database`	Creates the migrations table.
`migrate:refresh`	`--database --force --path --seed --seeder`	Resets and run all migrations again.
`migrate:reset`	`--database --force --pretend`	Rollback all database migrations.
`migrate:rollback`	`--database --force --pretend`	Rollback the last database migration.
`migrate:status`	`--database --path`	Show the status of each migration.
`cache:table`		Creates a migration for the cache table.
`queue:table`		Creates a migration for the queue jobs table.
`session:table`		Creates a migration for the sessions table.

5. Cache Management Commands

This section will provide a reference for the commands that are used to generate or clear various caches used by your application. Each command will be discussed in greater detail in later sections.

Command	Description
`cache:clear`	Flushes the application cache.
`config:cache`	Creates a cache file to improve configuration performance.
`config:clear`	Removes the configuration cache file.
`route:cache`	Create a cache file to improve router performance.
`route:clear`	Remove the route cache file.
`view:clear`	Clear all compiled view files.

6. Default Artisan Commands

The following sections will discuss each of the default Artisan commands.

6.1 The `clear-compiled` Command.

The `clear-compiled` command is used to clear the compiled classes and services application cache. These two files are located in the `bootstrap/cache/` directory. The compiled classes cache is stored in a file named `compiled.php` and the services cache is stored in a file named `services.php`. This command will remove both of these files if they exist.

The following demonstrates how to use the `clear-compiled` command:

```
php artisan clear-compiled
```

The `clear-compiled` command is defined in the `src/Illuminate/Foundation/Console/-ClearCompiledCommand.php` file.

6.2 The `env` Command

The `env` command is used to simply return the name of the current framework environment. The environment name that is returned is set in the `.env` file (specifically whatever value is set for the `APP_ENV` entry).

```
# Display the current framework environment.
php artisan env
```

The `env` command is defined in the `src/Illuminate/Foundation/Console/Environment-Command.php` file.

6.3 The `migrate` Command

The `migrate` command is used to run migrations against the database. It provides many different options and flags (six in total). The following table lists and explains each of these options and flags.

Option/Flag	Description	Default Value
`--database=CONNECTION_NAME`	The name of the database connection to use. These can be found in the `databases.connections` configuration entry.	Assumes the value of the of the `DB_CONNECTION` environment variable, or `mysql` if no value has been set.
`--force`	Forces the migrations to run in a production environment.	Default behavior is to not run migrations in a production environment.
`--path=PATH`	The path to the migrations file.	The default migrations directory is `database/migrations/`.
`--pretend`	Does not run any migrations and instead displays the SQL queries that would be run.	The default behavior is to run the migrations.
`--seed`	Specifying this flag will cause the `db:seed` command to be executed after the migrations have run.	The default behavior is to not run the `db:seed` command afterwards.
`--step`	Specifying this flag will cause the migration's batch number to be incremented for *each* migration so that each migration can be rolled back separately from the others.	The default behavior is to give all newly ran migrations the same batch number and *not* increment the batch for each migration.

The following examples will demonstrate the different ways to call the `migrate` command. The examples are using the default `create_users_table`, `create_password_resets_table` and `create_cache_table` migrations.

```
# Run all migrations.
php artisan migrate
```

After the above command has executed the migrations will be ran and the migrations table will look similar to the following:

migration	batch
2014_10_12_000000_create_users_table	1
2014_10_12_100000_create_password_resets_table	1
2016_04_10_041139_create_cache_table	1

In contrast, if we had set the `--step` flag, each migration would have a different batch number:

```
php artisan migrate --step
```

would result in a table that contains value similar to the following (note the differences in the `batch` values):

migration	batch
2014_10_12_000000_create_users_table	1
2014_10_12_100000_create_password_resets_table	2
2016_04_10_041139_create_cache_table	3

The following example demonstrates how to use the --pretend flag:

```
php artisan migrate --pretend
```

Output similar to the following wold be displayed in the terminal (line breaks have been added to improve readability):

```
CreateUsersTable:
create table `users` (
    `id` int unsigned not null auto_increment primary key,
    `name` varchar(255) not null,
    `email` varchar(255) not null,
    `password` varchar(255) not null,
    `remember_token` varchar(100) null,
    `created_at` timestamp null,
    `updated_at` timestamp null
)
default character set utf8 collate utf8_unicode_ci

CreateUsersTable:
alter table `users` add unique `users_email_unique`(`email`)

CreatePasswordResetsTable:
create table `password_resets` (
    `email` varchar(255) not null,
    `token` varchar(255) not null,
    `created_at` timestamp not null
)
default character set utf8 collate utf8_unicode_ci

CreatePasswordResetsTable:
alter table `password_resets` add index `password_resets_email_index`(`email`)

CreatePasswordResetsTable:
alter table `password_resets` add index `password_resets_token_index`(`token`)

CreateCacheTable:
create table `cache` (
    `key` varchar(255) not null,
    `value` text not null,
    `expiration` int not null
)
default character set utf8 collate utf8_unicode_ci

CreateCacheTable:
alter table `cache` add unique `cache_key_unique`(`key`)
```

The following example sets the path to the database migrations. It uses the default migrations path, just as an example.

```
# Set the path to the database migrations.
php artisan migrate --path=database/migrations
```

The following example sets the database connection name. It uses the default connection name as an example.

```
# Specify the database connection to use.
php artisan migrate --database=mysql
```

A common cause of confusion is the name of the `--database` option. The value supplied for this option is really the name of the database *connection*, not the name of the actual database.

The following example would force the migrations to be ran in a production environment:

```
# Force the migrations to run in production.
php artisan migrate --force
```

The following example would run the `db:seed` command after the migrations have been created:

```
# Run the db:seed command after the migrations.
php artisan migrate --seed
```

The `migrate` command is defined in the `src/Illuminate/Database/Console/Migrations/MigrateCommand.php` file.

6.4 The `optimize` Command

The `optimize` command optimizes various aspects to improve the performance of the Laravel application. The command provides two flags. The `--force` flag can be set to indicate that the compiled class file should be written (by default the compiled class is not written when the application is in debug mode). The `--psr` flag can be set to indicate that Composer should not create an optimized class map loader (class maps are generally better for performance reasons).

The compiled files cache will be written to the `bootstrap/cache/compiled.php` cache file. The files that are compiled and written to the cache any files that meet the following criteria:

- Any files specified in the `compile.files` configuration entry;
- Any files that are specified by any service providers listed in the `compile.providers` configuration entry;
- Any framework files listed in the `src/Illuminate/Foundation/Console/Optimize/-config.php` file.

The following examples demonstrate how to use the `optimize` command:

```
# Generate a compiled class file with default options.
php artisan optimize

# Generate a compiled class file without optimizing the Composer
# autoload file.
php artisan optimize --psr

# Generate a compiled class file on a development or debug machine.
php artisan optimize --force
```

When specifying the `--psr` flag, the `optimize` command will call the `composer dump-autoload` command. Without the `--psr` flag, the Composer command `composer dump-autoload -o` will be called instead.

The `optimize` command is defined in the `src/Illuminate/Foundation/Console/OptimizeCommand.php` file.

6.5 The `serve` Command

The `serve` command is used to run the application using the PHP development server. This command is generally used for development and testing purposes. The `serve` command accepts two optional options: `host` can be used to change the address of the application (by default this is `localhost`) and `port`. The `port` option can be used to change the port the application responds to (by default this is `8000`).

When using HHVM for development and testing purposes, HHVM must be at least version `3.8.0` to use the internal development server.

The following examples demonstrate how to use the `serve` command:

```
# Start the development server with default options.
php artisan serve

# Change the development server address.
php artisan serve --host=homestead.app

# Change the development server port.
php artisan serve --port=8080

# Change the development server address and port.
php artisan serve --host=laravel.app --port=8080
```

The `serve` command is defined in the `src/Illuminate/Foundation/Console/ServeCommand.php` file.

Writing Artisan Console Commands

”... it is shameful that there are so few women in science... In China there are many, many women in physics. There is a misconception in America that women scientists are all dowdy spinsters. This is the fault of men. In Chinese society, a woman is valued for what she is, and men encourage her to accomplishments yet she remains eternally feminine.”

– Chien-Shiung Wu (1912 - 1997), Physicist

It is often very useful to create custom Artisan commands specifically for your application or package. Custom commands are, by default, stored in the `app/Console/Commands` directory (commands can be stored at any path that can be autoloaded based on the applications `composer.json` settings).

The `app/Console/Commands` directory contains an `Inspire` command by default (stored in the `app/Console/Commands/Inspire.php` file). This command is extremely simple and just displays one of the following messages at random:

```
When there is no desire, all things are at peace. - Laozi
Simplicity is the ultimate sophistication. - Leonardo da Vinci
Simplicity is the essence of happiness. - Cedric Bledsoe
Smile, breathe, and go slowly. - Thich Nhat Hanh
Simplicity is an acquired taste. - Katharine Gerould
Well begun is half done. - Aristotle
He who is contented is rich. - Laozi
Very little is needed to make a happy life. - Marcus Antoninus
```

These quotes are stored in the `vendor/laravel/framework/src/Illuminate/Foundation/Inspiring` class. The `Inspire` example commands ships as part of the Laravel framework and it’s definition can be found in the `routes/console.php` file.

The `Inspire` command is very short and readable (as it is defined in the `routes/console.php` file):

```
Artisan::command('inspire', function () {
    $this->comment(Inspiring::quote());
});
```

Re-implementing this command as a class, as opposed to using the `console.php` file approach, might look something like this (stored within the `app/Console/Commands/` directory):

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;
use Illuminate\Foundation\Inspiring;

class Inspire extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'inspire';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Display an inspiring quote';

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $this->comment(PHP_EOL.Inspiring::quote().PHP_EOL);
    }
}
```

It is this class-based implementation of the `Inspiring` command that will be used throughout the remainder of this section.

Commands, at their most basic, contain a `$signature`, `$description` and a `handle()` method. A commands signature is similar to a method or function signature in that it defines the name, parameters and options of the command. The description is simply a helpful description of what the command does. The `handle()` method implementation is the code that actually performs the action described in the commands description. The `Inspire` command can be executed like so:

```
php artisan inspire
```

After the command has executed it would display one of the random quotes listed above.

It is entirely possible that when attempting to execute the `inspire` command the following error will be displayed instead of a random quote:

```
[Symfony\Component\Console\Exception\CommandNotFoundException]
  Command "inspire" is not defined.
```

This error is caused when the application cannot locate a command. The most common cause of this is that the command has not been included in the applications console commands. The following section "The Application Console Kernel" will explain this in further detail.

7. The Console Kernel

Each application contains a console kernel. The console kernel is defined in the `app/Console/Kernel.php` (this will be referred to as the application console kernel) file. The kernel class that exists in that file extends Laravel's console kernel (which can be found in the `vendor/laravel/framework/src/Illuminate/Foundation/Console/Kernel.php` file; this kernel will be defined as the framework console kernel). From the applications point of view, the application console kernel is responsible for specifying which custom commands should be made available to users and when to automatically execute various commands and tasks (by using the task scheduler).

It is important to note that while this the application and console kernels are separated into their own sections, they are the same thing, and are only being differentiated in this book to make explaining them easier. The "application" console kernel, located in the `app/Console/Kernel.php` file, is simply a convenient way to expose the features of the "framework" console kernel to your application. Any features, limitations, or otherwise that are discussed in the section "The Framework Console Kernel" can be applied to the application console kernel class.

7.1 The Application Console Kernel

The application console kernel is mainly used for two purposes in relation to your application:

- Registering commands for use;
- Scheduling when various commands and tasks should be ran (by using the scheduler).

The kernel is fairly simple by default (the vast majority of the logic is contained within the framework's console kernel):

```php
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
    ];

    /**
     * Define the application's command schedule.
     *
     * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
     * @return void
     */
    protected function schedule(Schedule $schedule)
    {

    }

    /**
     * Register the Closure based commands for the application.
     *
     * @return void
     */
    protected function commands()
    {

    }

}
```

The kernel has an empty `$commands` array and an empty `schedule()` method. The `$commands` property is used to specify which commands are to be loaded when the console application (Artisan) starts. In the introduction to this chapter the following error message was discussed briefly:

```
[Symfony\Component\Console\Exception\CommandNotFoundException]
  Command "inspire" is not defined.
```

The most common cause of this error is the command that a user is attempting to run has not been added to the `$commands` array. The act of adding the name of a class to the `$commands` array is referred to as registering the command with the application.

To register a command with the application, simply add the fully qualified name to the command class to the `$commands` array. In earlier versions of Laravel (targeting older versions of PHP), this was done by specifying the class name as a string; however, starting with PHP version 5.5, the `::class` class constant. It is also important to remember to include the class if it is located in a different namespace (using the `use` PHP keyword). Application commands are, by default, stored within the `app/Console/Commands/` directory (under the `App\Console\Commands` namespace). Alternatively, commands can be registered using the command shorthand syntax.

The `commands` method is automatically called by the framework and is a convenient place to register commands using the shorthand syntax.

In the introduction to this chapter, the `inspire` Artisan command was used as an example. This command is stored within the `app/Console/Commands/Inspire.php` file and can be accessed by using the `App\Console\Commands\Inspire` class (this is the fully qualified name of the `Inspire` command). The following examples demonstrate how to register the `Inspire` command with the application.

The following example uses the `::class` class constant, and is generally the recommended way of referring to classes:

```php
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        // Remember that namespaces are relative so the
        // `Commands\Inspire` class is resolved in
        // relation to the current App\Console
        // namespace to get the final class.
        Commands\Inspire::class
    ];


    // Schedule method omitted.

}
```

The following example registers the `Inspire` command by supplying the fully qualified class name as a string:

```php
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
    /**
     * The Artisan commands provided by your application.
     *
     * @var array
     */
    protected $commands = [
        'App\Console\Commands\Inspire'
    ];


    // Schedule method omitted.

}
```

Scheduling commands and tasks will be discussed in its own section.

7.2 The Framework Console Kernel

The framework console kernel contains most of the pieces to make the Artisan console application work. For example, it is responsible for the registration, running and outputting the results of various console commands. The application console kernel inherits all its behavior from the framework's kernel.

7.2.1 Framework Kernel Public API

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

7.2.1.1 `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
<?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:

```
// 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

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:

```
// 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_migratio\
ns
```

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.

7.2.1.2 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
// 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.

7.2.1.3 `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'
]);
```

8. Writing Commands

The first step to creating custom commands is to generate the basic structure, or scaffolding, for the new command. This can be done by issuing the `make:console` Artisan command from the terminal. The `make:console` command accepts the name of the new command class, as well as accepts an optional `--command` option that can be used to specify the terminal command name (for example, `make:console` is the terminal command name of the `ConsoleMakeCommand` class).

By default, custom Artisan commands are stored in the `app/Console/Commands` directory, and are namespaced under the `App\Console\Commands` namespace. The following command would create a new class called `CreateFileCommand` and set the terminal command name to `create:file`. The command scaffolding would be generated and saved to the `app/Console/Commands/CreateFileCommand.php` file.

```
# Generate scaffolding for a new command.
php artisan make:console CreateFileCommand --command=create:file
```

The generated scaffolding would look similar to the following output:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class CreateFileCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'create:file';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Command description';

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

    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        //
    }
}
```

As can be seen in the above output, each command contains a `$signature` and `$description` property. The `$signature` property is used to specify what arguments and options the command can accept (this property did not exist in Laravel versions prior to version 5.1; options and arguments used to be defined using the `getOptions` and `getArguments` methods). The `$description` property can be used to provide a description for the newly generated command.

Just because the command has been generated, does not mean it is available for use within the Artisan console application. If the following was executed:

```
php artisan create:file
```

an error similar to the following would be generated and displayed as output:

```
There are no commands defined in the "create" namespace.
```

In order for the command to be available, it needs to be registered with the application's console kernel (located in the `app/Console/Kernel.php` file). The following can be added to the `$commands` property within the `Kernel.php` file to register the newly generated command (only the `$commands` property is being shown here):

```php
protected $commands = [
    // ...
    CreateFileCommand::class
    // ...
];
```

After saving the `Kernel.php` file, the `create:file` command should be available for use. If the command still cannot be found, regenerating Composer's class auto-loader file generally resolves the issue. To do this, simply issue the following command within the application's root directory:

```
# Regenerate the Composer auto-loader file.
composer dump-autoload -o
```

At this point, a simple `CreateFileCommand` command exists, but does not accomplish anything. When a command is executed from the terminal, or elsewhere, its `handle` method is ultimately called. It is within the `handle` method that the command's logic should be placed, even if it is to make calls to other libraries or services.

The following `handle` method implementation would simply create a new file with the current time inside a custom directory in the applications storage directory:

```php
public function handle()
{
    // Create a new file with the current time as the name.
    touch(storage_path('create_file/'.time()));
}
```

The above example references a directory that is not included in the application storage directory by default. The `create_file` directory must be created manually before attempting to execute the example command or else an error stating something similar to "Unable to create file because no such file or directory exists" will be issued.

Each time the `CreateFileCommand` command is called, a new file will be created inside the `app/storage/create_file` directory.

8.1 Command Input and Command Signatures

Most useful commands need to accept some sort of input from the user. The input can be as simple as requiring users to supply information in the form of parameters and options or can be more complicated by interactively entering data at the terminal. Command parameters are generally required data, but can have default values. Options are not necessarily required, and can be used as a sort of "flag" to change the behavior of the command's execution. Options must start with the `--` (double hyphen) prefix.

Parameters and options are defined in the command's `$signature` property. The `$signature` is a simply a string constructed from specialized syntax for defining parameters and options). The signature must start with the terminal name of the command (such as `make:model`). Following the terminal name, any number of parameters or options can be defined as long as they are enclosed within curly braces (`{` and `}`). When defining options within the signature, options must still start with the `--` option prefix (options can also have shortcuts, which start with the `-` prefix).

8.1.1 Input Parameters

For example, the following signature could be used to accept a user's first and last name:

```
protected $signature = 'example:command {firstName} {lastName}';
```

Since the `{firstName}` and `{lastName}` input requirements were defined as *parameters* and do not define a default value, the command requires that arguments be supplied for them. Running the following command (without arguments):

```
# Execute the example command without the required parameters.
php artisan example:command
```

would result in an error similar to the following output:

```
Not enough arguments (missing: "firstName, lastName").
```

The error message is indicating that the arguments for the `firstName` and `lastName` parameters have not been supplied. Arguments are given to command parameters in the order they appear in the command's signature. Supplying input to Artisan commands is as simple as supplying the value, separated by a space character. The following example would call the `example:command` command again with `John` being the value for `firstName` and `Doe` as the value for `lastName`:

```
# Execute the example command with the required parameters.
# 'John' would be the value of the firstName parameter.
# 'Doe'  would be the value of the lastName  parameter.
php artisan example:command John Doe
```

8.1.2 Input Parameter Default Values

Command arguments can also have default values. Default values can be specified by assigning the argument a value within the signature. The following would assign the default value `Doe` to the `lastName` parameter. If a user does not specify any value when calling the command, the default value is used:

```
// Assigning the lastName parameter a default value.
protected $signature = 'example:command {firstName} {lastName=Doe}';
```

Now, when a user is executing the command, an argument for the `lastName` parameter is not required. The following two command calls would are equivalent:

```
# Call the command, keeping the default value.
php artisan example:command John

# Call the command, specifying a value for lastName.
php artisan example:command John Doe
```

Any parameters that define default values are considered optional parameters. These parameters must be defined *after* any required parameters. A signature of `example:command {firstName} {lastName=Doe}` is valid; a signature of `example:command {lastName=Doe} {firstName}` is invalid and would raise an error stating something similar to "Cannot add a required argument after an optional one".

8.1.3 Adding Descriptions to Command Parameters

If we were to display the help information for the hypothetical `example:command` at this point by adding the `-h` flag to the command execution:

```
# Show the help information for the example:command
php artisan example:command -h
```

The following information would be included in the output (the "Options" section of the output has been omitted for this section):

```
Usage:
  example:command <firstName> [<lastName>]

Arguments:
  firstName
  lastName               [default: "Doe"]
```

The help information shows us each of the arguments, their order and any default values. As expected, a default value for `lastName` is shown (and is set to `Doe`). All of this information comes from the command's signature. As nice as the help information is, there are times when extra clarification for what an input argument is used for is not clear. To work around this issue, descriptions can be added to any input parameter or option by separating the argument or option declaration and description using a colon `:` character. The argument or option declaration should appear on the left side of the `:` character, and the description should appear on the right. The revised signature might look something like this:

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName=Doe : The last name of the user}';
```

The resulting information would not include the descriptions in its output:

```
Usage:
  example:command <firstName> [<lastName>]

Arguments:
  firstName              The first name of the user.
  lastName               The last name of the user [default: "Doe"]
```

8.1.4 Command Options

So far the discussion has focused on command parameters. Options are also a useful way for users to provide input and control the execution of commands. Defining commands is similar to defining command parameters with the exception that option names begin with the two hyphen (`--`) prefix. The following signature defines the same two `firstName` and `lastName` parameters with an optional `--age` option:

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName=Doe : The last name of the user}
        {--age=}';
```

Notice, in the above example, that the `--age` option has a trailing equals (=) sign. This indicates that the end user can supply a value to the option. The absence of the trailing equals sign indicates that the option behaves like a switch or flag; flags can only assume a `true` or `false` value (a flag evaluates to `true` if the user specifies the flag and `false` if they do not).

A command with the signature above could be executed like so:

```
# Call the command with all parameters, without options.
php artisan example:command John Doe

# Call the command with all parameters and options.
php artisan example:command John Doe --age=26

# Call the command with only the required parameters
# and the available options.
php artisan example:command John --age=26
```

8.1.5 Command Option Default Values

It is also possible to specify a default value for the options. The following signature is similar to the previous one, except that the `--age` option will have a default value of `0` that will be used if the end user does not specify an age:

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName=Doe : The last name of the user}
        {--age=0}';
```

The new `--age` option will also appear in the help information:

```
Usage:
  example:command [options] [--] <firstName> [<lastName>]

Arguments:
  firstName              The first name of the user.
  lastName               The last name of the user [default: "Doe"]

Options:
      --age[=AGE]        [default: "0"]
```

8.1.6 Adding Descriptions to Command Options

You can see from the above output that the command accepts an `--age` option and it has a default value of `0`. Although `--age` is fairly descriptive, it would be nice to give the option a good description. The process to add a description to options is the same as adding descriptions to parameters:

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName=Doe : The last name of the user}
        {--age=0 : The age of the user (optional);
            "0" indicates age not disclosed.}';
```

The help information would now display the description for the `--age` option:

```
...

Options:
      --age[=AGE]       The age of the user (optional); "0" indicates age not\
 disclosed. [default: "0"]

...
```

8.1.7 Command Option Shortcuts

In practice, options can become long to type for the end user; this is partly a side effect of having to type the `--` prefix along with the option name. To help alleviate this, options support shortcuts. In the previous example the `--age` option was used to demonstrate options. A shortcut could be created to allow the user to type `-a` instead of `--age` each time they wanted to use the `--age` option. Shortcuts are defined before the full option name, separated by a vertical bar (`|`) delimiter:

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName=Doe : The last name of the user}
        {--a|age=0}';
```

An options shortcut requires a slightly different prefix. Instead of using the double hyphen (--) prefix, shortcuts use a single hyphen (-) prefix. Executing a command with the above signature might look something like this:

```
# Use the full option name
php artisan example:command John Doe --age=26

# Use the option shortcut.
php artisan example:command John Doe -a=26
```

Any option shortcuts that have been defined will also appear in the commands help output. Shortcuts will appear before the option in the output. Shortcuts will appear before the full option name in the help output.

The example command that has been used so far would produce help information similar to the following:

```
...

Options:
  -a, --age[=AGE]        [default: "0"]

...
```

8.1.8 Using Command Options as Flags

So far the examples of command options make the assumption that the user is required to enter a value for a command option. Options can be used as flags or switches to change the execution behavior of the command. To have an option behave like a flag, simply omit the trailing equals (=) sign when defining the option in a command's signature. The following example signature adds an option to allow end users to indicate whether or not a particular user is an administrator:

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName=Doe : The last name of the user}
        {--age=0 : The age of the user (optional);
            "0" indicates age not disclosed.}
        {--admin : Indicates whether not a user is an administrator.}';
```

A command with the signature above could be executed like so:

```
# Indicate the user is an administrator.
php artisan example:command John Doe --admin

# The user is not an administrator (the flag is omitted).
php artisan example:command John Doe
```

8.1.9 Array Parameters

Users can supply a list of information to commands using array parameters. To create an array parameter add an asterisk (*) character after the name of the input parameter in the command's signature. Array parameters do not have default value.

The following example adds a `websites` parameter to the example command. Notice the `*` character after the parameter name.

```
protected $signature = 'example:command
        {firstName : The first name of the user.}
        {lastName : The last name of the user}
        {websites* : The websites the user contributes to.}
        {--age=0 : The age of the user (optional);
            "0" indicates age not disclosed.}
        {--admin : Indicates whether not a user is an administrator.}';
```

In the above example signature the default value for the `lastName` parameter has been removed. This is because adding a default value to the `lastName` parameter makes it an optional parameter. Arguments cannot appear after optional arguments (an error stating something similar to "Cannot add a required argument after an optional one" would be displayed). Additionally, we could not have simply changed the order of the arguments because array parameters must appear last (an error stating something similar to "Cannot add an argument after an array argument" would be displayed).

Displaying the help information for this command would display something similar to the following output. Pay attention to how the `websites` parameter is expressed in the usage section:

```
Usage:
  example:command [options] [--]
  <firstName> <lastName> <websites> (<websites>)...

Arguments:
  firstName              The first name of the user.
  lastName               The last name of the user
  websites               The websites the user contributes to.
```

An array parameter can be identified by the repeated parameter name in parenthesis. In the above example, it can be seen that the `websites` parameter is in fact an array because it is represented as `<websites> (<websites>)...` in the usage section.

The following example demonstrates how to execute a command with a signature similar to the previous example. The `firstName` is set to `John`; the last name is set to `Doe`; and the websites supplied are `laravel.com`, `laracon.us` and `laracon.eu`.

```
php artisan example:command John Doe laravel.com laracon.us laracon.eu
```

There is now way to define a default value for an array argument in the command's signature. However, an effect similar to default array argument values can be achieved in the implementation of the command itself.

8.1.10 Array Options

Lists of data can also be collected from the user using command options. The syntax is similar to defining command parameter arrays, the only difference is that the double hyphen (`--`) option prefix is required as well as the addition of an equals sign (`=`) when defining the array option. The following example rewrites the command array argument example as a command option instead:

```
protected $signature = 'example:command
    {firstName : The first name of the user.}
    {lastName : The last name of the user}
    {--age=0 : The age of the user (optional);
        "0" indicates age not disclosed.}
    {--websites=* : The websites the user contributes to.}
    {--admin : Indicates whether not a user is an administrator.}';
```

Unlike when defining command arguments, options can be defined in any order without worrying about errors related to optional and required arguments being displayed. However, command options are usually defined after all the arguments.

Notice that the `websites` option includes both the prefix and the `=*` suffix. The `=*` suffix is important and array options will not work without it. The following help information output shows how the command array option will be represented when viewing the help information. Array options can be identified easily since the name of the option will appear in brackets (this is `[=WEBSITES]` in the example output) as well as have the trailing text `(multiple values allowed)`.

```
Options:
    ...
    --websites[=WEBSITES]  The websites the user contributes to.
      (multiple values allowed)
    ...
```

A command with a signature similar to the above example could be executed like so:

```
php artisan example:command John Doe --websites=laravel.com
    --websites=laracon.us
```

When the command above is issued, the `firstName` value would be `John`, the `lastName` value would be `Doe` and the array of websites would contain `laravel.com` and `laracon.us`.

Similar to array arguments, array options cannot define a default value in the command's signature. A similar effect could be added later in the command's implementation.

8.1.11 Making Input Arguments Optional

So far we've looked at defining command arguments with either required or default values. Another useful thing is to make command arguments *optional*. By default, an argument with a default value is categorized as an optional argument. However, arguments can be created without a default value and still be optional. To create an optional command argument, simply add a question mark (?) symbol after the argument's name.

The following example signature would define an argument named `system` that is optional:

```
protected $signature = 'example:command
    {system? : The system architecture.}';
```

There are a few things to take note of when viewing the help information for a command with a signature similar to the previous example:

```
Usage:
  example:command [<system>]

Arguments:
  system                The system architecture.
```

In the "Usage" section, we can see that the `<system>` argument has been placed inside brackets. Any argument or option that appears in brackets in the usage section is an optional argument or option.

A command with a signature similar to the previous example could be executed like so:

```
# Execute the command without supplying an argument.
php artisan example:command

# Execute the command while supplying an argument.
php artisan example:command x64
```

Any type of command argument can be made optional, including argument arrays.

8.1.12 Command Input Syntax Reference

The following table serves as a quick reference for the common command signature definitions. The `Syntax` column demonstrates the syntax required, the `Description` column provides a description or quick explanation of the syntax and the `Required` column indicates if the end user will be required to provide a value for the resulting parameter or option when executing the resulting command.

Syntax	Description	Required
{test}	A required parameter `test` with no default value.	Yes
{test} : Description}	Adding a description to a parameter.	Yes
{test=Default}	Specifying a default value for a parameter.	No
{test?}	Making a parameter optional.	No
{test*}	Accept an array of input and store it in the value `test`.	Yes
{test?*}	Accept an optional array of input, and store it in the value `test`.	No
–test	Define a boolean flag named `--test`.	No
–test Description	Adding a description to an option.	No
–Ttest	Define a boolean flag named `--test` with a shortcut named `-T`.	No
–test=Default	Define an option named `--test` with the default value `Default`.	No
–test=*	Accept an optional array of input, and store in the value `--test`.	

8.1.13 Command Signature Alternatives

The command signature feature was introduced in Laravel version 5.1. In older versions of the framework the instance methods `getArguments` and `getOptions` were used to define the input expectations for commands. Both methods were expected to return arrays. These methods still exist and can still be used. This section will explain how these methods are used and the subtle changes that are required in order to use them.

When looking at a basic command class we see something similar to this (the methods have been omitted in the following example):

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class NewCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'new:command';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Command description';

    // Methods omitted...

}
```

The first thing to notice is that there are two protected variables. One is named `$signature` and the other is named `$description`; for this section, the `$description` variable is irrelevant. When the `$signature` variable is present in the command class, Laravel will expect there to be a valid command signature. It will then parse it and use that to determine the command's name, parameters and options.

The following code example is the start of a typical command class from older versions of the Laravel framework (again, instance methods have been omitted):

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class NewCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $name = 'new:command';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Command description';

    // Methods omitted...

}
```

At first glance there does not look to be any difference between the two code examples. However, in the second example there is a `$name` protected variable instead of a `$signature` protected variable. When this is the case, Laravel will use the `getArguments` and `getOptions` instance methods to determine the input expectations for the command.

If you would like to use the `getArguments` or `getOptions` methods for defining input expectations, or simply want to ease the transition from an older code base, the `$signature` property **cannot** be defined in your command class. The signature will be used if it is present, not necessarily if it has been supplied a value. This means that if both a `$name` and a `$signature` property have been defined, the signature method for defining input expectations will take precedence.

The `getArguments` and `getOptions` methods should return an array containing arrays that represent the command arguments (referred to as parameters in previous sections) and options. The format for defining arguments versus options is slightly different.

The following table shows the difference between the formats for options and arguments. Both of arguments and options are represented as an array:

Input Type	Format
Argument	`[$name, $mode, $description, $defaultValue]`
Option	`[$name, $shortcut, $mode, $description, $defaultValue]`

As you can see, the only difference between the two formats is that options can specify a shortcut.

When working with the `getOptions` and `getArguments` methods, there are two classes that are useful to work with. Both the options and arguments are part of the Symfony code base (Laravel's console application is an extension of Symfony's console application). The following two classes are required:

- `Symfony\Component\Console\Input\InputArgument` - This class is required when

defining input arguments.
- `Symfony\Component\Console\Input\InputOption` - This class is required when defining input options.

The `InputArgument` class defines a few constants that are required when adding command arguments to the `getArguments` method. These constants are used to set each of the arguments mode. The following table lists each of these constants, a description of them and their value (the value can be used in situations where you do not want to import the class, but this practice is generally not recommended).

Constant	Value	Description
`REQUIRED`	1	Indicates that the input argument is required.
`OPTIONAL`	2	Indicates that the input argument is optional.
`IS_ARRAY`	4	Indicates that the input argument is an array, and users can supply multiple values for the same argument.

Default values can only be defined for arguments that are using the `OPTIONAL` mode.

The `InputOption` class is similar to the `InputArgument` class in that it defines useful constants that are required when implementing the `getOptions` method. Like the arguments, the constants below set the mode of the option. The following table lists each of these constants:

Constant	Value	Description
`VALUE_NONE`	1	Indicates that the option should not accept any input value from the user. This will cause the option to behave like a switch or flag and adopt a boolean value.
`VALUE_REQUIRED`	2	Indicates that the option is required.
`VALUE_OPTIONAL`	4	Indicates that the option is optional.
`VALUE_IS_ARRAY`	8	Indicates that the option should accept multiple values.

The following sections will take a second look at all the previously explored examples using the `getOptions` and `getArguments` instance methods. Each section will contain code samples for both methods as well as the equivalent command signature.

8.1.13.1 Input Parameters

Input parameters require the `Symfony\Component\Console\Input\InputArgument` class to be imported in the current PHP file. Input parameters are defined in the `getArguments` instance method in the command class. The following signature will be implemented using the `InputArgument`:

```
test:command {firstArgument}
```

The corresponding `getArguments` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputArgument;

// Beginning of class omitted...

public function getArguments()
{
    return [
        ['firstArgument', InputArgument::REQUIRED]
    ];
}

// End of class omitted...
```

In the above example you will notice that we did not have to specify a description or default value when defining the input argument, but we did have to specify a value for the mode. All of the values for the input argument have a default value, except for the name. The default value for the mode is set to `InputArgument::OPTIONAL`, the description has an empty string as a default value and the arguments default value is set to `null` by default.

8.1.13.2 Input Parameter Default Values

The signature that will be implemented using the `getArguments` method is:

```
test:command {firstArgument=DefaultValue}
```

The corresponding `getArguments` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputArgument;

// Beginning of class omitted...

public function getArguments()
{
    return [
        ['firstArgument', InputArgument::OPTIONAL, '', 'DefaultValue']
    ];
}

// End of class omitted...
```

It is important to note that the mode changed from `InputArgument::REQUIRED` to `InputArgument::OPTION`. Only optional arguments can have a default value. Also, an empty string has been supplied for the argument's description field.

8.1.13.3 Adding Descriptions to Command Parameters

The signature that will be implemented using the `getArguments` method is:

```
test:command {firstArgument : This is the description }
```

The corresponding `getArguments` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputArgument;

// Beginning of class omitted...

public function getArguments()
{
    return [
        ['firstArgument', InputArgument::REQUIRED, 'This is the description']
    ];
}

// End of class omitted...
```

8.1.13.4 Command Options

Input options require the `Symfony\Component\Console\Input\InputOption` class to be imported in the current PHP file. Input options are defined in the `getOptions` instance method in the command class. The following signature will be implementing using the `InputOption`:

```
test:command {--optionName}
```

The corresponding `getOptions` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputOption;

// Beginning of class omitted...

public function getOptions()
{
    return [
        ['optionName']
    ];
}

// End of class omitted...
```

In the above example we only had to set the name of the option to `optionName`. When using the `getOptions` method we do not have to prefix option names with the double hyphen (`--`) prefix. Additionally, the shortcut, mode, description and default value for the option did not have to be set.

The following table details the default values that are assigned to each property of an `InputOption` if none are supplied:

Property	**Default Value**
Shortcut	`null`, meaning no shortcut will be assigned.
Mode	`InputOption::VALUE_NONE`. This means that options by default will behave like flags or switches and assume a boolean value.
Description	The description will be an empty string by default.
Default Value	The default value for options is a literal value of `null`.

8.1.13.5 Command Option Default Values

The signature that will be implemented using the `getOptions` method is:

```
test:command {--optionName=DefaultValue}
```

The corresponding `getOptions` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputOption;

// Beginning of class omitted...

public function getOptions()
{
    return [
        ['optionName', null, InputOption::VALUE_OPTIONAL, '', 'DefaultValue']
    ];
}

// End of class omitted...
```

8.1.13.6 Adding Descriptions to Command Options

The signature that will be implemented using the `getOptions` method is:

```
test:command {--optionName : Option description.}
```

The corresponding `getOptions` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputOption;

// Beginning of class omitted...

public function getOptions()
{
    return [
        ['optionName', null, InputOption::VALUE_NONE, 'Option description.']
    ];
}

// End of class omitted...
```

8.1.13.7 Command Option Shortcuts

The signature that will be implemented using the `getOptions` method is:

```
test:command {--q|optionName}
```

The corresponding `getOptions` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputOption;

// Beginning of class omitted...

public function getOptions()
{
    return [
        ['optionName', 'q']
    ];
}

// End of class omitted...
```

8.1.13.8 Array Parameters

The signature that will be implemented using the `getOptions` method is:

```
test:command {argumentName=*}
```

The corresponding `getArguments` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputOption;

// Beginning of class omitted...

public function getArguments()
{
    return [
        ['argumentName', InputArgument::IS_ARRAY | InputArgument::REQUIRED]
    ];
}

// End of class omitted...
```

8.1.13.9 Array Options

The signature that will be implemented using the `getOptions` method is:

```
test:command {--optionName=*}
```

The corresponding `getOptions` implementation would be:

```php
<?php

use Symfony\Component\Console\Input\InputOption;

// Beginning of class omitted...

public function getOptions()
{
    return [
        ['optionName', null, InputOption::VALUE_IS_ARRAY | InputOption::VALUE\
_OPTIONAL]
    ];
}

// End of class omitted...
```

In the above example two option mode values have been combined. If you were to specify just the `InputOption::VALUE_IS_ARRAY` option an error stating something similar to "Impossible to have an option mode `VALUE_IS_ARRAY` if the option does not accept a value".

8.2 Retrieving User Input

After a command's input expectations have been defined it useful to know how to retrieve that input from the user. There are multiple ways to receive input from users such as arguments, options and even interactively asking users for input.

The most common public methods for retrieving user input are the `argument` and `option` methods. The `argument` method is used to retrieve a single input argument or all input arguments from the user. Conversely, the `option` method is used to retrieve either a single input option or all input options that the user entered.

8.2.1 `argument($key = null)`

The `argument` method is used to retrieve either a single input argument (or parameter) from the user or all input arguments. To retrieve a single argument, supply a value for the `$key` method parameter. To retrieve all the input arguments, supply a value of `null` (the default value) for the `$key` method parameter. This method returns either a string or an array as its return value.

Assuming a command signature like so:

```
example:command {firstName} {lastName}
```

Executed like so:

```
php artisan example:command John Doe
```

We could retrieve user input arguments like so:

```php
<?php

// ...

function handle()
{
    // Get the firstName argument. In this
    // example, the $firstName value would
    // be set to 'John'.
    $firstName = $this->argument('firstName');

    // Get the lastName argument. In this
    // example, the $lastName value would
    // be set to 'Doe'.
    $lastName = $this->argument('lastName');

    // Get all the arguments from the user.
    // The $input variable would then
    // reference an array containing
    // all the input that the user
    // supplied when executing
    // the command.
    $input = $this->argument();
}

// ...
```

When a command similar to the above example has been executed, the `$input` variable would contain an array similar to the following output:

```
array [
  "command"   => "example:command"
  "firstName" => "John"
  "lastName"  => "Doe"
]
```

In the above output you can see that there are arguments returned by the `argument` method. The first element of the commands input arguments will always be the name of the command (this item is set early on in the command execution life cycle). The remaining items will be the input arguments in the order they were defined in the command's signature. The items are stored such that the key of the argument is the name given to the argument and the associated value is either the user supplied value or the specified default value of the argument.

Because the `command` argument is set internally by framework components the argument name `command` is considered a reserved argument name. Attempting to name a command argument `command` will result in an error stating something similar to "An argument with the name 'command' already exists" to be issued. Similar errors will be issued whenever there is conflict between argument or option names regardless if they are built-in or user defined.

8.2.2 `hasArgument($name)`

The `hasArgument` method can be used to determine if an argument exists by name. It accepts the `$name` of the argument as its only argument. This method returns a boolean value indicating whether or not the argument exists.

```php
<?php

// ...

function handle()
{
    // Check if an argument exists.
    if ($this->hasArgument('firstName')) {
        // The firstName argument was supplied.
    }
}

// ...
```

8.2.3 `option($key = null)`

The `option` method is used to retrieve either a specific option from the users input or all options from the user input. To access a single option supply a value for the `$key` method parameter. To retrieve all the input options, supply a value of `null` (the default value) for the `$key` method parameter.

Assuming a command signature like so:

```
example:command {--path=} {--psr}
```

Executed like so:

```
php artisan example:command --psr --path=storage/framework
```

We could access user input options like so:

```php
<?php

// ...

function handle()
{
    // Get the psr option value. Since the option
    // is acting like a flag, its value will be
    // either true or false, depending on if
    // the options presence when executed.
    $psr = $this->option('psr');

    // Get the option value of the path option.
    $path = $this->option('path');

    // Get all the input options.
    $options = $this->option();
}

// ...
```

When a command similar to the above example has been executed, the `$psr` variable would hold a value of `true` since the `psr` option was present when the command was executed. If the `psr` option was not present when the command was executed (such as when issuing the command `example:command`—with no options) the `$psr` variable would contain a value of `false`. The `$path` variable would hold the value `storage/framework` (since that was specified in the example command execution).

When options are not supplied and do not have a default value their value will be set to `null`. As an example, issuing the following command will result in the `path` option value to be set to `null`:

```
php artisan example:command
```

Similar to how arguments have a `command` value that is always set, there are numerous default options that are available to your commands. After the above example command has been executed the `$options` variable would reference an array similar to the following output:

```
array [
  "path"           => "storage/framework"
  "psr"            => true
  "help"           => false
  "quiet"          => false
  "verbose"        => false
  "version"        => false
  "ansi"           => false
  "no-ansi"        => false
  "no-interaction" => false
  "env"            => null
]
```

Any options that your command explicitly defines will appear at the beginning of the resulting array. The default options that appear serve different purposes that are common to most commands.

8.2.4 `hasOption($name)`

The `hasOption` method is used to determine if an option exists. It accepts the `$name` of the option as its only argument. This method returns a boolean value indicating whether or not the option exists.

```php
<?php

// ...

function handle()
{
    // Determine if an option is present.
    if ($this->hasOption('no-interaction')) {
        // The `--no-interaction` option has been set.
    }
}

// ...
```

8.3 Default Command Options (Global Command Options)

As stated in the previous section there are numerous default options that are available to your commands. These options are generally referred to as global command options and are part of Symfony's Console Component[1].

[1]http://symfony.com/doc/current/components/console/introduction.html

The following table provides a general overview of each of the global options. Each global option will also be discussed in its own section with more details.

Option	Shortcut	Description
`help`	`h`	Displays help information about the command.
`version`	`V`	Displays the application version.
`ansi`		Enable output coloring (ANSI output), if available.
`no-ansi`		Disables output coloring (ANSI output).
`no-interaction`	`n`	Disables any interactive questions.
`env`		Determines the environment the command should run under.
`quiet`	`q`	Suppresses any messages the command may generate.
`verbose`	`v, vv, vvv`	Determines the verbosity of messages the command generates.

8.3.1 Application Version and Command Environments

The `--version` option can be used to view the current version of the Laravel framework. This option will override any commands default behavior. This option is generally used without specifying the name of any command:

```
# Using the full option name.
php artisan --version

# Using the shortcut. Notice the uppercase V.
php artisan -V
```

The above example commands would output something similar to `Laravel Framework version 5.2.29`. The actual phrasing and version number will possibly be different from your results.

The `--env` option can be used to set which environment the command should be ran under. The `env` command shows the current configured Laravel environment. Assuming the default environment is `local` we can observe the effects of the `--env` option by running the following example commands. The command output appears above the commands as a comment.

```
# Current application environment: local
php artisan env

# Current application environment: production
php artisan env --env=production
```

8.3.2 Command Verbosity Levels and Output Control

By default commands output messages with output coloring, otherwise known as ANSI output (except for on Windows systems). The `--ansi` option can be used to force ANSI output:

```
# Force ANSI output.
php artisan example:command --ansi
```

Conversely, the `--no-ansi` option can be used to disable ANSI output:

```
# Disable ANSI output.
php artisan example:command --no-ansi
```

Commands often output numerous messages or interactively ask for input. The `--quiet` option can be used to suppress all output from the command. As an example the following command invocations would not display any output in the console:

```
# Using the full option name.
php artisan env --quiet

# Using the option shortcut.
php artisan env -q
```

The `--verbose` option is the logical opposite of the `--quiet` option. The `--verbose` option can be used to specify the verbosity of the messages a command generates. While this option is available on all commands it is up to each individual command to recognize the verbosity levels and update its output accordingly. The verbosity levels are defined as constants in Symfony's `Symfony\Component\Console\Output\OutputInterface` class. The following table details the verbosity levels, their integer value and their corresponding option:

Verbosity Level	Value	Option
`OutputInterface::VERBOSITY_QUIET`	16	`-q`, `--quiet`
`OutputInterface::VERBOSITY_NORMAL`	32	No option. This is the default verbosity level.
`OutputInterface::VERBOSITY_VERBOSE`	64	`-v`, `--verbose`
`OutputInterface::VERBOSITY_VERY_-VERBOSE`	128	`-vv`
`OutputInterface::VERBOSITY_DEBUG`	256	`-vvv`

The following code example can be used to determine which verbosity level is currently in use.

```php
<?php

use Symfony\Component\Console\Output\OutputInterface;

// Beginning of class omitted...

function handle()
{
    // First get the verbosity level.
    $verbosity = $this->output->getVerbosity();

    if ($verbosity == OutInterface::VERBOSITY_QUIET) {
        // Verbosity level is quiet.
        // Set by using the --quiet or -q options.
    } else if ($verbosity == OutputInterface::VERBOSITY_NORMAL) {
        // Verbosity level is normal.
        // Set by not supplying any verbosity options.
    } else if ($verbosity >= OutputInterface::VERBOSITY_VERBOSE) {
        // Verbosity level is at least verbose.
        // Set using the --verbose or -v options.
    } else if ($verbosity >= OutputInterface::VERBOSITY_VERY_VERBOSE) {
        // Verbosity level is at least very verbose.
        // Set using the -vv option.
    } else if ($verbosity >= OutputInterface::VERBOSITY_DEBUG) {
        // Verbosity level is at the highest level.
        // Set using the -vvv option.
    }

}

// Ending of class omitted...
```

Laravel also exposes a few methods to make comparing verbosity levels easier. The following example shows a cleaner way to check verbosity levels compared to the previous example:

```php
<?php

// Beginning of class omitted...

function handle()
{

    if ($this->output->isQuiet()) {
        // Verbosity level is quiet.
    }

    if ($this->output->isVerbose()) {
        // Verbosity level is at least verbose.
    }

    if ($this->output->isVeryVerbose()) {
        // Verbosity level is at least very verbose.
    }

    if ($this->output->isDebug()) {
        // Verbosity is at the highest level.
    }

}

// Ending of class omitted...
```

Sometimes when interacting with commands exceptions are thrown. For example, the following command purposely throws an exception:

```php
<?php

// Beginning of class omitted...

function handle()
{
    throw new \Exception('An example exception');
}

// Ending of class omitted...
```

When the above command is executed the default behavior is to simply indicate that an exception has been thrown and would display something similar to the following output:

```
  [Exception]
  An example exception.
```

When the verbosity level is at least `OutputInterface::VERBOSITY_VERBOSE` (by setting the `-v` or `--verbose` option) the entire stacktrace for the exception will be displayed in the command's output. For example called the above command output and setting the verbosity to at least `OutputInterface::VERBOSITY_VERBOSE` would output something similar to the following (some of the output has been indented to avoid wrapping):

```
  [Exception]
  An example exception.


Exception trace:
 () at /home/vagrant/Code/Laravel/app/Console/Commands/ExampleCommand.php:42
 App\Console\Commands\ExampleCommand->handle() at n/a:n/a
 call_user_func_array() at /home/vagrant/Code/Laravel/vendor/laravel/
    framework/src/Illuminate/Container/Container.php:507
 Illuminate\Container\Container->call() at /home/vagrant/Code/Laravel/vendor/
    laravel/framework/src/Illuminate/Console/Command.php:169
 Illuminate\Console\Command->execute() at /home/vagrant/Code/Laravel/vendor/
    symfony/console/Command/Command.php:256
 Symfony\Component\Console\Command\Command->run() at /home/vagrant/Code/
    Laravel/vendor/laravel/framework/src/Illuminate/Console/Command.php:155
 Illuminate\Console\Command->run() at /home/vagrant/Code/Laravel/vendor/
    symfony/console/Application.php:791
 Symfony\Component\Console\Application->doRunCommand() at /home/vagrant/Code/
    Laravel/vendor/symfony/console/Application.php:186
 Symfony\Component\Console\Application->doRun() at /home/vagrant/Code/
    Laravel/vendor/symfony/console/Application.php:117
 Symfony\Component\Console\Application->run() at /home/vagrant/Code/Laravel/
    vendor/laravel/framework/src/Illuminate/Foundation/Console/Kernel.php:107
 Illuminate\Foundation\Console\Kernel->handle() at /home/vagrant/Code/
    Laravel/artisan:35
```

The `--no-interaction` default option will be discussed in the section "Prompting for User Input". The `--no-interaction` option (when specified) will effectively disable all the features discussed in that section.

8.4 Prompting for User Input

The previous sections have discussed how to gather user input using arguments and options. However, sometimes it is beneficial to gather input from users interactively by asking questions, providing choices are simply prompting users for information at specific points in a command's execution. Laravel makes this a relatively simple thing to accomplish.

Laravel commands provide the following instance methods to interactively gather information from users:

Method	Description
`ask`	Prompts the user for input.
`anticipate`	Prompts the user for input. This method can be used to provide auto-completion to suggest default choices. This method is an alias of the `askWithCompletion` method.
`askWithCompletion`	Prompts the user for input. This method can be used to provide auto-completion to suggest default choices.

Method	Description
`secret`	Prompt the user for input without showing the answer in the console.
`choice`	Allows the user to select an answer from a defined set of answers.
`confirm`	Prompts a user with a yes or no question.

Each of the mentioned methods provide numerous options to customize the behavior of the prompt that is displayed to the user. All of the methods accept default values as well as allow you to customize the message that is displayed to the end user when the command executes. All input interactive input methods will return the user's input to the command. Each of these methods will be discussed in greater detail in their own sections.

8.4.1 Prompting for User Input

The `ask` method is the simplest way to to prompt a user for input when executing a command. The `ask` method requires a question (the message displayed to the user) to be supplied as its first argument and an optional default value. The following example `handle` method demonstrates its usage:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class ExampleCommand extends Command
{

    protected $signature = 'example:command';

    function handle()
    {
        // Ask the user for their name.
        $name = $this->ask('What is your name?');
    }

}
```

When the command is executed, the user will see something similar to the following in their terminal:

```
 What is your name?:
 >
```

If the user attempts to continue execution of the command without specifying a value they will receive an error stating "[ERROR] A value is required". This error can be avoided by providing a default value that will be used when the user does not supply a value:

```php
<?php

// ...

function handle()
{
    // Ask the user for their name.
    $name = $this->ask('What is your name?', 'John Doe');
}

// ...
```

Another useful thing to prompt users for is an answer to a simple "yes or no" question. The following code example demonstrates a naive attempt at implementing this behavior. It simply asks the user if they want to continue the execution of the command; it does this by checking the user's input. If the user supplied the string `n` (or even `N` since everything is converted to lowercase) the command will stop execution; everything else will cause the command to continue running.

```php
<?php

// ...

public function handle()
{
    $shouldContinue = $this->ask('Do you want to continue? [Y|n]', 'Y');

    if (strtolower($shouldContinue) == 'n') {
        // Do not continue.
        return 0;
    }

    // Other command actions.

}

// ...
```

Laravel provides a simpler way to accomplish this via the `confirm` instance method. The `confirm` method defines two parameters. The first is required and is the message to prompt the user with. The second argument that can be supplied to the `confirm` method is the default response. The following examples demonstrate the `confirm` method:

```php
<?php

// ...


function handle()
{
    $shouldContinue = $this->confirm('Do you want to continue?');

    if (!$shouldContinue) {
        // Do not continue.
        return 0;
    }

    // Other command actions.
}

// ...
```

When the above command is executing the following prompt would be displayed to users:

```
 Do you want to continue? (yes/no) [no]:
 >
```

The `confirm` method will present the user with the supplied message as well as the accepted values (in this case the acceptable values are either `yes` or `no`) in parenthesis. The method will also display the default value in square brackets (in the previous example, and by default, that value is `no`). The default value can be changed by supplying a boolean value as the second argument to the `confirm` method:

```php
<?php

// ...


$shouldContinue = $this->confirm('Do you want to continue?', true);

// ...
```

Now when the user executes the command they will see the following prompt (take note of the default value that is being displayed):

```
Do you want to continue? (yes/no) [yes]:
>
```

Most of Laravel's command features are wrappers or extensions of Symfony's console helpers. Developers coming from a Symfony background will recognize the `confirm` method, but it should be noted that there are subtle differences. In Laravel, the the default response for `confirm` is `no`, while in Symfony the default response is `yes`.

The `confirm` method is not very customizable. It defaults to expecting either a yes or no response. There are times when it might be beneficial to customize the default behavior. The `confirm` method is a helpful wrapper around Symfony's `ConfirmationQuestion`. We can directly instantiate an instance of this class to customize the behavior. The definition for the `ConfirmationQuestion` constructor is:

```
/**
 * Constructor.
 *
 * @param string $question        The question to ask to the user
 * @param bool   $default         The default answer to return, true or false
 * @param string $trueAnswerRegex A regex to match the "yes" answer
 */
public function __construct(
    $question,
    $default = true,
    $trueAnswerRegex = '/^y/i') {}
```

The first argument that must be supplied to the `ConfirmationQuestion` is the question that will be asked of the end user. The other two parameters are not required by default and are used to customize the default behavior. An argument for the `$default` parameter determines the default behavior (or response) of the question. The `$trueAnswerRegex` works in conjunction with the `$default` value; it should contain a regular expression that represents the accepted value for the truth answer. The following example demonstrates how to use the `ConfirmationQuestion` within a Laravel command:

```
<?php

namespace App\Console\Commands;

use Symfony\Component\Console\Question\ConfirmationQuestion;
use Illuminate\Console\Command;

class ExampleCommand extends Command
{

    // ...

    public function handle()
    {
        $question = new ConfirmationQuestion(
            'Do you want to continue? (HIja'/ghobe')',
            true,
            '/^h/i'
        );

        $shouldContinue = $this->output->askQuestion($question);

        if (!$shouldContinue) {
            // Do not continue.
            return 0;
        }

        // Other command actions.
    }

}
```

When a command similar to the previous example has executed the user would see something similar to the following output. The question will accept `HIja'` yes as the answer for `yes`.

```
 Do you want to continue? (HIja'/ghobe') (yes/no) [yes]:
 >
```

The above example still contains the `(yes/no) [yes]` default string. This string is hard-coded into Symfony's `Symfony\Component\Console\Helper\SymfonyQuestionHelper`. The `SymfonyQuestionHelper` is used internally by Symfony's `SymfonyStyle`, which in turn is used by Laravel's `OutputStyle`. This default text *can* be changed but would require overriding many methods and classes in the inheritance hierarchy.

8.4.2 Suggesting Input

The `ask` method is useful for retrieving arbitrary input from the user; however, it is sometimes useful to suggest options to users while also providing the option to enter any arbitrary value. The `anticipate` and `askWithCompletion` methods can be used to prompt a user for input while also providing default suggestions. The definition for both the `anticipate` and `askWithCompletion` methods (except for the method name) is:

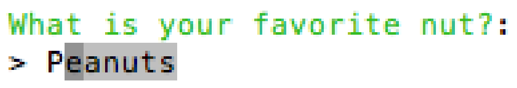
```php
/**
 * Prompt the user for input with auto completion.
 *
 * @param  string  $question
 * @param  array   $choices
 * @param  string  $default
 * @return string
 */
public function anticipate($question, array $choices, $default = null) {}
```

An argument for the `$question` parameter will become the prompt that is displayed to the user. An array of `$choices` must be supplied as these will become the options that are suggested to the end user. An optional default value can be supplied (via the `$default` parameter) which will be used if the user does not enter a value.

The following example demonstrates the usage of the `anticipate` method (note that the `askWithCompletion` method is used in exactly the same way):

```php
<?php

// ...


function handle()
{
    $favoriteNut = $this->anticipate('What is your favorite nut?', [
        'Peanuts',
        'Almond',
        'Cashew'
      ]);
}

// ...
```

When the user executes the command in a terminal, they will see something similar to the following figure:



```
What is your favorite nut?:
> Peanuts
```

As the user types any matching choices will be suggested to them. In most terminals the style changes and displays the suggestion with a different background color (such as in the image above). Other terminals may even display a list of suggestions that match the user's input.

8.4.3 Prompting for Sensitive User Input

At times it is required to prompt users for sensitive information at the command line (such as passwords or secret keys). This information could be gathered with the `ask` method, but the user's input would still be written to the console. The `secret` method can be used to prompt users for input while also hiding the typed characters from the console. The signature for the `secret` method is:

```php
/**
 * Prompt the user for input but hide the answer from the console.
 *
 * @param  string  $question
 * @param  bool    $fallback
 * @return string
 */
public function secret($question, $fallback = true) {}
```

It accepts a `$question`, or message, that will be displayed to the user as the prompt. The method also defines a `$fallback` parameter (which defaults to `true`) which accepts a truth value as its argument. The `$fallback` argument is used to determine if the command should continue to collect the user's input if the user's input cannot be hidden. If `$fallback` is set to `true`, the command will continue to collect user input even if the user's input cannot be hidden in the console. If `$fallback` is set to `false` and the input cannot be hidden an instance of `Symfony\Component\Console\Exception\RuntimeException` will be thrown with the message "Unable to hide the response".

> Laravel internally relies on Symfony to attempt to hide the user's input from the console. On Windows systems the `hiddeninput.exe` binary will be used to attempt to hide input from the user. On Linux and Unix systems with stty available (a utility used to change command line options and settings), ssty will be used to hide the user's input. If stty is not available on a non-Windows system certain shells will be tried (such as `bash`, `zsh`, `ksh`, or `csh`) to attempt to hide the user's input. If none of the previously mentioned methods work to hide the user's input an exception will be thrown stating something similar to "Unable to hide the response".

The following example demonstrates how to call the `secret` method:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class ExampleCommand extends Command
{

    // ...

    public function handle()
    {
        // Previous command actions, such as getting
        // the user's email or username.

        $password = $this->secret('Password');

        // Other command actions.
    }

    // ...

}
```

After a command similar to the following has executed the user would see output similar to the following in the console:

```
 Password:
 >
```

When the user types their response in the console (assuming the input could be hidden and the `$fallback` has been set to `true`) the input would be hidden and their response would be assigned to the `$password` variable. Typically you would gather the user's username or email address before requesting their password.

8.4.4 Providing Choices for User Input

There are times when it is beneficial to allow a user to select an option for an existing set of choices. This can be accomplished with the `ask` method but would require logic to maintain the state of users responses over multiple questions that could quickly become unmaintainable and frustrating to use. The `choice` method can be used to accomplish this task quite easily. The signature for the `choice` method is:

```php
/**
 * Give the user a single choice from an array of answers.
 *
 * @param  string  $question
 * @param  array   $choices
 * @param  string  $default
 * @param  mixed   $attempts
 * @param  bool    $multiple
 * @return string
 */
public function choice(
    $question,
    array $choices,
    $default = null,
    $attempts = null,
    $multiple = null) {}
```

An argument for the `$question` parameter will become the prompt that is displayed to the user. The `$choices` array is also required and will be displayed to the user. The following code example demonstrates the basic usage of the `choice` method:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class ExampleCommand extends Command
{

    // ...

    function handle()
    {
        // Ask a user what their favorite juice is.
        $favoriteChoice $this->choice('What is your favorite juice?', [
                'Cranberry Apple',
                'Orange',
                'Pomegranate'
            ]);
    }

    // ...

}
```

When the command is executed, the user would see something similar to the following output in the terminal:

```
 What is your favorite juice?:
  [0] Cranberry Apple
  [1] Orange
  [2] Pomegranate
 >
```

At this point the user can type their response. A valid response is either the key or value of the choices (the first valid value that can satisfy the user's input will be used). An associative array can also be supplied; the key/value pairs will displayed to the user. Consider the following example and output:

```php
<?php

// ...

function handle()
{
    $favoriteJuice = $this->choice('What is your favorite juice?', [
                'Cra'    => 'Cranberry Apple',
                'Ora'    => 'Orange',
                'Pom'    => 'Pomegranate',
                'Orange' => 'Prune'
            ]);
}

// ...
```

The prompt the user would see would be similar to the following:

```
 What is your favorite juice?:
  [Cra    ] Cranberry Apple
  [Ora    ] Orange
  [Pom    ] Pomegranate
  [Orange] Prune
 >
```

If a user supplies an invalid value an error stating something similar to `[ERROR] Value <VALUE> is invalid` where `<VALUE>` is user supplied value. It is important to note that the error is not a fatal error or an exception that can be caught. The command will continue to ask for valid input until the user terminates the command (see the section "Limiting the Number of Attempts" for more information on how to get around this).

8.4.4.1 Resolving Ambiguity Between Choices

The previous example was chosen specifically because of the ambiguity between the `Orange` value and the `Orange` key. If the user types the value `Orange` at the terminal the `$favoriteJuice` variable would contain the value `Ora`. This is because matching values are given precedence over matching keys. In the following example the value would still be `Ora`:

```php
<?php

// ...

function handle()
{
    $favoriteJuice = $this->choice('What is your favorite juice?', [
                'Cra'    => 'Cranberry Apple',
                'Pom'    => 'Pomegranate',
                'Orange' => 'Prune',
                'Ora'    => 'Orange'
            ]);
}

// ...
```

8.4.4.2 Providing a Default Value for Choices

The third `$default` parameter allows you to specify a default value. The default value follows the same precedence rules as when a user supplies their own value. The following example demonstrates how to supply a default value:

```php
<?php

// ...


function handle()
{
    $favoriteJuice = $this->choice('What is your favorite juice?', [
                'Cra'    => 'Cranberry Apple',
                'Pom'    => 'Pomegranate',
                'Orange' => 'Prune',
                'Ora'    => 'Orange'
            ], 'Ora');

}

// ...
```

If a command similar to the above example was executed and the user did not supply a value the `$favoriteJuice` variable would contain the value `Ora`.

8.4.4.3 Limiting the Number of Attempts

If a user enters an invalid value when making a choice an error message will be displayed and they will be prompted to select a valid choice. This will continue until the user makes a valid selection or they terminate the execution of the command. This behavior can be avoided by specifying a maximum number of attempts the user has to make a valid selection. This can be done by supplying an argument for the `$attempts` parameter. By default `$attempts` is set to `null` (a value of `null` allows the user to make as many attempts as they want).

The following example demonstrates the usage of the `$attempts` parameter:

```php
<?php

// ...


function handle()
{
    $favoriteJuice = $this->choice('What is your favorite juice?', [
                'Cra' => 'Cranberry Apple',
                'Ora' => 'Orange'
            ], null, 2);

}

// ...
```

When a user runs a command similar to the previous example and provides an invalid choice two times in a row an instance of `Symfony\Component\Console\Exception\InvalidArgumentException` will be thrown. This exception can be handled within the command to deal with the invalid user input if desired. If the exception is not handled the command's execution will be halted and the exception's error message will be displayed to the user.

8.4.4.4 Allowing Users to Select Multiple Values

It is also possible to allow users to select multiple values. To allow users to specify multiple values supply a truth value that evaluates to `true` for the fifth (`$multiple`) parameter. A user separates multiple values by commas.

The following example demonstrates how to allow users to select multiple values (there is also no default value and users have unlimited attempts):

```php
<?php

// ...


function handle()
{
    // Allow users to specify multiple juice varieties. Notice the fifth
    // argument is `true`.
    $favoriteJuice = $this->choice('What are your favorite juices?', [
                'Cra' => 'Cranberry Apple',
                'Ora' => 'Orange'
            ], null, null, true);

}

// ...
```

If the user typed `Cra, Ora` as their input both of the options would be placed in an array and assigned to the `$favoriteJuice` variable. The resulting array would be structured similar to the following output:

```
array(2) {
  [0] =>
  string(3) "Cra"
  [1] =>
  string(3) "Ora"
}
```

8.4.5 The `--no-interaction` Default Option

The `--no-interaction` default option is similar to the verbosity options in that it allows users of your commands to adjust the level of output your command generates. However, unlike the verbosity options the `--no-interaction` option has drastic effects on your command if you rely on the interactive prompts to retrieve user input. The option is used as a flag and will effectively disable all of the interactive prompts (such as `ask`, `secret`, etc):

```
# Execute an example command, disabling all interactive user input.
php artisan example:command --no-interaction

# The same command using the option shortcut.
php artisan example:command -n
```

Since this command disables all of the interactive prompts consider providing alternative ways to provide the input to your command (think of such situations where users want to automate the calls to your command). You can check to see the state of user interaction in the following ways:

```php
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class TestCommand extends Command
{

    // ...

    public function handle()
    {
        // Checking to see if the user has disabled
        // interactive mode by checking for the
        // existence of the --no-interaction
        // user defined option.
        if ($this->hasOption('no-interaction')) {
            // The user has disable interactive mode.
        }

        // Check to see if the current command environment
        // supports interactive communication with the
        // user by calling an instance method on the
        // commands input implementation.
        if ($this->input->isInteractive()) {
            // Interactive prompts are available for the
            // current command session. It is okay to
            // prompt the user interactively to get
            // their input.
        }
    }

    // ...

}
```

9. Command Output

The previous sections have discussed the various ways to defined input expectations and retrieve the input from users. This section will focus on how to communicate to the users of your command using various command output methods. Laravel makes it very easy to communicate with your users, providing useful helper methods to quickly generate tables, progress bars and other common output elements.

The following table contains all of the output methods that Laravel provides directly. There are additional methods that are available that are inherited from Symfony's `Symfony\Component\Console\Style` console output style. These additional methods will be discussed after the Laravel provided methods. The table contains the name of the method, a general description of the method, the default font color for the underlying console style and the default background color of the underlying console style. Any color specified as `Inherited` indicates that the underlying console style does not specify an explicit color or styling.

Method	Description	Default Font Color	Default Background Color
`warn`	Writes a line as a warning.	Yellow	Inherited
`error`	Writes a line formatted as an error.	White	Red
`question`	Writes a line formatted as a question.	Black	Cyan
`comment`	Writes a line formatted as a comment. Same formatting as Symfony's `note`.	Yellow	Inherited
`line`	Writes a line without any special formatting.	Inherited	Inherited
`info`	Writes a line formatted as an information line.	Green	Inherited
`table`	Generates the padding and formatting to display a table in the console.	See section on tables.	See section on tables.

9.1 General Output

Outputting simple lines (without styling) is a straightforward process using the `line` method. The signature for the `line` method is:

```
/**
 * Write a string as standard output.
 *
 * @param  string  $string
 * @param  string  $style
 * @param  null|int|string  $verbosity
 * @return void
 */
public function line($string, $style = null, $verbosity = null) {}
```

The `line` method defines only one required parameter: `$string`. An argument supplied for the `$string` argument will be the text that is written to the console. The `$style` parameter is used to control the styling of the written line. By default the `$style` is `null` which will cause the `$string` to be written without any special styling. The `$verbosity` parameter can be used to control when the message will be displayed to the end user. By default `$verbosity` is set to `null` which means the `$string` will be displayed under all verbosity levels.

The following example will output a simple string to the user:

```
<?php

// ...

function handle()
{
    $this->line('Hello, there!');
}

// ...
```

After the above code has executed, the user would see something similar to the following in their console:

```
Hello, there!
```

The `$style` parameter (when not `null`) requires a string argument. The following table lists the styles that are available by default:

Style Name
comment
question
error
warning
info

The following examples demonstrate how to use the `$style` parameter. These styles will be discussed further in the section “Outputting Lines with Colors”.

```php
<?php

// ...

function handle()
{
    // Outputting lines with styles using the `line` method.
    $this->line('Hello, there!', 'question');
    $this->line('Hello, there!', 'error');
    // ...
}

// ...
```

The `$verbosity` parameter can be used to greatly reduce the amount of code when dealing with different verbosity levels. The following code example can be simplified by taking advantage of the `$verbosity` parameter:

```php
<?php

// ...

function handle()
{
    if ($this->output->isVeryVerbose()) {
        $this->line(
          'Only shown for the VERY_VERBOSE verbosity level.'
        );
    }
}

// ...
```

The same functionality can be achieved using the `$verbosity` parameter and one of the verbosity level constants (defined in Symfony’s `Symfony\Component\Console\Output\OutputInterface`). The following table serves as a refresher of these constants, and the code sample that follows demonstrates how they can be used with the `line` method.

Verbosity Level	Value	Option
`OutputInterface::VERBOSITY_QUIET`	16	`-q`, `--quiet`
`OutputInterface::VERBOSITY_NORMAL`	32	No option. This is the default verbosity level.
`OutputInterface::VERBOSITY_VERBOSE`	64	`-v`, `--verbose`
`OutputInterface::VERBOSITY_VERY_-VERBOSE`	128	`-vv`
`OutputInterface::VERBOSITY_DEBUG`	256	`-vvv`

```php
<?php

use Symfony\Component\Console\Output\OutputInterface;

// ...

function handle()
{
    $this->line(
      'Only shown for the VERY_VERBOSE verbosity level.',
      null,
      OutputInterface::VERBOSITY_VERY_VERBOSE
    );
}

// ...
```

If you would prefer not to import the `OutputInterface` interface the `$verbosity` parameter also understands the option versions of the verbosity levels. The following example is functionally equivalent to the previous two:

```php
<?php

// ...

function handle()
{
    $this->line(
      'Only shown for the VERY_VERBOSE verbosity level.',
      null,
      'vv'
    );
}

// ...
```

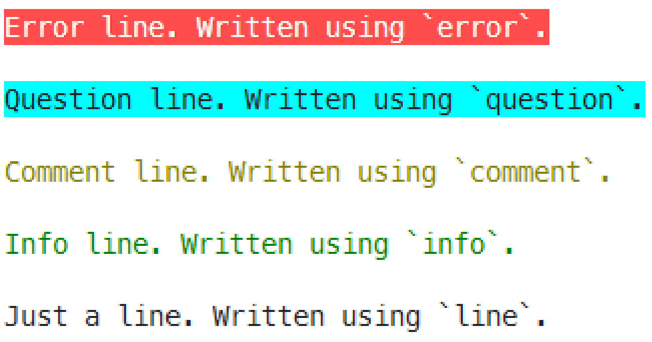
The previous three examples would all output the same result when the very verbose (`-vv`) option is specified when executing the command.

9.2 Outputting Lines with Colors

Laravel commands provide a number of helper methods that build on top of the `line` method. The following code example provides a quick overview of all the available styled output methods:

```php
<?php

// ...

function handle()
{
    // Output a warning message.
    $this->warn('Warning line. Written using `warn`.');

    // Output an error message.
    $this->error('Error line. Written using `error`.');

    // Output a message styled as a question.
    $this->question('Question line. Written using `question`.');

    // Output a message styled as a comment.
    $this->comment('Comment line. Written using `comment`.');

    // Output an informative message.
    $this->info('Info line. Written using `info`.');
}

// ...
```

When the messages are printed in a console window that supports colored output the user would see something similar to the following (note that the blank lines have been added for clarity in the output):



```
Error line. Written using `error`.

Question line. Written using `question`.

Comment line. Written using `comment`.

Info line. Written using `info`.

Just a line. Written using `line`.
```

The `warn`, `error`, `question`, `comment` and `info` methods all support the `$verbosity` parameter found in the `line` method. Using the `$verbosity` parameter in these methods is exactly the same as using it with the `line` method. The following demonstrates how to write a warning only for a certain verbosity level:

```php
<?php

// ...

function handle()
{
    // Output a warning message.
    $this->warn('A warning message only for the very verbose.', 'vv');
}

// ...
```

9.3 Tables

Manually attempting to display tabular output from a command can be a challenging processes. The `table` method can be used to easily generate an ASCII table with the correct padding and headers to display to the console. The signature for the `table` method is:

```php
/**
 * Format input to textual table.
 *
 * @param  array   $headers
 * @param  \Illuminate\Contracts\Support\Arrayable|array  $rows
 * @param  string  $style
 * @return void
 */
public function table(array $headers, $rows, $style = 'default'){}
```

Arguments `$headers` and `$rows` parameters are required, but can be an empty array. If the `$headers` array is not empty and the `$rows` array is empty, the `table` method will simply generate the headers. If there are no headers but there are data rows the `table` method will print the data without any headers. The `$style` parameter is used to control the styling of the table and is set to `default` by default.

The following example demonstrates a simple usage of the `table` method:

```php
<?php

// ...

function handle()
{
    $headers = ['Name', 'Email'];

    $rows = [
       ['John Doe', 'john@example.org'],
       ['Jane Doe', 'jane@example.org']
    ];

    $this->table($headers, $rows);
}

// ...
```

After the command has executed the user would see something similar in their terminal:

```
+----------+------------------+
| Name     | Email            |
+----------+------------------+
| John Doe | john@example.org |
| Jane Doe | jane@example.org |
+----------+------------------+
```

The `table` method can also handle mismatched headers and data elements, but it is generally recommended to have all of the headers and data required before using the `table` method. This is especially true when there are more headers than there are data as it can get confusing when data is placed in the wrong column.

There are a number of default table `$styles` that can be used to quickly change the way the generated tables appear. The following default table styles are available for your convenience:

`default`
: The `default` table style separates the headers and table data as well as provides padding and borders between columns. Each border intersection is accented with the + character. The headers are styled using the `info` text style.

```
+----------+------------------+
| Name     | Email            |
+----------+------------------+
| John Doe | john@example.org |
| Jane Doe | jane@example.org |
+----------+------------------+
```

`borderless`
The `borderless` table style is similar to the `default` table style, but does not have any borders between columns. The header dividers are also larger to make it easier to distinguish between headers and table data.

```
 ========== ==================
  Name       Email
 ========== ==================
  John Doe   john@example.org
  Jane Doe   jane@example.org
 ========== ==================
```

`compact`:
The `compact` table style displays headers and table data without any borders between the rows and cells.

```
 Name     Email
 John Doe john@example.org
 Jane Doe jane@example.org
```

`symfony-style-guide`
The `symfony-style-guide` table style is similar to the `borderless` table style with the difference being the header dividers are less pronounced.

```
 ---------- ------------------
  Name       Email
 ---------- ------------------
  John Doe   john@example.org
  Jane Doe   jane@example.org
 ---------- ------------------
```

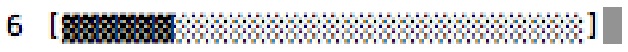
9.4 Progress Bars

Progress bars can be used to update the user on the progress of long running tasks without taking up to much vertical space in the console window. The progress helper methods are available via the `$output` (`Illuminate\Console\OutputStyle`) instance. The methods are inherited from Symfony's `Symfony\Component\Console\Style\SymfonyStyle` class.

A simple progress bar can be created like this:

```php
<?php


// ...

function handle()
{
    $this->output->progressStart();

    for (i = 0; i <= 50; i++) {
        $this->output->progressAdvance();
    }

    $this->output->progressFinish();
}

// ...
```

When the command is executed in the console the user would see output similar to the following example:

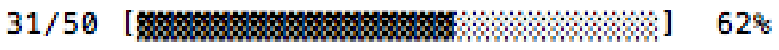

```
6 [▓▓▓▓▓▓▓▓░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░]
```

The `progressStart` method creates a new progress bar instance for the output. It defines a `$max` parameter which can be used to control the maximum amount of steps that the progress bar will be displayed. If it is omitted, as in the previous example, there is no set end to the progress bar and will continue to fill until the `progressFinish` method is called. When an argument for `$max` is supplied each step will take up a percentage of the bar relative to the total amount of steps.

The following example demonstrates how to use the `$max` parameter:

```php
<?php

// ...

function handle()
{
    // Create a new progress bar with a total of 50 steps.
    $this->output->progressStart(50);

    for (i= 0; <= 49; i++) {
        $this->output->progressAdvance();
    }

    $this->output->progressFinish();
}

// ...
```

Progress bars that have a set number of steps are displayed slightly differently in the terminal. The default style is to display the number of steps completed out of the total number of steps (for example `31/50`).



```
31/50 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓░░░░░░░░░░░]  62%
```

9.5 Additional Output Methods

Laravel's `Illuminate\Console\OutputStyle` (an instance of this class is assigned to each commands `$output` property) extends Symfony's `Symfony\Component\Console\Style\SymfonyStyle` class. The `SymfonyStyle` parent class provides many additional methods for sending output to the console window. Each of the following sections will look at one these methods and look at how they are used as well as example output from each command.

9.5.1 `block`

The `block` method is used to format a message as a block of text. Blocks of text generally have a different background color and can have optional padding and labels. The signature for the `block` method is:

```
/**
 * Formats a message as a block of text.
 *
 * @param string|array $messages The message to write in the block
 * @param string|null  $type     The block type (added in [] on first line)
 * @param string|null  $style    The style to apply to the whole block
 * @param string       $prefix   The prefix for the block
 * @param bool         $padding  Whether to add vertical padding
 */
public function block(
    $messages,
    $type = null,
    $style = null,
    $prefix = ' ',
    $padding = false
){}
```

A string or array are expected arguments for the `$messages` parameter (if you have a lot messages to output to the console, consider supplying an array instead of a single string). The messages `$type` determines what will be displayed in brackets when the block is rendered. For example a block with the `SUCCESS` type might render like the following example:

```php
<?php

// ...

function handle()
{
    $this->output->block('This is an example message.', 'SUCCESS');
}

// ...
```

```
[SUCCESS] This is an example message.
```

Supplying a value of `null` for the `$type` will not render any type information for the block. A `$prefix` can be set which will placed at the beginning of the block when it is rendered (note that `null` has been supplied for the `$style` parameter in the following example):

```php
<?php

// ...


function handle()
{
    $this->output->block(
        'This is an example message.',
        'SUCCESS',
        null,
        'Prefix'
    );
}

// ...
```

```
Prefix[SUCCESS] This is an example message.
```

A string argument can be supplied to the `$style` parameter to change how the block is displayed. Customizing styles will be discussed in the section “Creating Custom Console Styles”. However, the following example would create a new block with green background and black text:

```php
<?php

// ...

function handle()
{
    $this->output->block(
        'This is an example message.',
        'SUCCESS',
        'fg=black;bg=green'
    );
}

// ...
```

9.5.2 `title`

The `title` method can be used to format a command's title. It accepts a string (`$message`) that will be printed to the console. This method prints content using the `comment` text style. The following example demonstrates the usage of the `title` method:

```php
<?php

// ...

function handle()
{
    $this->output->title('An example title.');
}

// ...
```

The above would print something similar to the following output:

```
An example title.
=================
```

9.5.3 `section`

The `section` method is similar to the `title` method, but prints a title that is generally used for subsections within a command (the underlining is less pronounced). Like the `title` method, the `section` method also formats text using the `comment` text style. The `section` method is used in the same way as the `title` method:

```php
<?php

// ...

function handle()
{
    $this->output->section('An example section.');
}

// ...
```

The above would print something similar to the following output:

```
An example section.
-------------------
```

9.5.4 `listing`

The `listing` method is useful for displaying an unordered list of elements. It accepts an array of `$elements` (the items that should be displayed as a list) as its only argument. The following example demonstrates its usage:

```php
<?php

// ...

function handle()
{
    $this->output->listing([
        'Chapter One',
        'Chapter Two',
        'Chapter Three'
    ]);
}

// ...
```

The user would see something similar to the following in their console window:

```
 * Chapter One
 * Chapter Two
 * Chapter Three
```

This method does not handled nested arrays. If nested arrays are supplied as an argument an instance of `ErrorException` will be thrown with a message similar to “Array to string conversion”.

9.5.5 `writeln`

The `writeln` method is used to write line(s) of content to the output while adding a newline at the end of each line. The `writeln` method takes either a single string or an array of `$messages` as its first argument. It also allows you to specify options that control how the lines are written via the `$options` parameter. The `$options` parameter allow you to control the verbosity of the message, the output format (raw, colored, etc). This method is generally used internally by various components.

The following demonstrates how to call the `writeln` method:

```php
<?php

// ...

function handle()
{
    $this->output->writeln([
        'Chapter One',
        'Chapter Two',
        'Chapter Three'
    ]);
}

// ...
```

The user would see something similar to the following in their console window. Note that each message in the array appears on its own line.

```
Chapter One
Chapter Two
Chapter Three
```

9.5.6 `write`

The `write` method is similar to the `writeln` method in that it accepts either a single string or an array of `$messages`, as well as a bitmask of `$options`. However, the `write` method does not add a newline character to the end of each message by default. The signature of the `write` method is:

```
/**
 * Writes a message to the output.
 *
 * @param string|array $messages The message as an array of lines or a single
                                 string
 * @param bool         $newline  Whether to add a newline
 * @param int          $options  A bitmask of options (one of the OUTPUT or
                                 VERBOSITY constants), 0 is considered the
                                 same as self::OUTPUT_NORMAL |
                                         self::VERBOSITY_NORMAL
 */
 public function write($messages, $newline = false, $options = 0) {}
```

The following example demonstrates how to call the `write` method, and how it is similar to the `writeln` method (the `writeln` method internally makes a call to the `write` method):

```php
<?php

// ...

function handle()
{
    // Call the `write` method. All items
    // would appear on the same line.
    $this->output->write([
        'Chapter One',
        'Chapter Two',
        'Chapter Three'
    ]);

    // Call the `write` method. All items would appear on their
    // own line. This is is effectively the same as calling
    // the `writeln` method with the same exact arguments
    $this->output-write([
        'Chapter One',
        'Chapter Two',
        'Chapter Three'
    ], true);
}

// ...
```

9.5.7 `newLine`

The `newLine` method is a simple utility method to write newlines (blank lines) to the console window. It accepts a `$count` of the number of blank lines to write; by default it will print one blank line to the console window.

The newLine method was used extensively in the writing of this book to provide ample room in the console window to get clean screen grabs. The newLine method can be used like so:

```php
<?php

// ...

function handle()
{
    $this->line('The first message.');

    // Write one blank line to the console.
    $this->output->newLine();

    $this->line('The second message.');

    // Write six blank lines to the console.
    $this->output->newLine(6);

    $this->line('The third message.');
    $this->line('The fourth message.');
}

// ...
```

The output would be:

```
The first message.

The second message.






The third message.
The fourth message.
```

The following sections discuss methods that have similar method names to those that are available on Laravel's Command class. However, the following methods are defined in Symfony's Symfony\Component\Console\Style\SymfonyStyle class. Each of these methods define only one parameter $message. A single string or an array of messages can be supplied for $message. These methods make an internal method call to the block method.

9.5.8 `text`

The `text` method accepts either an single string or an array of messages. It writes each message to the console without any special formatting. Multiple messages will be written on their own line.

```php
<?php

// ...

function handle()
{
    $this->output->text([
        'Message One',
        'Message Two'
    ]);
}

// ...
```

The output would be similar to:

```
Message One
Message Two
```

9.5.9 `comment`

The `comment` method is very similar to the `text` method. The only difference is that the `comment` method will prefix each message with ‘ // ‘. The following examples demonstrate its usage:

```php
<?php

// ...

function handle()
{
    $this->output->comment([
        'Comment One',
        'Comment Two'
    ]);
}

// ...
```

The output would be:

```
// Comment One
// Comment Two
```

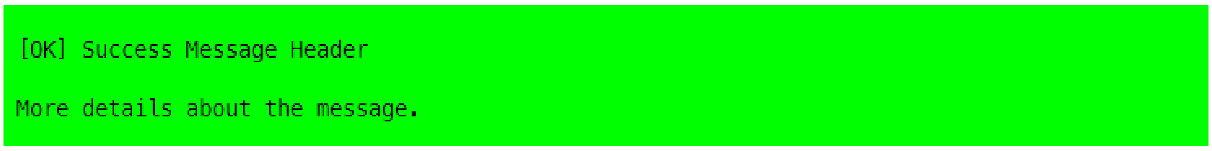
9.5.10 success

The `success` method will generate a block message with the given message or messages. On supported systems the block's background will be green and the foreground color will be black. The rendered block will have the `[OK]` block type.

The following code sample:

```php
<?php

// ...

function handle()
{
    $this->output->success([
        'Success Message Header',
        'More details about the message.'
    ]);
}

// ...
```

would generate output similar to the following image:



```
 [OK] Success Message Header

More details about the message.
```

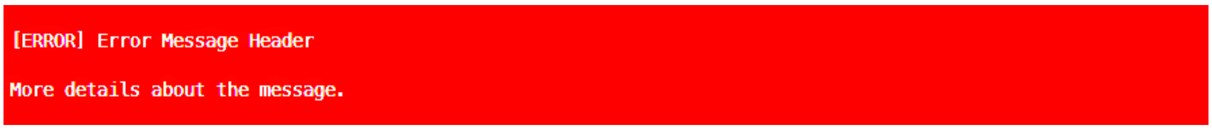
9.5.11 error and warning

The `error` method will generate a block message with the given message or messages. On supported systems the block's background will be red and the foreground color will be white. The rendered block will have the `[ERROR]` block type.

The following code sample:

```php
<?php

// ...

function handle()
{
    $this->output->error([
        'Error Message Header',
        'More details about the message.'
    ]);
}

// ...
```

would generate output similar to the following image:

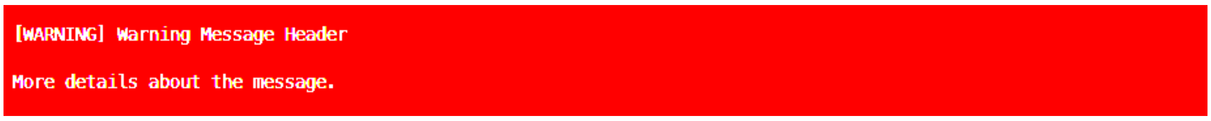


```
[ERROR] Error Message Header

More details about the message.
```

The warning method is identical to the error method. Each method produces the same block styling. The only difference is that the warning method produces a block with the [WARNING] block type. The following example:

```php
<?php

// ...

function handle()
{
    $this->output->warning([
        'Warning Message Header',
        'More details about the message.'
    ]);
}

// ...
```

would produce the following output on supported systems:



```
[WARNING] Warning Message Header

More details about the message.
```

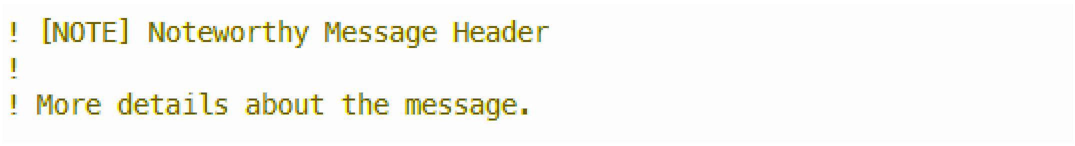
9.5.12 note

The note message will generate a block message the given message or messages. On supported systems the block's background will inherit the consoles default background color (typically white). The foreground, or text color, will be yellow. The rendered block will have the [NOTE] block type as well as the ! prefix.

The following code sample:

```php
<?php

// ...

function handle()
{
    $this->output->note([
        'Noteworthy Message Header',
        'More details about the message.'
    ]);
}

// ...
```

would generate output similar to the following image:



```
! [NOTE] Noteworthy Message Header
!
! More details about the message.
```

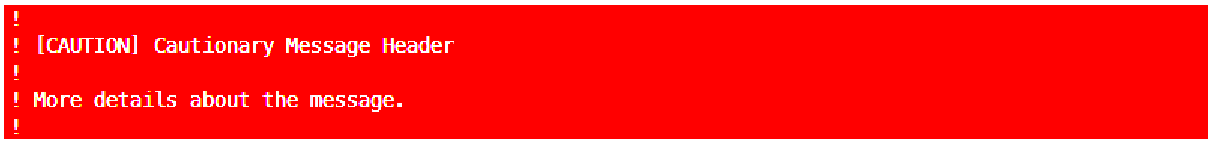
9.5.13 caution

The caution method will generate a block message that combines aspects of many of the previous block methods. On supported systems the block's background will be red and the foreground color will be white. The rendered block will have the [CAUTION] block type and the ! prefix.

The following code sample:

```php
<?php

// ...

function handle()
{
    $this->output->caution([
        'Cautionary Message Header',
        'More details about the message.'
    ]);
}

// ...
```

would generate output similar to the following image:



```
!
! [CAUTION] Cautionary Message Header
!
! More details about the message.
!
```

Helper Functions

“Science makes people reach selflessly for truth and objectivity; it teaches people to accept reality, with wonder and admiration, not to mention the deep awe and joy that the natural order of things brings to the true scientist.”

– Lise Meitner (1878 - 1968), Physicist

Laravel has a number of helper functions that will make developing applications even easier. Most of them are utility based, such as determining if an array has an item, or if a string starts with a particular string or character. All functions will be organized by category and then grouped by their behavior. Functions that do not have a logical grouping will then appear in the order they appear in when viewing the helper files.

10. String Helper Functions

String operations in PHP often seem complicated to developers coming to the language from other language, specifically languages that have a robust object-oriented approach to the provided data types. Operations such as checking if a particular string contains a substring, or if a string starts or ends with another string are not impossible, nor are they difficult, they just appear over-complicated or obscure. Luckily for us, Laravel has an amazing collection of string helper functions that we can use to perform some of the most common actions. The string helper functions are kept in the static class `Illuminate\Support\Str`. The following functions are not necessarily in the same order that they will appear in the class. They have been grouped based on their functionality and purpose.

10.1 Immutable Strings

Laravel's string helper functions treat string values as immutable. This means that any given string passed as a function argument will be unchanged, and a new, modified copy of the string will be returned from the function (unless stated otherwise).

```php
use Illuminate\Support\Str;

$firstString = 'my words';

// Returns `MyWords`
Str::studly($firstString);

// Displays 'my words'
echo $firstString;
```

10.2 `ascii($value)`

The `ascii` helper method accepts only one argument: `$value`. `$value` should always be a string, or something that can be cast into a string. The function converts a string in the UTF-8[1] encoding into it's ASCII[2] equivalent.

This function is useful when communicating with other software platforms that require ASCII, or when complying with protocols (such as HTTP) about how headers, or some values, need to be formatted.

Example use:

[1]http://en.wikipedia.org/wiki/UTF-8
[2]http://en.wikipedia.org/wiki/ASCII

```php
use Illuminate\Support\Str;

// The following will output "a test string"
// in the ASCII encoding.

echo Str::ascii('a test string');
```

As of Laravel 5, the `ascii` function internally uses the stringy[3] library. Previous versions of Laravel have used the patchwork/utf8[4] package.

10.3 `studly($value)`

Studly caps is a way of formatting text with capital letters, according to some pattern. Laravel's pattern is to remove the following word separators and capitalize the first letter of each word it finds (while not affecting the case of any other letters):

- Any number of spaces ‘ ‘
- The underscore _
- The hyphen -

Let's take a look at a few examples to see how this would format a few example strings. The string returned will appear above the function call as a comment. In fact, all of the function calls below will return the string `MyWords`:

```php
use Illuminate\Support\Str;

// MyWords
echo Str::studly('my words');

// MyWords
echo Str::studly('my    words');

// MyWords
echo Str::studly(' my-words');

// MyWords
echo Str::studly(' my-words_');
```

[3]https://github.com/danielstjules/Stringy
[4]https://github.com/tchwork/utf8

Pascal Case

Laravel's `studly` method can also be used to generate Pascal Cased style strings. Both styles are the same as camel case with the first letter capitalized.

Because the `studly` method does not affect the case of any letters after the first letter, we can arrive at output that some users might not expect. Again, the output string will appear above the method call in a comment.

```php
use Illuminate\Support\Str;

// MyWORDS
echo Str::studly('my-WORDS');

// MYWORDS
echo Str::studly('MY_WORDS');
```

10.3.1 `studly_case($value)`

The `studly_case` function is a shortcut to calling `Str::studly`. This function is declared in the global namespace.

10.4 `camel($value)`

Camel casing is similar to studly case such that each word starts with a capitalized letter, with the difference being the first character is lowercased. Like the `studly` method, the `camel` method will not affect the casing of the rest of the word.

The following examples will all return the string `myWords`:

```php
use Illuminate\Support\Str;

// myWords
echo Str::camel('my words');

// myWords
echo Str::camel('my-words');

// myWords
echo Str::camel('my_words');

// myWords
echo Str::camel('   my-words_');
```

Again, the `camel` function does not affect the casing of any characters after the first one.

```php
use Illuminate\Support\Str;

// mYWORDS
echo Str::camel('MY WORDS');

// mywords
echo Str::camel('mywords');
```

10.4.0.1 `camel_case($value)`

The `camel_case` function is a shortcut to calling `Str::camel`. This function is declared in the global namespace.

10.5 `snake($value, $delimiter = '_')`

The `snake` helper method replaces all uppercase letters within the string with the lowercased variant prefixed with the given `$delimiter`. The only exception to this rule is that if the first character of the string is capitalized, it will be replaced by the lowercased variant without the `$delimiter` being prefixed. Because of this behavior, the entire string will be lowercased. The `snake` method trims extraneous whitespace from any string being converted:

Here are a few examples with the result above the function call in comments:

```php
use Illuminate\Support\Str;

// my_words
echo Str::snake('MyWords');

// my-words
echo Str::snake('MyWords', '-');

// m_y_w_o_r_d_s
echo Str::snake('MYWORDS');

// this is _pretty_cool
echo Str::snake('this_is_PrettyCool');
```

10.5.1 `snake_case($value, $delimiter = '_')`

The `snake_case` function is a shortcut to calling `Str::snake`. This function is declared in the global namespace.

10.6 `replaceFirst($search, $replace, $subject)`

The `replaceFirst` helper method is used to replace the first occurrence of a given `$search` string with the provided `$replace` string in the `$subject` string. All three parameters are required and must be strings.

The following examples highlight the basic usage of the `replaceFirst` method. The results of the method call will appear above the call as a comment.

```php
use Illuminate\Support\Str;

// //maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css
Str::replaceFirst(
    'http://', '//',
    'http://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css'
);

// Hello, there!
Str::replaceFirst('there', 'Hello', 'there, there!');
```

10.6.1 `str_replace_first($search, $replace, $subject)`

The `str_replace_first` function is a shortcut to calling `Str::replaceFirst`. This function is declared in the global namespace.

10.7 `trans($id = null, $parameters = [], $domain = 'messages', $locale = null)`

Adapted from the chapter on Translation:

The `trans` helper function is used to return the translation for the given key. It defines a `$key` parameter which corresponds to an array key within the group file. It also accepts an array of replacements (which are passed through the `$parameters` array parameter) and a `$locale` parameter, which can be used to specify the desired locale.

Replacements are passed as a key/value pair. The replacement keys will be matched to any placeholders within the translation. Placeholders begin with a single colon (`:`) followed by a name and a space. For example, in the `validation.php` language file, the following lines can be found:

```php
<?php

return [

    ...

    'accepted'             => 'The :attribute must be accepted.',
    'array'                => 'The :attribute must be an array.',

    ...

];
```

In the above translation lines, `:attribute` is a placeholder than can be replaced using the `$parameters` parameter. The following code examples will demonstrate the results of using the `$parameters` parameter. The results will appear above the method call as a comment.

```php
// The :attribute must be an array.
trans('validation.array');

// The provided value must be an array.
trans('validation.array', ['attribute' => 'provided value']);
```

It should be noted that the placeholder name in the `$parameters` parameter should not contain the leading `:` character.

10.8 `trans_choice($id, $number, array $parameters = [], $domain = 'messages', $locale = null)`

Adapted from the chapter on Translation:

The `trans_choice` helper function is used to pluralize a given `$id`, translating it for the given `$locale`. The `choice` method accepts the `$number` of some collection of objects that it should use when making pluralization decisions. Like the `trans` helper function, it also accepts and array of replacements, using the `$parameters` array.

Assuming the following group file is located at `/resource/lang/en/plural.php`:

```php
<?php

return [
    'books' => 'There is one book.|There are :count many books.',
];
```

It should be noted that pluralized strings allow for multiple messages to be stored in one translation line. Messages are separated by the `|` character, and are ordered such that the message that corresponds to the lower `$number` appear first and messages that appear for any higher `$number` appear sequentially afterwards.

The following code example would select the first message:

```
// There is one book.
trans_choice('plural.books', 1);
```

The following code example would select the second message:

```
// There are 4 books.
trans_choice('plural.books', 4, ['choice' => 4]);
```

The following code examples highlight some more ways the `trans_choice` helper function can be called, and ways they will mostly appear within applications:

```
// There are 4 books.
trans_choice('plural.books', 4, ['choice' => 4]);

// There is one book.
trans_choice('plural.books', 1, ['choice' => 1]);
```

See the section Pluralization section in the Translation chapter for more details specific to just the pluralization syntax.

10.9 `e($value)`

The `e` function is a simple wrapper of PHP's `htmlentities` function. The `e` function utilizes the `UTF-8` character encoding. The `e` function will sanitize user input when displaying it to the browser.

Let's assume that a malicious user was posting on a forum and set the subject of their post to this:

```
<script>alert("hello everyone");</script>
```

If the forum software did not sanitize user output, perfectly valid JavaScript code would be sent to the browser. Since browsers are overly happy to execute any JavaScript included in the document, any forum visitor would see an alert box with `hello everyone` every time a page was loaded that displayed that particular forum post.

To prevent this, use the `e` function to sanitize user input when sending it to the browser:

```
$unsafeClientCode = '<script>alert("hello everyone");</script>';

$safeClientCode   = e($unsafeClientCode);
```

The value of `$safeClientCode` would then be:

```
&lt;script&gt;alert(&quot;hello everyone&quot;);&lt;/script&gt;gt;
```

At this point the browser will render a string that literally represents what they had typed.

11. Array Helper Functions

Laravel provides many helper functions for interacting and manipulating array data structures. Laravel's array functions offer additional functionality on top of PHP's built in array functions. The helper functions in this section are located within the `Illuminate\Support\Arr` static class. There are functions defined in the static class that have counterparts defined in the global namespace. These functions appear below each static function.

11.1 Immutable Arrays

All of the array helper functions below, unless stated otherwise, treat arrays as an immutable data structure. This means that any changes made to an array are returned as a copy of the original array, with the original array remaining unchanged.

Here is a quick example:

```php
use Illuminate\Support\Arr;

$originalArray = [
    'fruit'     => 'orange',
    'vegetable' => 'carrot'
];

$anotherArray = array_add($originalArray, 'vehicle', 'car');
```

Executing `var_dump` on `$originalArray` would produce the following output:

```
array(2) {
    ["fruit"] "orange"
    ["vegetable"] "carrot"
}
```

This is because Laravel's functions do not affect the original array (unless stated otherwise).

Running `var_dump` on `$anotherArray`, however, would produce the expected results:

```
array(3) {
    ["fruit"] "orange"
    ["vegetable"] "carrot"
    ["vehicle"] "car"
}
```

11.2 “Dot” Notation

In addition to treating arrays as immutable, Laravel’s functions support “dot” notation when accessing items in an array. This is similar to JavaScript’s dot notation when accessing object properties.

JavaScript:

```
forms = document.forms;
```

PHP, using Laravel’s helper functions:

```
use Illuminate\Support\Arr;

// Create a multi-dimensional array, to simulate a document and forms.
$testDocument = [
    'document' => [
        'forms' => [

        ]
    ]
];

// Get the forms.
$forms = Arr::get($testDocument, 'document.forms');

// Traditionally, with vanilla PHP:
$forms = $testDocument['document']['forms'];
```

Using dot notation with Laravel’s functions will be explored in more detail in the following sections.

11.3 `add($array, $key, $value)`

The `add` helper method adds the given `$key` and `$value` to an `$array` if the `$key` doesn’t already exist within the given `$array`.

Consider the following code snippet:

```php
use Illuminate\Support\Arr;

$myArray = [
    'animal' => 'Araripe Manakin',
    'plant' => 'Pineland Wild Petunia'
];

$myArray  = Arr::add($myArray, 'insect', 'Extatosoma Tiaratum');
```

The the end result would be:

```
array(3) {
    ["animal"] "Araripe Manakin"
    ["plant"] "Pineland Wild Petunia"
    ["insect"] "Extatosoma Tiaratum"
}
```

11.3.1 `array_add($array, $key, $value)`

The `array_add` function is a shortcut to calling `Arr::add`. This function is declared in the global namespace.

11.4 `get($array, $key, $default = null)`

The `get` helper method will retrieve an item from the given `$array` using dot notation. This allows developers to retrieve items from the array at arbitrary depths quickly, without having to use PHP's array syntax.

Assuming the following array:

```php
$anArray = [
    'nested_array_one' => [
        'nested_array_two' => [
            'key' => 'value'
        ]
    ]
];
```

We can quickly retrieve the value for `key` like so:

```php
use Illuminate\Support\Arr;

$value = Arr::get($anArray, 'nested_array_one.nested_array_two.key');
```

Where the alternative syntax would be:

```php
$value = $anArray['nested_array_one']['nested_array_two']['key'];
```

While PHP's array access syntax may be a little shorter, using the dot notation is easier to read. We can also specify a `$default` value, which will be returned if there is no matching `$key` found in the array.

```php
use Illuminate\Support\Arr;

$value = Arr::get(
            $anArray,
            'nested_array_one.nested_array_two.does_not_exist',
            'default value'
);
```

`$value` would then have the value of `default value`. Using this method is *shorter* than using PHP's array access syntax and the `isset` function, where we would have to check at each level if the key exists or not.

11.4.1 `array_get($array, $key, $default = null)`

The `array_get` function is a shortcut to calling `Arr::get`. This function is declared in the global namespace.

11.5 `pull(&$array, $key, $default = null)`

The `pull` helper method is similar to the `get` method in that it will return a value for the given `$key` in the `$array` (or the `$default`, if it is specified and the `$key` does not exists). The difference is that the `pull` method will remove the item it finds from the array.

Mutable Function

This function affects the original `$array`.

Consider the following array, which is probably familiar to most people these days:

```
$weekDays = [
    1 => 'Sunday',
    2 => 'Monday',
    3 => 'Tuesday',
    4 => 'Wednesday',
    5 => 'Thursday',
    6 => 'Friday',
    7 => 'Saturday'
];
```

A lot of people do not like Mondays. Let's demote `Monday` to the `$theWorstDayOfTheWeek` while also removing it from our `$weekDays`:

```
use Illuminate\Support\Arr;

// Demote 'Monday'
$worstDayOfTheWeek = Arr::pull($weekDays, 2);
```

The `$weekDays` array would now look like this:

```
array(6) {
    [1] "Sunday"
    [3] "Tuesday"
    [4] "Wednesday"
    [5] "Thursday"
    [6] "Friday"
    [7] "Saturday"
}
```

11.5.1 `array_pull(&$array, $key, $default = null)`

The `array_pull` function is a shortcut to calling `Arr::pull`. This function is declared in the global namespace.

12. Application and User Flow Helper Functions

12.1 app($make = null, $parameters = [])

The app function provides access to the `Illuminate\Container\Container` instance. Because `Container` is a singleton, any call to `app()` will return the *same* `Container` instance.

The app function can also be used to resolve registered dependencies from the `Container` instance. For example, to return an instance of the `Illuminate\Auth\AuthManager` class (the class behind the `Auth` facade), invoke `app` with the `auth` argument:

```php
// $authManager will be an instance of
// Illuminate\Auth\AuthManager
$authManager = app('auth');
```

12.2 auth($guard = null)

Used without supplying any arguments for `$guard`, the `auth` helper function is an alternative to using the `Auth` facade. With no arguments, the `auth` function returns an instance of `Illuminate\Contracts\Auth\Factory` (which by default is an instance of `Illuminate\Auth\AuthManager`).

The following example shows to equivalent ways of retrieving the currently logged in user:

```php
use Illuminate\Support\Facades\Auth;

// Get the current user using the `Auth` facade.
$currentUser = Auth::user();

// Get the current user using the `auth` helper.
$currentUser = auth()->user();
```

You can quickly access any custom authentication guards by supplying an argument for the `$guard` parameter. For example if a custom guard was added with the name `customGuard`:

```
if (auth('customGuard')->attempt([
        'email'    => $email,
        'password' => $password
    ])) {
    // Authentication passed.
}
```

When `$guard` is set to `null`, the `AuthManager` instance will internally set `$guard` equal to the value of the `auth.defaults.guard` configuration key (by default this is `web`). The following code samples would produce the same results:

```
// Not passing anything for `$guard`.
if (auth()->attempt([
        'email'    => $email,
        'password' => $password
    ])) {

}

// Passing the default configuration value for `$guard`.
if (auth('web')->attempt([
        'email'    => $email,
        'password' => $password
    ])) {

}
```

12.3 `policy($class)`

The `policy` helper function can be used to retrieve a policy (a policy class can be any valid PHP class) instance for a given `$class`. The `$class` can be either a string or an object instance. If no policies for the given `$class` have been registered with the application an instance of `InvalidArgumentException` will be thrown.

12.4 `event($event, $payload = [], $halt = false)`

The `event` helper is a convenient way to dispatch a particular `$event` to its listeners. An optional `$payload` of data can also be supplied when dispatching the event. When the event is dispatched, all it's listeners will be called and any responses from the listener will be added to an array. The array of responses is generally the return value of the `event` function.

If the `$halt` parameter is set to `true`, the dispatcher will stop calling listeners after any of the listeners return a value that is not `null`. If no listeners have returned a response and all listeners were called, the `event` function will return `null`.

The following examples will highlight the various ways to use the `event` function:

```
/**
 * The following code is simply creating a new event class
 * to signal that a video was watched.
 */

namespace App\Events;

use App\Video;
use App\Events\Event;
use Illuminate\Queue\SerializesModels;

class VideoWasWatched extends Event
{
    use SerializesModels;

    public $video;

    /**
     * Create a new event instance.
     *
     * @param  Video  $video
     * @return void
     */
    public function __construct(Video $video)
    {
        $this->video = $video;
    }
}
```

An event can be fired by the name of a class like so:

```
event('\App\Events\VideoWasWatched', $video);
```

Alternatively, an event can be fired by instantiating a new instance of the event class itself:

```
event(new VideoWasWatched($video));
```

In the above example, a `$payload`was not supplied. This is because the event instance *itself* will become the payload. It is equivalent to the following:

```
$event = new VideoWasWatched($video);
event('App\Events\VideoWasWatched', $event);
```

12.5 `dispatch($command)`

The `dispatch` helper function can be used to push a new job onto the job queue (or dispatch the job). The function resolves the configured `Illuminate\Contracts\Bus\Dispatcher` implementation from the Service Container and then calls the `dispatch` method in the `Dispatcher` instance.

Assuming a variable `$job` exists that contains a valid queueable `Job` the following examples are all equivalent:

```
use Illuminate\Contracts\Bus\Dispatcher;
use Illuminate\Support\Facades\Bus;

dispatch($job);
app(Dispatcher::class)->dispatch($job);
Bus::dispatch($job);
```

12.6 `factory()`

The `factory` function is used to create Eloquent models, generally used when testing an application. The `factory` method can generally take one of four forms:

```
// Return an instance of the 'User' model.
$user = factory('App\User')->make();

// Return two instances of the 'User' model.
$users = factory('App\User', 2)->make();

// Return one instance of the 'User' model, with some modifications
// that were defined earlier and bound to the 'admin' description.
$adminUser = factory('App\User', 'admin')->make();

// Return two instances of the 'User' model, with some modifications
// that were defined earlier and bound to the 'admin' description.
$adminUsers = factory('App\User', 'admin', 2)->make();
```

12.7 `method_field($method)`

The `method_field` helper function is a simple function that returns a new instance of `Illuminate\Support\HtmlString`. The contents of the `HtmlString` instance is a hidden HTML input field with the name _method and the a value of `$method`.

The following example highlights the results of the `method_field` function:

```
method_field('PUT');
method_field('POST');
method_field('GET');
method_field('PATCH');
method_field('DELETE');
```

The return value of the `method_field` function is an instance of `HtmlString`. The corresponding HTML string value for each function call above would be:

```
<input type="hidden" name="_method" value="PUT">
<input type="hidden" name="_method" value="POST">
<input type="hidden" name="_method" value="GET">
<input type="hidden" name="_method" value="PATCH">
<input type="hidden" name="_method" value="DELETE">
```

12.8 validator(array $data = [], array $rules = [], array $messages = [], array $customAttributes = [])

The `validator` helper function is a versatile helper function. If no arguments are supplied to the `validator` function (all arguments are optional), the function will return a `Illuminate\Contracts\Validation\Factory` implementation instance (the default implementation will be an instance of `Illuminate\Validation\Factory`).

Using the the `validator` function without arguments and gaining access to the instance methods is essentially the same as using the `Illuminate\Support\Facades\Validator` facade. The following examples are equivalent; all the examples will make use of the `request` helper function.

```
use Illuminate\Support\Facades\Validator;

// Using the Validator facade.
$validator = Validator::make(request()->all(), [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required'
]);

// Using the validator helper function.
$anotherValidator = validator()->make(request()->all(), [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required'
]);
```

The second example above can be shortened even further. You can supply the arguments for the `make` instance method directly to the `validator` helper function. When you supply arguments to the `validator` helper function it will call and return the value of the `make` instance method:

```
$validator = validator(request()->all, [
    'title' => 'required|unique:posts|max:255',
    'body'  => 'required'
]);
```

13. Configuration, Environment, Security and Logging Helper Functions

13.1 `bcrypt($value, $options = [])`

The `bcrypt` function will return a hashed representation of the given `$value`. The `bcrypt` function also accepts an array of `$options` which can be used to affect how the hash is computed. Each invocation of the `bcrypt` function should produce a different result, even if the input remains the same.

The following example demonstrates how to call the `bcrypt` function:

```php
for ($i = 0; $i < 10; $i++)
{
    // echo the hash and an HTML line break
    echo bcrypt('test'),'<br>';
}
```

The above example would output something similar to this:

```
$2y$10$6b8WZt.Ugwnjjb3JZQH51ecaG.VSjOOO2xCZ3t4s/MGGHU112hhD2
$2y$10$o/uJXcnrNDQraGgk1.VG9.LwssnANCyOEO8tCuiL5RlO33CpGo.Lq
$2y$10$7qWDkO43obCCN4hpNDt2Hut2xbg8xmKQHzZF/m4EdsGUHApXcKLyi
$2y$10$e4srCMoCOaIl9qd2wuk.8e2pBGTxaAu/bDi2CrNlRcNyXxvtYePIy
$2y$10$1MhsM.KaYpwoODuoBi7wmO6jUrMJ0xGaigL6/JMKAgb48CgyFz8tK
$2y$10$wTdq3XAG7/UKT0aO4u9lO.ZRcDiaF5p4fXMViticodID9oC/CTsJO
$2y$10$yHwchZ9HCKZjfnqqulQ7eu61noEwIVZBXKwSZ8.rvYyk9p0SXFNKG
$2y$10$5XvPyJE9EQ6DpOdYzM.NYeR4eDjAzntn2ogytDh1tNU4ebWrHaYvS
$2y$10$V1yb7D7rqqUL7BZkR2c3HOjYHvVB/lRg5cvrL/Hl/KYzrKrTV/tvC
$2y$10$6DAP/IjDTOH3iezOzx/CyuH37ZEDtc6.ADkDEfJUUn/msgUGe5A4S
```

13.1.1 `bcrypt` Options

The following table lists all of the options that are available to use in with the `bcrypt` function.

`bcrypt` Options

Option	Description
`rounds`	Determines how many iterations the `bcrypt` function will internally use.

13.1.1.1 `rounds`

The `rounds` option is used to control how many iterations the underlying Blowfish implementation will use when generating the final hash. The `rounds` value can be any positive integer between `4` and `31` (including both `4` and `31`). If an integer outside the stated range is supplied an `ErrorException` will be thrown. The `rounds` option is synonymous with the `cost` option when using PHP's `password_hash` or `crypt` functions. Increasing the rounds will also increase the time required to compute the hash. The default value for this option is `10`.

```php
// Compute a hash with a 'cost' of 4
bcrypt('test', ['rounds' => 4]);

// Compute a hash with a 'cost' of 5
bcrypt('test', ['rounds' => 4]);

// These will throw an `ErrorException`
bcrypt('test', ['rounds' => 3]);
bcrypt('test', ['founds' => 32]);
```

The number of iterations that will be used can be determined with the following equation:

$$iterations = 2^{rounds}$$

Calling `bcrypt` with the `rounds` option set to `10` will use $2^1 0$ or `1024` iterations.

13.2 `encrypt($value)`

The `encrypt` helper function can be used to encrypt a given `$value`. The function resolves the configured `Illuminate\Contracts\Encryption\Encrypter` implementation from the Service Container and then calls the `encrypt` method on the `Encrypter` instance.

The following examples are all equivalent:

```php
use Illuminate\Support\Facades\Crypt;

// Encrypt using the Crypt facade:
Crypt::encrypt('Hello, Universe');

// Encrypt using the encrypt helper:
encrypt('Hello, Universe');
```

For more information on the `encrypt` instance method, see the "Encryption: Encrypting and Decrypting Data" chapter.

13.3 `info($message, $context = [])`

The `info` helper will write an information entry into Laravel's log files. It allows a `$message` to be set, as well as an optional `$context` (which is an array of data). Whole the `$context` must be an array, the actual elements of the array do not have to be arrays themselves.

The following example shows the usage of the `info` helper. An example of what the function calls would produce in the log files follows the code examples.

```php
// An example message context.
$context = ['name' => 'John Doe', 'email' => 'you@homestead'];

// A log message without context.
info('This is a log message without context');

// A log message where the context is an array.
info('This is a log message', $context);

// A log message where the context is an array of objects.
info('This is another log message', [(object) $context]);
```

The above code would produce results similar to the following (some lines have been indented to prevent wrapping and improve readability):

```
...
[2015-06-15 02:14:43] local.INFO: This is a log message without context
[2015-06-15 02:14:43] local.INFO: This is a log message
  {"name":"John Doe","email":"you@homestead"}
[2015-06-15 02:14:43] local.INFO: This is another log message
  [
   "[object] (stdClass: {\"name\":\"John Doe\",\"email\":\"you@homestead\"})"
  ]
...
```

14. URL Generation Helper Functions

14.1 `action($name, $parameters = [], $absolute = true)`

The `action` helper function can be used to generate a URL to a controller action, which is supplied as an argument to the `$name` parameter. If the controller action requires or accepts parameters these can be supplied by utilizing the `$parameters` parameter.

The following examples will use the following example controller (stored in `app/Http/Controllers/ExampleController.php`):

```
<?php

namespace App\Http\Controllers;

class ExampleController extends Controller {

    public function sayHello($name) {
        return "Hello, {$name}!";
    }

}
```

as well as the following route definition (added to the `web` middleware group in `app/Http/routes.php`):

```
Route::get('sayHello/{name}', 'ExampleController@sayHello');
```

Calling the `action` helper function like so:

```
action('ExampleController@sayHello', ['name' => 'Jim']);
```

would return the following URL:

```
http://laravel.artisan/sayHello/Jim
```

The exact URL that is generated will depend on the current domain of the site.

Notice that when the `action` function was called, we did not need to include the root namespace of the controller (`App\Http\Controllers`). Laravel will automatically add the namespace for you. Any namespaces that are not part of the root namespace must be included (such as `Auth`), however.

14.1.1 Relative URLs

If you do want to generate the full URL to a controller action, simply supply a truth value of `false` for the `$absolute` parameter. The following example demonstrates this:

```
action('ExampleController@sayHello', ['name' => 'Jim'], false)
```

The resulting URL of the function call would then be:

```
/sayHello/Jim
```

15. Miscellaneous Helper Functions

15.1 `dd()`

The dd function is a debug utility that will *dump* a set of values and then die. This means it will create a human-readable representation of the values and then stop execution of the script.

The dd function will take into account if the application is currently running in the console, or if it is returning a response to a client (such as a browser), and then update it's output accordingly.

15.2 `data_get($target, $key, $default = null)`

The `data_get` function is identical in behavior to both the `array_get` and `object_get` function. The difference is that the `data_get` function will accept both objects *and* arrays as it's `$target`.

15.3 `windows_os`

The `windows_os` helper function can be used to determine if the host server is running a Microsoft Windows® operating system. The function returns either `true`—if the server is running Windows®—or `false`—if the server is not running Windows®.

Using this function you can write code like so:

```php
<?php

if (windows_os()) {
    // Windows® specific commands or instructions.
} else {
    // Anything that is not Windows®.
}
```

instead of code like this:

```php
<?php

if (strtolower(substr(PHP_OS, 0, 3)) === 'win') {
    // Windows&reg; specific commands or instructions.
} else {
    // Anything that is not Windows&reg;.
}
```

15.4 `tap($value, $callback)`

The `tap` helper function is used to call a given Closure (provided as an argument to the `$callback` parameter) with the given `$value`. The `tap` helper function will return the reference to the original `$value` as its return value. This allows you to call any number of methods with any type of return values within the `$callback` while maintaining a reference to the original object. This behavior is particularly useful during method chaining. Another way to think of this functions behavior is that it allows you to specify and fix the return value of a series of method calls, regardless of what methods are called within the internal block.

The `tap` helper function was added in Laravel version 5.3 and defined in the `src/Illuminate/Support/helpers.php` file. This function can easily be integrated into older Laravel code bases as it does not depend on any new fundamental framework components.

The following example PHP classes will be used to demonstrate a trivial usage of the `tap` helper function. The classes represent a crude "talker" system where one can add messages and retrieve the final message back:

Stored in the `app/Message.php` file:

```php
<?php

namespace App;

class Message
{

    /**
     * The actual message.
     *
     * @var string
     */
    protected $message = '';

    /**
     * Gets the original message.
     *
     * @return string
     */
    public function getOriginal()
    {
        return $this->message;
    }

    /**
     * Changes the message.
     *
     * @param  $message
     * @return Message
     */
    public function change($message)
    {
        $this->message = $message;

        return $this;
    }

    public function __toString()
    {
        return strtolower($this->message);
    }

}
```

The `Message` class if fairly simple. It is simply a value object that wraps PHP's native string data type. It provides a few methods that can be used to retrieve the original value and change (mutate) the internal value of the message. When the message is cast back to a string it is converted to its lowercased variant. The important thing to notice is that the `change` method returns a reference back to itself.

Stored in the `app/Talker.php` file:

```
<?php

namespace App;

class Talker
{

    /**
    * The actual parts of the message.
    *
```

```
     * @var array
     */
    protected $messageParts = [];

    /**
     * Adds a message part to the end of the message.
     *
     * @param  $message
     * @return Message
     */
    public function atEnd(Message $message)
    {
        array_push($this->messageParts, $message);
        return $message;
    }

    /**
     * Adds a message part to the start of the message.
     *
     * @param  $message
     * @return Message
     */
    public function atBeginning(Message $message)
    {
        array_unshift($this->messageParts, $message);
        return $message;
    }

    /**
     * Returns the string representation of the message.
     *
     * @return string
     */
    public function talk()
    {
        return ucfirst(implode(' ', $this->messageParts));
    }

}
```

The `Talker` class is also very simple. It maintains an array of `Message` instances and allows you to add them to either the beginning or ending of the array of messages. In addition, it allows you to retrieve a string made up of all the individual messages with the first character upper cased. Annoyingly the `atEnd` and `atBeginning` methods return the `Message` instance instead of the `Talker` instance, which makes chaining rather difficult.

The following example will create a simple helper function to remove some of the steps of using the `Talker` and `Message` APIs:

```php
<?php

use App\Talker;
use App\Message;

/**
 * Say something.
 *
 * string  $message
 * @return string
 */
function saySomething($message) {
    $talker = new Talker;
    $talker->atBeginning(new Message)->change($message);
    return $talker->talk();
}
```

Using the new helper function might look like this:

```php
<?php

echo saySomething('Hello World');
```

The user would see the following text in the browser:

```
Hello world
```

It would be nice if we could return the value of the method chain from our function like so:

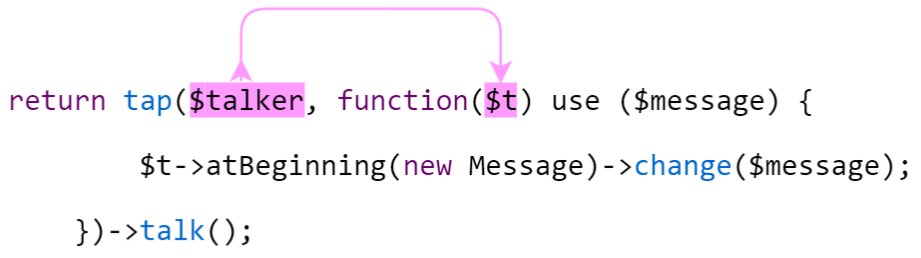
```php
<?php

// ...

function saySomething($message) {
    $talker = new Talker;
    return $talker->atBeginning(new Message)->change($message)->talk();
}

// ...
```

However, that code would not work. This is because the `atBeginning` method on the `Talker` class returns an instance of `Message`. The `talk` method does not exist on the `Message` class, and if it did would probably not have the same behavior as the `Talker` instance method. We can use the `tap` function to get around this quite easily:

```php
<?php

// ...

function saySomething($message) {
    $talker = new Talker;
    return tap($talker, function($t) use ($message) {
        $t->atBeginning(new Message)->change($message);
    })->talk();
}

// ...
```

Tap Function Closure Arguments

In the previous example the Closure defines a parameter named `$t`. When the `tap` function executes, the value of `$talker` will be used as the inner value of `$t` and they will refer to the same `Talker` instance. The following diagram visually demonstrates the flow of data in the `tap` helper function.



```php
return tap($talker, function($t) use ($message) {
        $t->atBeginning(new Message)->change($message);
    })->talk();
```

In the above example we are using the `tap` function to keep our method chain going. In addition we are passing the `$message` parameter into the inner callback function so that we can use it. The above example could actually be simplified further by using the `with` helper function to return a new instance of `Talker`:

```php
<?php

// ...

function saySomething($message) {
    return tap(with(new Talker), function($t) use ($message) {
        $t->atBeginning(new Message)->change($message);
    })->talk();
}
```

```

// ...
```

The previous example is very contrived but does demonstrate a fairly basic use of the `tap` helper function. More common use cases might be when passing around Eloquent models with extensive method chaining.

The following example will make use of the Eloquent model factory builder to create a new instance, modify an attribute, save the model and then return the created model instance:

```php
<?php

use App\User;

$user = tap(factory(User::class)->make(), function($user) {
    $user->setAttribute('name', 'No such a random name')
         ->save();
});
```

After the above example has executed, the attributes of the `$user` model instance might look similar to the following output:

```
array [
    "name"           => "No such a random name"
    "email"          => "Batz.Bridget@example.com"
    "password"       => "$2y$10$aXsXCEx3pkgmNcWpJFkAE.AMHoyk3o8XC6Kth3..."
    "remember_token" => "nayR2BKwaQ"
    "updated_at"     => "2016-06-03 22:29:01"
    "created_at"     => "2016-06-03 22:29:01"
    "id"             => 7
  ]
```

The actual attribute values will differ as they are randomly assigned a value from the model factory. Because of the way we used the `tap` helper function we were able to modify the `name` attribute elegantly, save the model and get the exact return value we wanted.

Collections

"I was told I'd never make it to VP rank because I was too outspoken. Maybe so, but I think men will always find an excuse for keeping women in their 'place.' So, let's make that place the executive suite and start more of our own companies."

– Jean Bartik (1924 - 2011), Computer Programmer

The `Illuminate\Support\Collection` class provides a wrapper around PHP's arrays. The `Collection` class has numerous methods for retrieving, setting and manipulating the data of the underlying array. Each `Collection` instance has an internal array which is used to store the actual collection items.

16. Basic Usage - Collections as Arrays

Because Laravel's `Collection` class is an extravagant wrapper around PHP arrays, it's basic usage should be very familiar to most PHP developers. The `Collection` class also implements PHP's `ArrayAccess`[1] interface so that developers can use PHP's array syntax when working with collections:

```
// Creating a new collection instance.
$collection = new Collection;

// Appending to the collection.
$collection[] = 'First';
$collection[] = 'Second';
$collection[] = 'Third';

// Adding key/value pairs to the collection.
$collection['key']  = 'Some value';
$collection['key2'] = 'Some other value';

foreach ($collection as $key => $value) {
    // Iterate through the array with access to key/value pairs.
}

// Retrieve the value associated with a known key.
$someValue = $collection['key'];
```

After the above code executes, the variable `$someValue` would hold the value `Some value`. It should be noted that even though array syntax works with collections, PHP's array specific functions will not work on an instance of a collection directly. Instead we have to retrieve the underlying array from the collection.

The following code will throw an instance of `ErrorException` stating that `array_values` expects the first parameter to be an object:

[1]http://php.net/manual/en/class.arrayaccess.php

```php
use Illuminate\Support\Collection;

$collection = new Collection;
$collection[] = 'First';
$collection[] = 'Second';
$collection[] = 'Third';

$values = array_values($collection);
```

Instead, the `toArray` method can be used to retrieve a representation of the underlying array:

```php
use Illuminate\Support\Collection;

$collection = new Collection;
$collection[] = 'First';
$collection[] = 'Second';
$collection[] = 'Third';

$values = array_values($collection->toArray());
```

After the above code executes, `$values` would contain a value similar to the following:

```
array (size=3)
  0 => string 'First'  (length=5)
  1 => string 'Second' (length=6)
  2 => string 'Third'  (length=5)
```

16.1 Notes on Basic Usage

Any caveats that apply when working with arrays in PHP also apply to `Collection` instances when treating them as arrays. This means that developers have to check for the existence of items themselves when accessing or iterating over the collection. For example, the following code will produce an `ErrorException` stating that the index `does_not_exist` does not exist:

```php
use Illuminate\Support\Collection;

$collection = new Collection;

$value = $collection['does_not_exist'];
```

Interacting with collections in this manner should be relatively uncommon in practice. The `Collection` class provides many methods that aid developers in such situations. As an example, the `get` method allows retrieval of data from the collection, with the option of returning a default value:

```php
use Illuminate\Support\Collection;

$collection = new Collection;

$value = $collection->get('does_not_exist', 'But I do');
```

After the above code is executed the `$value` variable would have the value `But I do`, and the code will not throw an exception. The chapter Collections: The Public API contains more information about the public methods available.

16.2 Creating a New Collection Instance

There are a few different ways to create a new `Collection` instance with existing data. The first way to create a new instance is to pass an array of items to the `Collection` class's constructor:

```php
use Illuminate\Support\Collection;

// An array of test data.
$testData = [
    'first'  => 'This is the first',
    'second' => 'This is the second',
    'third'  => 'This is third'
];

// Create a new collection instance.
$collection = new Collection($testData);
```

Another way to create a `Collection` instance is to call the static `make` method on the `Collection` class itself:

```php
use Illuminate\Support\Collection;

// An array of test data.
$testData = [
    'first'  => 'This is the first',
    'second' => 'This is the second',
    'third'  => 'This is third'
];

// Create a new collection instance static 'make' method.
$collection = Collection::make($testData);
```

A convenient way to create a new `Collection` instance is to use the `collect` helper function. The `collect` helper function internally returns a new instance of `Collection` passing in any arguments to the class's constructor. It's most basic usage is:

```php
// An array of test data.
$testData = [
    'first'  => 'This is the first',
    'second' => 'This is the second',
    'third'  => 'This is third'
];

// Create a new collection instance using the 'collect' helper.
$collection = collect($testData);
```

17. Collections: The Public API

The `Collection` class exposes a generous public API, consisting of 64 public methods. The `Collection` API exposes methods for retrieving values from a collection, transforming the underlying array, pagination and simple aggregate functions, amongst others. This section will cover most of the methods in the public API, excluding those methods required to implement PHP's `ArrayAccess`[1] and `IteratorAggregate`[2]interfaces, as well as the `getCachingIterator` method.

17.1 `all`

The `all` method can be used to retrieve the underlying array that the collection is using to hold its data. The following code demonstrates the usage of the `all` method:

```php
use Illuminate\Support\Collection;

$items = [
    'first' => 'I am first',
    'second' => 'I am second'
];

$collection = Collection::make($items);

$returnedItems = $collection->all();
```

After the above code has been executed, the `$returnedItems` variable would hold the following value:

```
array (size=2)
  'first'  => string 'I am first'  (length=10)
  'second' => string 'I am second' (length=11)
```

We can also verify that the two arrays are indeed equal:

```php
// true
$areEqual = $items === $returnedItems;
```

It should also be noted that the `all` method will preserve any nested collections:

[1] http://php.net/manual/en/class.arrayaccess.php

[2] http://php.net/manual/en/class.iteratoraggregate.php

```
use Illuminate\Support\Collection;

$items = [
    'first' => 'I am first',
    'second' => 'I am second',
    'third' => new Collection([
            'first' => 'I am nested'
    ])
];

$collection = Collection::make($items);

$returnedItems = $collection->all();
```

At this point, the `$returnedItems` variable would a value similar to the following:

```
array (size=3)
  'first'  => string 'I am first'          (length=10)
  'second' => string 'I am second'         (length=11)
  'third'  =>
    object(Illuminate\Support\Collection)[132]
      protected 'items' =>
        array (size=1)
          'first' => string 'I am nested' (length=11)
```

To return an array, and have any nested collections converted to arrays, see the `toArray` method.

17.2 `toArray`

The `toArray` method is similar to the `all` method in that it will return the underlying array that the collection instance is using. The difference, however, is that the `toArray` method will convert any object instance it can into arrays (namely any object that implements the `Illuminate\Contracts\Support\Arrayable` interface). Consider the following code:

```
use Illuminate\Support\Collection;

$items = [
    'first'  => 'I am first',
    'second' => 'I am second',
    'third'  => new Collection([
            'first' => 'I am nested'
    ])
];

```

```php
$collection = Collection::make($items);

$returnedItems = $collection->toArray();
```

After the above code has executed, the `$returnedItems` variable would contain a value similar to the following output:

```
array (size = 3)
  'first'  => string 'I am first'     (length=10)
  'second' => string 'I am second'    (length=11)
  'third'  =>
    array (size=1)
      'first' => string 'I am nested' (length=11)
```

17.3 chunk($size, $preserveKeys = false)

The `chunk` method is useful when working with large collections in that it allows developers to create smaller collections to work with. The `chunk` method defines two parameters: `$size`, which is used to control how large each newly created collection will be; and `$preserveKeys`, which indicates if the `chunk` method will preserve array keys when creating the new collections.

Assuming the following code:

```php
$testArray = [
    'one'   => '1',
    'two'   => '2',
    'three' => '3',
    'four'  => '4',
    'five'  => '5',
    'six'   => '6'
];

$collection = new Collection($testArray);

$chunks = $collection->chunk(3);
```

The `$chunks` variable will contain a value similar to the following:

```
object(Illuminate\Support\Collection)[135]
  protected 'items' =>
    array (size=2)
      0 =>
        object(Illuminate\Support\Collection)[133]
          protected 'items' =>
            array (size=3)
              0 => string '1' (length=1)
              1 => string '2' (length=1)
              2 => string '3' (length=1)
      1 =>
        object(Illuminate\Support\Collection)[134]
          protected 'items' =>
            array (size=3)
              0 => string '4' (length=1)
              1 => string '5' (length=1)
              2 => string '6' (length=1)
```

As you can see, the `chunk` method returned a new collection that contains two new collections, each containing three elements. It should also be noted that the original array keys are missing (there are no numerical word names). This can be changed by passing `true` as the second argument:

```
$chunks = $collection->chunk(3, true);
```

Now the `$chunks` variable would have a value similar to the following:

```
object(Illuminate\Support\Collection)[135]
  protected 'items' =>
    array (size=2)
      0 =>
        object(Illuminate\Support\Collection)[133]
          protected 'items' =>
            array (size=3)
              'one'   => string '1' (length=1)
              'two'   => string '2' (length=1)
              'three' => string '3' (length=1)
      1 =>
        object(Illuminate\Support\Collection)[134]
          protected 'items' =>
            array (size=3)
              'four' => string '4' (length=1)
              'five' => string '5' (length=1)
              'six'  => string '6' (length=1)
```

The resulting collections now have the original array keys.

17.4 `only($keys)`

The `only` method is the logical opposite of the `except` method. The `only` method is used to return all the key/value pairs in the collection where the keys in the collection are *in* the supplied `$keys` array. Internally, this method makes a call to the `Illuminate\Support\Arr::only($array, $keys)` helper function.

Using the same example from the `except` method section, we can get only the `first_name` and `last_name` of the users in a collection:

```
// Create a new collection.
$collection = collect([
    'first_name' => 'John',
    'last_name'  => 'Doe',
    'password'   => 'some_password'
]);

// Retrieve only the first and last
// names from the collection.
$data = $collection->only(['first_name', 'last_name']);
```

After the above code has executed, the `$data` variable would contain a value similar to the following output:

```
Collection {
  #items: array:2 [
    "first_name" => "John"
    "last_name"  => "Doe"
  ]
}
```

17.5 `keyBy($keyBy)`

The `keyBy` method is used to create a new `Collection` instance where the keys of the new key/value pairs are determined by a `$keyBy` callback function (a `string` value can also be supplied instead of a function). The signature of the callback function is `keyBy($item)`, where the `$item` is the actual item in the collection. This method does not change the original `Collection` instance will will return a new, modified, `Collection` instance.

The following sample will be used to demonstrate the `keyBy` method:

```
// Create a new collection.
$people = collect([
    ['id' => 12, 'name' => 'Alice', 'age' => 26],
    ['id' => 52, 'name' => 'Bob',   'age' => 34],
    ['id' => 14, 'name' => 'Chris', 'age' => 26]
]);
```

In the following example, we will create a new collection instance where the key of each item is the persons `id`. We will not supply a callback function in this example, but rather pass the name of the value we want to use as the key as as string:

```
$results = $people->keyBy('id');
```

After the above code has executed, the `$results` variable will contain a value similar to the following output:

```
Collection {
  #items: array:3 [
    12 => array:3 [
      "id"   => 12
      "name" => "Alice"
      "age"  => 26
    ]
    52 => array:3 [
      "id"   => 52
      "name" => "Bob"
      "age"  => 34
    ]
    14 => array:3 [
      "id"   => 14
      "name" => "Chris"
      "age"  => 26
    ]
  ]
}
```

We can also achieve the exact same results by using a callback function like so:

```
$results = $people->keyBy(function($item) {
    return $item['id'];
});
```

However, the above example is overly complicated if all we were interested in was just the `id` of the person. We could do something a little more interesting, such as returning a simple hash of each persons `id` to use as the new key:

```
$results = $people->keyBy(function($item) {
    return md5($item['id']);
});
```

After the above code has executed, the new `$results` variable would contain a value similar to the following output:

```
Collection {
  #items: array:3 [
    "c20ad4d76fe97759aa27a0c99bff6710" => array:3 [
      "id"   => 12
      "name" => "Alice"
      "age"  => 26
    ]
    "9a1158154dfa42caddbd0694a4e9bdc8" => array:3 [
      "id"   => 52
      "name" => "Bob"
      "age"  => 34
    ]
    "aab3238922bcc25a6f606eb525ffdc56" => array:3 [
      "id"   => 14
      "name" => "Chris"
      "age"  => 26
    ]
  ]
}
```

The same method could just as easily be applied to other hashing libraries, such as the hashids.php[3] library.

[3]https://github.com/ivanakimov/hashids.php

18. Message Bags

The `Illuminate\Suport\MessageBag` class is an elaborate key/value storage system for storing different types of messages. It it also allows developers to specify a format which is used when returning the messages. The format makes it incredibly simple to format messages on HTML forms, emails, etc. The `MessageBag` class implements both the `Illuminate\Contracts\Support\MessageProvider` and `Illuminate\Contracts\Support\MessageBag` interfaces.

18.1 `Illuminate\Contracts\Support\MessageProvider` Interface

The `MessageProvider` interface defines only one method, `getMessageBag`. The expected behavior of the `getMessageBag` method is to return an instance of an implementation of the `Illuminate\Contracts\Support\MessageBag` interface. Any class that implements the `MessageProvider` interface can be expected to be capable of returning `MessageBag` implementations.

The following classes extend, implement, or make use of the `MessageProvider` interface by default:

Class	Description
`Illuminate\Contracts\Validation\ValidationException`	The `ValidationException` is often thrown when a validation error occurs by some process within an applications request life-cycle.
`Illuminate\Contracts\Validation\Validation`	The `Validator` interface defines the methods and behaviors of the frameworks validator component. The `Validator` component actually extends the `MessageProvider` interface.
`Illuminate\Http\RedirectResponse`	The `RedirectReponse` class is responsible for setting the correct headers required to redirect a user's browser to another URL.
IlluminateSupportMessageBag	The `MessageBag` class is the default implementation of the `Illuminate\Contracts\Support\MessageBag` interface. It also implements the `MessageProvider` interface to return a reference to itself.

Class	Description
`Illuminate\View\View`	The `View` class uses instances of `MessageProvider` implementations to facilitate the communication of errors between the application logic and the front-end code of an application.

18.2 View Error Bags

The `Illuminate\Support\ViewErrorBag` class is generally used to communicate error messages with views and responses. The `ViewErrorBag` itself is essentially a container for `Illuminate\Contracts\Support\MessageBag` implementation instances.

An instance of `ViewErrorBag` is shared with views if the current request contains any errors. This functionality is facilitated by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware. The `ViewErrorBag` instance that is shared with views is given the name `errors`. Interacting with the `ViewErrorBag` instance should look familiar to most developers:

```
@if($errors->count())
<div class="alert alert-danger">
    <p>There were some errors that need to be fixed!</p>
    <ul>
        {!! implode($errors->all('<li>:message</li>')) !!}
    </ul>
</div>
@endif
```

The following examples will assume that a form exists that asks a user for their `username` and a `secret`. The form will be validated from a controller using the following code sample (in this case there is no response when validation was successful):

```
$this->validate($request, [
    'username' => 'required|max:200',
    'secret'   => 'required'
]);
```

If a user supplied only the `username`, the above Blade template code would generate the following HTML:

```
<div class="alert alert-danger">
    <p>There were some errors that need to be fixed!</p>
    <ul>
        <li>The secret field is required.</li>
    </ul>
</div>
```

If the user did not supply either a `username` or a `secret`, the following HTML code would have been generated:

```
<div class="alert alert-danger">
    <p>There were some errors that need to be fixed!</p>
    <ul>
        <li>The username field is required.</li>
        <li>The secret field is required.</li>
    </ul>
</div>
```

18.2.1 `ViewErrorBag` Public API

The `ViewErrorBag` exposes few methods itself in its public API (at the time of writing, only five methods exist intended for public use). However, the `ViewErrorBag` will redirect calls to methods that do not exist explicitly on the `ViewErrorBag` instance to the `default` `MessageBag` instance. Any public method that can be called on a `MessageBag` instance can be called on the `default` `MessageBag` instance without having to do any extra work:

```
use Illuminate\Support\ViewErrorBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Determine if the default MessageBag
// instance is empty, which in this case
// is true.
$viewErrorBag->isEmpty();
```

18.2.1.1 count

The `count` method returns the number of messages stored within the `default` `MessageBag` instance. It is also the only `MessageBag` method that specifically handled by the `ViewErrorBag` instance itself because it needs to be declared within the `ViewErrorBag` class itself to satisfy the requirements of PHP's `Countable`[1] interface.

The following code example demonstrates the behavior of the `count` method:

[1]http://php.net/manual/en/class.countable.php

```php
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

// Add the new MessageBag instance to
// the ViewErrorBag
$viewErrorBag->put('formErrors', $messageBag);

// Get the count from $viewErrorBag
//
// 0
$messageCount = $viewErrorBag->count();

// Change the 'default' MessageBag to the one
// created earlier.
$viewErrorBag->put('default', $messageBag);

// Get the count from $viewErrorBag
//
// 1
$messageCount = count($viewErrorBag);
```

As can be seen in the comments in the above example, the `count` method only operates on the `default` `MessageBag` instance. Because of this adding the `MessageBag` `formErrors` did not affect the count, but setting the same `MessageBag` instance as the value for the `default` key did change the results of the `count` method.

18.2.1.2 `getBag($key = null)`

The `getBag` method can be used to retrieve a `MessageBag` instance associated with the provided `$key`. If a `MessageBag` instance does not exist with the provided `$key`, a new instance of `Illuminate\Support\MessageBag` will returned instead.

The following code sample will demonstrate the usage of the `getBag` method. It also shows that because of the way PHP internally handles objects and references, that the `$messageBag` is the *same* as the value returned from the `getBag` method.

```
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

$viewErrorBag->put('formErrors', $messageBag);

// Demonstrate that the object returned by the getBag
// method is the same as $messageBag.
//
// true
$areTheSame = $messageBag === $viewErrorBag->getBag('formErrors');
```

Additionally, you can request a `MessageBag` instance with any `$key`, even if they have not been set:

```
// Request a MessageBag that has not been set yet.
$messageBagInstance = $viewErrorBag->getBag('paymentErrors');
```

It should be noted that the `getBag` method does not set the `MessageBag` instance that is returning. This behavior can lead to some confusion, and can be observed in the following code sample:

```
$messageBagInstance = $viewErrorBag->getBag('paymentErrors');

// Add a message to the $messageBagInstance
$messageBagInstance->add('ccn', 'The credit card number is invalid');

// Get the number of messages.
//
// 1
$messageCount = $messageBagInstance->count();

// Get the number of messages.
//
// 0
$messageCount = $viewErrorBag->getBag('paymentErrors')->count();
```

If the `ViewErrorBag` instance had set the `MessageBag` instance before returning it from the `getBag` method, both `$messageCount` variables would have contained the value `1`.

Another way to retrieve `MessageBag` instances from the `ViewErrorBag` instance is dynamically access a property, where the property name is the intended key. This technique exhibits the same behavior as using the `getBag` method.

```php
// Get the 'formErrors' MessageBag instance.
$formErrors = $viewErrorBag->formErrors;
```

18.2.1.3 getBags

The `getBags` method is used to return an associative array containing all the `MessageBag` instances that are currently stored within the `ViewErrorBag` instance.

```php
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

// Add some message bags to $viewErrorBag
$viewErrorBag->put('formErrors', $messageBag);
$viewErrorBag->put('paymentErrors', new MessageBag);

// Get the message bags as an array.
$messageBags = $viewErrorBag->getBags();
```

After the above code has executed, the `$messageBags` variable would be an array and contain a value similar to the following output:

```
array (size=2)
  'formErrors' =>
    object(Illuminate\Support\MessageBag)[142]
      protected 'messages' =>
        array (size=1)
          'username' =>
            array (size=1)
              ...
      protected 'format' => string ':message' (length=8)
  'paymentErrors' =>
    object(Illuminate\Support\MessageBag)[143]
      protected 'messages' =>
        array (size=0)
          empty
      protected 'format' => string ':message' (length=8)
```

18.2.1.4 `hasBag($key = 'default')`

The `hasBag` method is used to determine if a `MessageBag` instance exists within the `ViewErrorBag` instance with the given `$key`. The `$key` is set to `default` unless it is changed. The following example will highlight the usage of the `hasBag` method.

```php
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Check if the 'default' MessageBag instance
// exists.
//
// false
$doesExist = $viewErrorBag->hasBag();
$doesExist = $viewErrorBag->hasBag('default');

// Check if a 'payments' MessageBag instance
// exists.
//
// false
$doesExist = $viewErrorBag->hasBag('payments');

// Add a new 'payments' MessageBag instance.
$viewErrorBag->put('payments', new MessageBag);

// Check again if the 'payments' MessageBag instance
// exists.
```

```
//
// true
$doesExist = $viewErrorBag->hasBag('payments');
```

18.2.1.5 `put($key, Illuminate\Contracts\Support\MessageBag $bag)`

The put method is used to add a new `MessageBag` implementation instance to the `ViewErrorBag` instance, supplied as the argument to the `$bag` parameter with some name determined by the `$key` argument. The following code demonstrates how to add a new `MessageBag` instance to a `ViewErrorBag`:

```
use Illuminate\Support\ViewErrorBag;
use Illuminate\Support\MessageBag;

// Create a new ViewErrorBag instance.
$viewErrorBag = new ViewErrorBag;

// Create a new MessageBag instance.
$messageBag = new MessageBag;

// Add the $messageBag to the $viewErrorBag
// with some key.
$viewErrorBag->put('somekey', $messageBag);

// Get the number of MessageBag instances.
$messageBagCount = count($viewErrorBag->getBags());
```

After the above code has executed, the `$messageBagCount` variable would contain the value `1`.

Another way to add `MessageBag` instances to the `ViewErrorBag` is by dynamically setting an instance property:

```
// Add a new MessageBag instance without
// using the 'put' method.
$viewErrorBag->anotherKey = new MessageBag;
```

At this point, the `$viewErrorBag` instance would now contain two `MessageBag` instances with the keys `somekey` and `anotherKey`.

18.2.1.5.1 Resetting `MessageBag` Messages

Because neither `ViewErrorBag` or `MessageBag` provide a method to quickly remove all the messages from a `MessageBag` instance, the put method can be used to remove all `MessageBag` messages, or supply new ones, by changing the `MessageBag` instance for a given key:

```php
// Create a different MessageBag instance.
$newMessageBag = new MessageBag([
    'username' => [
        'The username is required.'
    ]
]);

// Get the 'somekey' MessageBag before changing it.
$beforeChange = $viewErrorBag->getBag('somekey');

// Change the MessageBag instance.
$viewErrorBag->put('somekey', $newMessageBag);

// Get the 'somekey' MessageBag after changing it.
$afterChange = $viewErrorBag->getBag('somekey');
```

After the above code has executed, the `$beforeChange` variable will contain the old `MessageBag` instance and the `$afterChange` variable will contain the new `MessageBag` instance.

$beforeChange MessageBag Output

```
object(Illuminate\Support\MessageBag)[142]
  protected 'messages' =>
    array (size=0)
      empty
  protected 'format' => string ':message' (length=8)
```

$afterChange MessageBag Output

```
object(Illuminate\Support\MessageBag)[143]
  protected 'messages' =>
    array (size=1)
      'username' =>
        array (size=1)
          0 => string 'The username is required.' (length=25)
  protected 'format' => string ':message' (length=8)
```

19. Fluent

The `Illuminate\Support\Fluent` class is a useful data type. It allows for the construction of a data “container” similar to an array or instance of `stdClass`. However, the `Fluent` class makes it easier to make assumptions about the data the class instance contains. The following array and `stdClass` instance will be used for the next couple of examples:

```php
// Create a new array containing sample data
$testArray = [
    'first'  => 'The first value',
    'second' => 'The second value',
    'third'  => 'The third value'
];

// Create a new stdClass instance containing sample data
$testObject = new stdClass;
$testObject->first  = 'The first value';
$testObject->second = 'The second value';
$testObject->third  = 'The third value';
```

We can access data from the `$testArray` like so:

```php
// Retrieve the 'first' item
$value = $testArray['first'];
```

The `$value` variable would now contain the value `The first value`. Similarly, data can be retrieved from the `stdClass` instance using object operator (often called the “arrow”):

```php
// Retrieve the 'first' property
$value = $testObject->first;
```

Like in the previous example, the `$value` variable would now contain the value `The first value`. There is nothing surprising going on here. However, what happens when a developer needs to make assumptions about the data their code is working with? Uncertain developers often litter code with unwieldy `if` statements and similar constructs. Failure to do so generally results in fatal errors.

For example, attempting to retrieve data from an array that does not exist results in an instance of `ErrorException` being thrown:

```
// Will raise an exception
$value = $testArray['does_not_exist'];
```

The exact error message will differ between single or multi-dimensional arrays, but the principal is the same. PHP does not like it when code attempts to access array elements that do not exist.

The same can be said for accessing an object's properties:

```
// Will raise an exception
$value = $testObject->doesNotExist;
```

The above code example will again throw an instance of `ErrorException`, with the error message being something similar to "Undefined property: stdClass::$doesNotExit". To work around this, the following code can be written:

```
// Get a value from an array, or a default value
// if it does not exist.

if (array_key_exists('does_not_exist', $testArray))
{
    $value = $testArray['does_not_exist'];
} else {
    $value = 'Some default value';
}


// Get a value from an object, or a default value
// if it does not exist.

if (property_exists('doesNotExist', $testObject))
{
    $objectValue = $testObject->doesNotExist;
} else {
    $objectValue = 'Some default value';
}
```

The above code example can be simplified using Laravel's array and object helper functions. Specifically see the sections on `array_get`, `object_get` and `data_get`.

In the above code example, we checked to see if an object instance has a value by using the `property_exists`[a] function instead of the `isset`[b] function. This is because the `property_-exist` function will return `true` if the property exists and has a value of `null`. The `isset` function will return `false` if the property exists but has a value of `null`.

[a]http://php.net/manual/en/function.property-exists.php

[b]http://php.net/manual/en/function.isset.php

Developers need to assume things about code quite often. In a perfect world, developers would know exactly what data an array or object their code interacts with contains. However, when dealing with remote data, such as data from external APIs, or when interfacing with code from multiple development teams, this is not always possible. The `Fluent` class can be used to simplify things for developers.

The following example will create a new `Fluent` instance with some data:

```
// Some example data, which could be obtained from
// any number of sources.
$testArray = [
    'first'  => 'The first value',
    'second' => 'The second value',
    'third'  => 'The third value'
];

// Create a new Fluent instance.
$fluent = new Fluent($testArray);
```

Now that we have a `Fluent` instance, data can be retrieved just like an object or an array:

```
// Accessing a value like an array.
$value = $fluent['first'];

// Accessing a value like an object.
$secondValue = $fluent->first;
```

Both `$value` and `$secondValue` would contain the value `The first value`. Accessing data that does not exist now simply returns `null`, without raising an error:

```
$value = $fluent['does_not_exist'];

$secondValue = $fluent->doesNotExist;
```

Both `$value` and `$secondValue` would contain the value `null`. The `Fluent` class does expose public methods in its API to custom the default value returned.

19.1 `Fluent` Public API

The following sections will highlight the usage of the various public methods that the `Fluent` class provides. There are, however, some omissions in this section, namely the methods required to implement PHP's `ArrayAccess`[1] interface. The examples that follow will assume the following test array and object, unless stated otherwise:

[1]http://php.net/manual/en/class.arrayaccess.php

```php
// A test array for use with Fluent.
$testArray = [
    'first'  => 'The first value',
    'second' => 'The second value',
    'third'  => 'The third value'
];

// A test object for use with Fluent.
$testObject = new stdClass;
$testObject->first  = 'The first value';
$testObject->second = 'The second value';
$testObject->third  = 'The third value';
```

19.1.1 `get($key, $default = null)`

The `get` method will return the value associated with the provided `$key`. If the `$key` does not exist, the `$default` value will be returned (which is `null` by default).

Retrieving values from a Fluent instance:

```php
$fluent = new Fluent($testArray);

// The first value
$message = $fluent->get('first');

$fluent = new Fluent($testObject);

// The first value
$message = $fluent->get('first');

// null
$willBeNull = $fluent->get('does_not_exist');
```

The `$default` value is evaluated using the `value` helper function, meaning that it can be the result of a function:

```php
$fluent = new Fluent($testArray);

// Does not exist yet!
$message = $fluent->get('does_not_exist', function() {
    return 'Does not exist yet!';
});
```

19.1.1.1 Fluent and Closures

If we look at the following code example, one might be tempted to say that the value of `$message` would be `Hello, world!`, but that would be incorrect:

```
$testObject = new stdClass;
$testObject->method = function() {
    return 'Hello, world!';
};

$fluent = new Fluent($testObject);

$message = $fluent->method;

// Or even this:

$message = $fluent->get('method');
```

However, that would be incorrect. It is important to remember that the `Fluent` object is an elaborate key/value storage container that can behave like an array or an object. When the fluent container is created for an object containing a closure, such as the above example, the closure instance is stored as the value. The following code will quickly prove this:

```
// true
$isClosure = ($fluent->method instanceof Closure):
```

To evaluate the closure and get the results, the `value` helper function can be used:

```
// Hello, world!
$message = value($fluent->method);

// Hello, world!
$message = value($fluent->get('method'));
```

19.1.2 `getAttributes()`

The `getAttributes` method simply returns an array containing all key/value pairs, representing the underlying data contained within the `Fluent` instance.

After the following code is executed:

```
$fluent = new Fluent($testObject);

$attributes = $fluent->getAttributes();
```

The `$attributes` variable would look have a value similar to the following:

```
array (size=4)
  'first'  => string 'The first value' (length=15)
  'second' => string 'The second value' (length=16)
  'third'  => string 'The third value' (length=15)
```

19.1.3 `toArray()`

The `toArray` method returns the exact same values as the `getAttributes` method.

19.1.4 `jsonSerialize()`

The `jsonSerialize` method internally returns a call to `toArray`. Because of this, `toArray`, `jsonSerialize` and `getAttributes` are all functionally equivalent. The `jsonSerialize` method exists to satisfy PHP's `JsonSerializable`[2] interface, which allows developers to customize how a class is represented when using the `json_encode` function.

19.1.5 `toJson($options = 0)`

The `toJson` method will return a JSON encoded version of the data stored within the fluent instance. It internally does this by returning a call to PHP's `json_encode`[3] function, passing in any `$options` that were supplied. Like the `json_encode` function, the `$options` parameter is a bitmask of the predefined JSON constants[4].

The following code:

```
$fluent = new Fluent($testObject);

$jsonValue = $fluent->toJson();
```

would be converted into the following JSON, stored in the `$jsonValue` variable:

```
{"first":"The first value","second":"The second value","third":
  "The third value","method":{}}
```

Alternatively, a well-formatted value can be returned by passing in the `JSON_PRETTY_PRINT` constant:

```
$fluent = new Fluent($testObject);

$jsonValue = $fluent->toJson(JSON_PRETTY_PRINT);
```

This time, the `$jsonValue` would contain the following value:

[2]http://php.net/manual/en/class.jsonserializable.php
[3]http://php.net/manual/en/function.json-encode.php
[4]http://php.net/manual/en/json.constants.php

```
{
    "first": "The first value",
    "second": "The second value",
    "third": "The third value",
    "method": {}
}
```

19.1.5.1 `toJson` and Deeply Nested Data Structures

The `toJson` method internally makes a call to PHP's `json_encode` function. Unlike `json_-encode`, `toJson` does not provide a way to specify the depth (essentially how many arrays are nested inside of each other) to which data will be encoded, which is by default set to `512`. To convert a fluent object into its JSON equivalent with a depth greater than `512`, the following method will be sufficient:

```
$fluent = new Fluent($testObject);

// Replace 512 with the desired depth.
$jsonValue = json_encode($fluent->jsonSerialize(), 0, 512);
```

20. Facades

Facades are a convenient way to access Laravel's components. Each facade is bound to some component already registered in the service container. Facades are typically used to provide a simpler interface to an complex underlying subsystem. While some facades provide "shortcut" methods, most of Laravel's facades act like proxies by providing a static interface to an actual class instance.

All of the default facades are located within the `Illuminate\Support\Facades` namespace and can be located within the `vendor/laravel/framework/src/Illuminate/Support/Facades` directory. Every facade extends the abstract `Facade` (`Illuminate\Support\Facades\Facade`) class and *must* provide a `getFacadeAccessor()` method which is defined to be *protected*.

The `getFacadeAccessor()` method indicates what class or component should be resolved from the service container when the facade is accessed. All method calls are then redirected to that class instance[1]. Laravel refers to this class instance as the *facade root*.

Facade roots are the class instance that is bound in the service container. For example, the `Auth` facade redirects all method calls to an instance of `Illuminate\Auth\AuthManager`, which the service container resolves with the `auth` binding.

20.1 Facade Aliases and Importing Facades

Each facade has an *alias*. An alias in PHP is just a shortcut to a longer class name. For example, when we use something *as* something else, we are creating a class alias:

```php
use Some\ReallyReallyLongClassName as ShorterName;
```

In the above example `ShorterName` is an alias of `ReallyReallyLongClassName`. Laravel creates aliases for all of the facades automatically, and the entire list of them can be found in the `aliases` array in the `config/app.php` file. The aliases that Laravel creates are in the global namespace, which means that developers can write code that looks like this:

```php
<?php namespace App\Http\Controllers;

use Input;

// Other code here.
```

instead of having to write code that references the full namespace:

[1]Some facades define additional static methods that may are not necessarily part of the underlying component or class. In these instances, the static method (that belongs to the facade) is called directly. One such example is the `Cookie` facade.

```php
<?php namespace App\Http\Controllers;

use Illuminate\Support\Facades\Input;

// Other code here.
```

Fully Qualified Names vs. Aliases

Fully qualified names, such as `Illuminate\Support\Facades\Input` are longer, but offer greater clarity on what class is being imported. At the end of the day, it comes down to personal preference.

20.2 Using Facades

Laravel's facades are used just like any other static class, with the exception that facades are redirecting method calls to an actual class instance. Facades are typically imported at the top of the source file, along with any other classes that are required.

For example, using a facade to retrieve an item from the cache:

```php
<?php namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class ExampleController extends Controller {

    public function getIndex()
    {
        $cachedItem = Cache::get('some_cache_item');
    }

}
```

The alternative[2] is to *inject* the cache repository directly into the controller:

[2]Dependency injection is not the only alternative to using facades, it is, however, arguably the most common in Laravel projects.

```php
<?php namespace App\Http\Controllers;

use Illuminate\Cache\Repository as CacheRepository;

class ExampleController extends Controller {

    /**
     * The cache repository.

     * @var CacheRepository
     */
    protected $cacheRepository = null;

    public function __construct(CacheRepository $cacheRepository)
    {
        $this->cacheRepository = $cacheRepository;
    }

    public function getIndex()
    {
        $cachedItem = $this->cacheRepository->get('some_cache_item');
    }

}
```

Both code examples would accomplish the same thing: retrieve `some_cache_item` from the cache storage. The facade example allows for shorter code, and code that is definitely easier to follow along with. The second example has its own advantages too, which will not be covered in this section.

20.3 Creating Facades

It may become necessary when developing certain applications, or when creating packages to *write* a facade class. Creating a facade class is fairly simple. The first thing that is required is some actual concrete class the facade will be accessing. For this example, the following class will be used:

```php
<?php

class Calculator {

    /**
     * Adds two numbers.

     * @param  mixed $firstNumber
     * @param  mixed $secondNumber
     * @return mixed
     */
    public function add($firstNumber, $secondNumber)
    {
        return $firstNumber + $secondNumber;
    }

    // Possibly many more methods.

}
```

The next thing that needs to happen is the class needs to be registered with the service container:

```php
App::singleton('math', function()
{
    return new Calculator;
});
```

Components are typically registered with the service container through the use of Service Providers.

The `Calculator` class instance was bound as a singleton because there does not need to possibly hundreds of instances of `Calculator` created: many consumers of the class can share a single instance (note that not all classes should be bound as a singleton). The important part is that we have an service container binding: `math` which will resolve to an instance of `Calculator`.

It is at this point a facade can be created. A facade is created by creating a new class and extending Laravel's abstract `Facade` class:

```php
<?php

use Illuminate\Support\Facades\Facade;

class Math extends Facade {

    /**
     * Get the registered name of the component.
     *
     * @return string
     */
    protected static function getFacadeAccessor() { return 'math'; }

}
```

It is important to note that the value returned by the `getFacadeAccessor()` function matches the name of the service container binding: `math`. The facade will use that value to request a class instance from the service container and redirect method calls to that instance.

We will now examine how we would have used the `Calculator` class without the facade:

```php
<?php namespace App\Http\Controllers;

use Calculator;

class TestController extends Controller {

    public function getIndex()
    {
        // We will create an instance, instead of having it supplied
        // through the constructor.
        $calculator = new Calculator;

        $result = $calculator->add(1, 2);
    }

}
```

now using the facade:

```php
<?php namespace App\Http\Controllers;

use Math;

class TestController extends Controller {

    public function getindex()
    {
        $result = Math::add(1, 2);
    }

}
```

20.3.1 Creating a Facade Alias

Creating a facade alias is completely optional. Package developers often include the following steps in their package installation instructions, but it is not required for a facade to work.

In the `config/app.php` file, there is an `aliases` configuration entry. By default it will look like this:

```php
// Other configuration items.

'aliases' => [

    'App'       => 'Illuminate\Support\Facades\App',
    'Artisan'   => 'Illuminate\Support\Facades\Artisan',
    'Auth'      => 'Illuminate\Support\Facades\Auth',
    'Blade'     => 'Illuminate\Support\Facades\Blade',
    'Bus'       => 'Illuminate\Support\Facades\Bus',
    'Cache'     => 'Illuminate\Support\Facades\Cache',
    'Config'    => 'Illuminate\Support\Facades\Config',
    'Cookie'    => 'Illuminate\Support\Facades\Cookie',
    'Crypt'     => 'Illuminate\Support\Facades\Crypt',
    'DB'        => 'Illuminate\Support\Facades\DB',
    'Event'     => 'Illuminate\Support\Facades\Event',
    'File'      => 'Illuminate\Support\Facades\File',
    'Gate'      => 'Illuminate\Support\Facades\Gate',
    'Hash'      => 'Illuminate\Support\Facades\Hash',
    'Input'     => 'Illuminate\Support\Facades\Input',
    'Lang'      => 'Illuminate\Support\Facades\Lang',
    'Log'       => 'Illuminate\Support\Facades\Log',
    'Mail'      => 'Illuminate\Support\Facades\Mail',
    'Password'  => 'Illuminate\Support\Facades\Password',
    'Queue'     => 'Illuminate\Support\Facades\Queue',
    'Redirect'  => 'Illuminate\Support\Facades\Redirect',
    'Redis'     => 'Illuminate\Support\Facades\Redis',
    'Request'   => 'Illuminate\Support\Facades\Request',
    'Response'  => 'Illuminate\Support\Facades\Response',
    'Route'     => 'Illuminate\Support\Facades\Route',
    'Schema'    => 'Illuminate\Support\Facades\Schema',
    'Session'   => 'Illuminate\Support\Facades\Session',
    'Storage'   => 'Illuminate\Support\Facades\Storage',
    'URL'       => 'Illuminate\Support\Facades\URL',
    'Validator' => 'Illuminate\Support\Facades\Validator',
    'View'      => 'Illuminate\Support\Facades\View',
],

// Other configuration items.
```

Examining the above code sample, it can be deduced that to add a new facade alias we simply need to add a new entry to the `aliases` array. The key of the entry will become the name of the alias and the value of the entry will become the class being aliased. To make this clearer, in the above list `Response` is aliasing `Illuminate\Support\Facades\Response`, and both classes can be used interchangeably.

We will use our `Math` facade from earlier, and we will assume it was defined in the `Our\Applications\Namespace`. We could create an alias like so:

```
    // Previous alias entries.

    'Math' => 'Our\Applications\Namespace\Math',
```

Aliasing Other Classes

Classes, other than facades, can be added to the `aliases` configuration entry. This will cause them to be available under whatever name was provided, and in the global namespace. Although this can be convenient and useful thing to do, other options should be examined first.

20.4 Facade Class Reference

The following tables will list all the facades that are available by default. In addition, they will display the name of the service container binding as well as the class behind the facade. If a particular facade provides additional methods (that are not necessarily available in the underlying class), it will appear in the third table “Facades Providing Additional Methods”. Any additional methods will be explained later in the section.

Facade Service Container Binding

Facade	Service Container Binding
App	`app`
Artisan	`Illuminate\Contracts\Console\Kernel`
Auth	`auth`
Blade	See “Notes on Blade”
Bus	`Illuminate\Contracts\Bus\Dispatcher`
Cache	`cache`
Config	`config`
Cookie	`cookie`
Crypt	`encrypter`
DB	`db`
Event	`events`
File	`files`
Gate	`Illuminate\Contracts\Auth\Access\Gate`
Hash	`hash`
Input	`request`
Lang	`translator`
Log	`log`
Mail	`mailer`
Password	`auth.password`
Queue	`queue`
Redirect	`redirect`
Redis	`redis`
Request	`request`
Response	`Illuminate\Contracts\Routing\ResponseFactory`
Route	`router`
Schema	See “Notes on Schema”
Session	`session`
Storage	`filesystem`
URL	`url`
Validator	`validator`
View	`view`

Facade Class Reference

Facade	Resolved Class
App	`Illuminate\Foundation\Application`
Artisan	`App\Console\Kernel`
Auth	`Illuminate\Auth\AuthManager`
Blade	`Illuminate\View\Compilers\BladeCompiler`
Bus	`Illuminate\Bus\Dispatcher`
Cache	`Illuminate\Cache\CacheManager`
Config	`Illuminate\Config\Repository`
Cookie	`Illuminate\Cookie\CookieJar`
Crypt	`Illuminate\Encryption\Encrypter`
DB	`Illuminate\Database\DatabaseManager`

Facade Class Reference

Facade	Resolved Class
Event	`Illuminate\Events\Dispatcher`
File	`Illuminate\Filesystem\Filesystem`
Gate	`Illuminate\Auth\Access\Gate`
Hash	`Illuminate\Hashing\BcryptHasher`
Input	`Illuminate\Http\Request`
Lang	`Illuminate\Translation\Translator`
Log	`Illuminate\Log\Writer`
Mail	`Illuminate\Mail\Mailer`
Password	`Illuminate\Auth\Passwords\PasswordBroker`
Queue	`Illuminate\Queue\QueueManager`
Redirect	`Illuminate\Routing\Redirector`
Redis	`Illuminate\Redis\Database`
Request	`Illuminate\Http\Request`
Response	`Illuminate\Routing\ResponseFactory`
Route	`Illuminate\Routing\Router`
Schema	`Illuminate\Database\Schema\MySqlBuilder`
Session	`Illuminate\Session\SessionManager`
Storage	`Illuminate\Contracts\Filesystem\Factory`
URL	`Illuminate\Routing\UrlGenerator`
Validator	`Illuminate\Validator\Factory`
View	`Illuminate\View\Factory`

Facades Providing Additional Methods

Facade	Number of Additional Methods
Cookie	2
Input	2
Schema	2

20.4.1 Notes on `Blade`

Most facades request a concrete class implementation from the service container based off of some abstract string representation. However, the `Blade` facade retrieve an `Illuminate\View\Engines\EngineResolver` instance from the `Illuminate\View\Factory`.

The `Illuminate\View\Engines\EngineResolver` is a class that returns template compilers based on a given key name. By default, the following compilers and engines are available:

Compiler/Engine Name	Concrete Class Implementation
`php`	`Illuminate\View\Engines\PhpEngine`
`blade`	`Illuminate\View\Compilers\BladeCompiler`

Developers can manually create an instance of the `BladeCompiler` themselves like so (this sample is provided for demonstration purposes):

```php
<?php

use Illuminate\Support\Facades\App;

$bladeCompiler = App::make('view')->getEngineResolver()
                  ->resolve('blade')->getCompiler();
```

20.4.2 Notes on `Schema`

Like the `Blade` facade, the `Schema` facade does not simply resolve some instance from the service container. The `Schema` facade returns an instance of `Illuminate\Database\Schema\Builder` configured to use the default connection that appears in the database database configuration.

20.4.3 Additional `Cookie` Methods

The `Cookie` facade defines two additional methods. These methods access other, related, components. These methods exist to simplify accessing related components.

20.4.3.1 `has($key)`

The `has` function will check if a cookie with the given `$key` exists for the current request.

20.4.3.2 `get($key, $default = null)`

The `get` function will retrieve a cookie from the current request with the given `$key`. A `$default` value can be supplied and will be returned if a cookie with the given `$key` does not exist.

20.4.4 Additional `Input` Methods

The `Input` facade defines one extra method. Facades define extra methods to provide simpler access to underlying sub-systems, or to call functions on other, related components.

20.4.4.1 `get($key = null, $default = null)`

The `get` method will *get* an item from the input data, such as when a user posts a form or an API request is being processed. The `get` method can be used for requests with the following HTTP verbs:

- GET
- POST
- PUT

- DELETE

> The `get` method will invoke the `input` method on an instance of the `Illuminate\Http\Request` class.

The `get` method looks up the data based on the given `$key`. A `$default` value can be supplied and will be returned if the given `$key` is not found in the request data.

20.4.5 Additional `Schema` Methods

The `Schema` facade defines one extra method. Facades define extra methods to provide simpler access to underlying sub-systems, or to call functions on other, related components.

20.4.5.1 `connection($name)`

The `connection` method will return a new schema builder (`Illuminate\Database\Schema\Builder`) instance for the given connection. The `$name` is the name of the connection as it appears in the database configuration file.

20.5 Resolving the Class Behind a Facade

It is possible to quickly resolve the class behind a facade. Facades expose a public method `getFacadeRoot` which will return the instance of the underlying object the facade is forwarding method calls to. It is convenient that `getFacadeRoot` returns an object instance because PHP's `get_class`[3] method can then be used to retrieve the fully-qualified name of the facade's underlying class implementation.

```
// Getting the class name of the underlying facade instance.
$className = get_class(Auth::getFacadeRoot());
```

In the above code example, the `$className` variable would contain the value `Illuminate\Auth \AuthManager`.

This method of determining a facade's underlying class can be expanded on to create a function that will list every facade's underlying class for the current Laravel installation:

[3]http://php.net/manual/en/function.get-class.php

```
/**
 * Generates an HTML table containing all registered
 * facades and the underlying class instances.
 *
 * @return string
 */
function getFacadeReferenceTable()
{
    $html = '';

    // An array of all the facades that should be printed in the table.
    $facades = [
        'App',  'Artisan', 'Auth', 'Blade', 'Bus',
        'Cache', 'Config', 'Cookie', 'Crypt', 'DB',
        'Event', 'File', 'Hash', 'Input', 'Lang', 'Log',
        'Mail', 'Password', 'Queue', 'Redis', 'Redirect',
        'Request', 'Response', 'Route', 'Schema', 'Session',
        'Storage', 'URL', 'Validator', 'View'
    ];

    // Boilerplate HTML to open an HTML table.
    $html =  '<table><thead><tr><th>Facade</th>';
    $html .= '<th>Underlying Class</th></tr><tbody>';

    foreach ($facades as $facade)
    {
        $html .= '<tr><td>',$facade,'</td><td>';
        $html .= get_class(call_user_func($facade.'::getFacadeRoot'));
        $html .= '</td></tr>';
    }

    // Boilerplate HTML to close an HTML table.
    $html .= '</tbody></table>';

    return $html;
}
```

Appendix A: Available Hash Functions

The chapter on hashing discussed creating implementations of `Illuminate\Contracts\Hashing\Hashser` for the various hash functions available for PHP's `crypt`[4] function. However, there are many more hash functions available through the use of PHP's `hash`[5] function. The following table lists the hash functions that are available on a default Homestead environment. These functions were discussed using PHP's `hash_algos`[6] function.

Homestead Hash Functions

md2	ripemd320	gost-crypto	haval224,3
md4	whirlpool	adler32	haval256,3
md5	tiger128,3	crc32	haval128,4
sha1	tiger160,3	crc32b	haval160,4
sha224	tiger192,3	fnv132	haval192,4
sha256	tiger128,4	fnvla332	haval224,4
sha384	tiger160,4	fnv164	haval256,4
sha512	tiger192,4	fnvla64	haval160,5
ripemd128	snefru	joaat	haval192,5
ripemd160	snefru256	haval128,3	haval224,5
ripemd256	gost	haval160,3	haval256,5

[4]http://php.net/manual/en/function.crypt.php
[5]http://php.net/manual/en/function.hash.php
[6]http://php.net/manual/en/function.hash-algos.php

Appendix B: HTTP Status Codes

RFC 2295[7], RFC 4918[8], RFC 5842[9], RFC 6585[10], RFC 7231[11], RFC 7232[12], RFC 7233[13], RFC 7235[14], RFC 7538[15], RFC 7540[16] define the HTTP status codes listed below. The status codes are listed below for convenience.

Code	Full Name
100	100 Continue
101	101 Switching Protocols
200	200 OK
201	201 Created
202	202 Accepted
203	203 Non-Authoritative Information
204	204 No Content
205	205 Reset Content
206	206 Partial Content
300	300 Multiple Choices
301	301 Moved Permanently
302	302 Found
303	303 See Other
304	304 Not Modified
305	305 Use Proxy
306	306 (Unused)
307	307 Temporary Redirect
308	308 Permanent Redirect
400	400 Bad Request
401	401 Unauthorized
403	403 Forbidden
404	404 Not Found
405	405 Method Not Allowed
406	406 Not Acceptable
407	407 Proxy Authentication Required
408	408 Request Timeout
409	409 Conflict
410	410 Gone
411	411 Length Required

[7]https://www.ietf.org/rfc/rfc2295.txt
[8]https://tools.ietf.org/html/rfc4918
[9]https://www.ietf.org/rfc/rfc2774.txt
[10]https://tools.ietf.org/html/rfc6585
[11]https://tools.ietf.org/html/rfc7231
[12]https://tools.ietf.org/html/rfc7232
[13]https://tools.ietf.org/html/rfc7233
[14]https://tools.ietf.org/html/rfc7235
[15]https://tools.ietf.org/html/rfc7538
[16]https://tools.ietf.org/html/rfc7540

Code	Full Name
412	412 Recondition Failed
413	413 Payload Too Large
414	414 URI Too Long
415	415 Unsupported Media Type
416	416 Range Not Satisfiable
417	417 Expectation Failed
421	421 Misdirect Request
422	422 Unprocessable Entity
423	423 Locked
426	426 Upgrade Required
428	428 Precondition Required
429	429 Too many Requests
451	451 Unavailable for Legal Reasons
500	500 Internal Server Error
501	501 Not Implemented
502	502 Bad Gateway
503	503 Service Unavailable
504	504 Gateway Timeout
505	505 HTTP Version Not Supported
506	506 Variant Also negotiates
507	507 Insufficient Storage
508	508 Loop Detected
510	510 Not Extended
511	511 Network Authentication Required