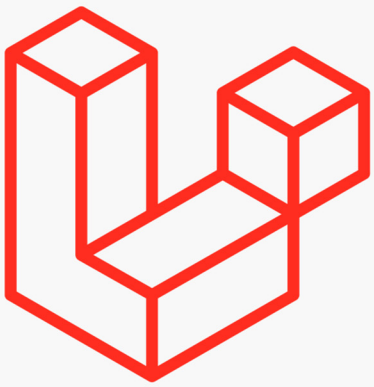

Laravel starters book

auto generated from the official docs

Table of Contents

Welcome 10

..... 11

Installation 11

- Meet Laravel 12
- Your First Laravel Project 14
- Initial Configuration 20
- Next Steps 21

Authentication 22

- Introduction 23
- Authentication Quickstart 28
- Manually Authenticating Users 32
- HTTP Basic Authentication 38
- Logging Out 40
- Password Confirmation 42
- Adding Custom Guards 45
- Adding Custom User Providers 48
- Events 52

Authorization 53

- Introduction 54
- Gates 55
- Creating Policies 62
- Writing Policies 65
- Authorizing Actions Using Policies 70

Blade Templates 78

- Introduction 80
- Displaying Data 81
- Blade Directives 85

Components 95
Building Layouts 116
Forms 120
Stacks 122
Service Injection 123
Extending Blade 124
Cache 127
Introduction 128
Configuration 129
Cache Usage 132
Cache Tags 138
Atomic Locks 140
Adding Custom Cache Drivers 143
Events 146
Collections 146
Introduction 148
Available Methods 150
Method Listing 151
Higher Order Messages 232
Lazy Collections 233
Configuration 237
Introduction 239
Environment Configuration 240
Accessing Configuration Values 243
Configuration Caching 244
Debug Mode 245
Maintenance Mode 246
Controllers 248
Introduction 249
Writing Controllers 250

Controller Middleware 253
Resource Controllers 254
Dependency Injection & Controllers 262
Database: Getting Started 264
Introduction 265
Running SQL Queries 269
Database Transactions 274
Connecting To The Database CLI 276
Eloquent: Collections 276
Introduction 277
Available Methods 278
Custom Collections 282
Eloquent: Mutators & Casting 282
Introduction 284
Accessors & Mutators 285
Attribute Casting 288
Custom Casts 297
Eloquent: Relationships 307
Introduction 308
Defining Relationships 309
Many To Many Relationships 324
Polymorphic Relationships 332
Querying Relations 344
Aggregating Related Models 351
Eager Loading 356
Inserting & Updating Related Models 366
Touching Parent Timestamps 372
Eloquent: Getting Started 373
Introduction 374
Generating Model Classes 375

Eloquent Model Conventions 377
Retrieving Models 384
Retrieving Single Models / Aggregates 390
Inserting & Updating Models 393
Deleting Models 401
Pruning Models 406
Replicating Models 410
Query Scopes 411
Comparing Models 419
Events 420
Encryption 426
Introduction 427
Configuration 428
Using The Encrypter 429
Error Handling 430
Introduction 431
Configuration 432
The Exception Handler 433
HTTP Exceptions 440
File Storage 441
Introduction 442
Configuration 443
Obtaining Disk Instances 447
Retrieving Files 448
Storing Files 452
Deleting Files 458
Directories 459
Custom Filesystems 461
Hashing 463
Introduction 464

Configuration 465
Basic Usage 466
Helpers 468
Introduction 469
Available Methods 470
Method Listing 472
Arrays & Objects 473
Paths 494
Strings 497
Fluent Strings 518
URLs 542
Miscellaneous 545
HTTP Client 561
Introduction 562
Making Requests 563
Concurrent Requests 571
Testing 572
Events 577
Mail 577
Introduction 579
Generating Mailables 584
Writing Mailables 585
Markdown Mailables 595
Sending Mail 599
Rendering Mailables 604
Localizing Mailables 605
Testing Mailables 607
Mail & Local Development 608
Events 609
Database: Pagination 609

Introduction 610
Basic Usage 611
Displaying Pagination Results 617
Customizing The Pagination View 619
Paginator / LengthAwarePaginator Instance Methods 622
Cursor Paginator Instance Methods 624
Database: Query Builder 625
Introduction 626
Running Database Queries 627
Select Statements 633
Raw Expressions 634
Joins 637
Unions 640
Basic Where Clauses 641
Ordering, Grouping, Limit & Offset 650
Conditional Clauses 654
Insert Statements 655
Update Statements 657
Delete Statements 659
Pessimistic Locking 660
Debugging 661
HTTP Requests 661
Introduction 662
Interacting With The Request 663
Input 670
Input Trimming & Normalization 678
Files 679
Configuring Trusted Proxies 681
Configuring Trusted Hosts 684
HTTP Responses 684

Creating Responses 686
Redirects 691
Other Response Types 695
Response Macros 698
Routing 699
Basic Routing 700
Route Parameters 704
Named Routes 708
Route Groups 711
Route Model Binding 713
Fallback Routes 720
Rate Limiting 721
Form Method Spoofing 725
Accessing The Current Route 726
Cross-Origin Resource Sharing (CORS) 727
Route Caching 728
HTTP Session 728
Introduction 729
Interacting With The Session 731
Session Blocking 737
Adding Custom Session Drivers 739
Directory Structure 741
Introduction 743
The Root Directory 744
The App Directory 747
URL Generation 749
Introduction 750
The Basics 751
URLs For Named Routes 753
URLs For Controller Actions 757

Default Values 758
Validation 759
Introduction 761
Validation Quickstart 762
Form Request Validation 769
Manually Creating Validators 776
Working With Validated Input 782
Working With Error Messages 784
Available Validation Rules 788
Conditionally Adding Rules 808
Validating Arrays 812
Validating Passwords 815
Custom Validation Rules 818
Views 824
Introduction 825
Creating & Rendering Views 826
Passing Data To Views 828
View Composers 830
Optimizing Views 834

Welcome

This book is automatically created from the markdown files of the official Laravel Documentation (laravel/docs).
This first book gives you a base overview of what Laravel is and what the base concepts of Laravel are.

The second book Laravel-advanced is the more advanced part of the documentation and is for more advanced usage.

Installation

- Meet Laravel
    - Why Laravel?
- Your First Laravel Project
    - Getting Started On macOS
    - Getting Started On Windows
    - Getting Started On Linux
    - Choosing Your Sail Services
    - Installation Via Composer
- Initial Configuration
    - Environment Based Configuration
    - Directory Configuration
- Next Steps
    - Laravel The Full Stack Framework
    - Laravel The API Backend

Meet Laravel

Laravel is a web application framework with expressive, elegant syntax. A web framework provides a structure and starting point for creating your application, allowing you to focus on creating something amazing while we sweat the details.

Laravel strives to provide an amazing developer experience while providing powerful features such as thorough dependency injection, an expressive database abstraction layer, queues and scheduled jobs, unit and integration testing, and more.

Whether you are new to PHP or web frameworks or have years of experience, Laravel is a framework that can grow with you. We'll help you take your first steps as a web developer or give you a boost as you take your expertise to the next level. We can't wait to see what you build.

Why Laravel?

There are a variety of tools and frameworks available to you when building a web application. However, we believe Laravel is the best choice for building modern, full-stack web applications.

A Progressive Framework

We like to call Laravel a "progressive" framework. By that, we mean that Laravel grows with you. If you're just taking your first steps into web development, Laravel's vast library of documentation, guides, and video tutorials will help you learn the ropes without becoming overwhelmed.

If you're a senior developer, Laravel gives you robust tools for dependency injection, unit testing, queues, real-time events, and more. Laravel is fine-tuned for building professional web applications and ready to handle enterprise work loads.

A Scalable Framework

Laravel is incredibly scalable. Thanks to the scaling-friendly nature of PHP and Laravel's built-in support for fast, distributed cache systems like Redis, horizontal scaling with Laravel is a breeze. In fact, Laravel applications have been easily scaled to handle hundreds of millions of requests per month.

Need extreme scaling? Platforms like Laravel Vapor allow you to run your Laravel application at nearly limitless scale on AWS's latest serverless technology.

A Community Framework

Laravel combines the best packages in the PHP ecosystem to offer the most robust and developer friendly framework available. In addition, thousands of talented developers from around the world have contributed to the framework. Who knows, maybe you'll even become a Laravel contributor.

Your First Laravel Project

We want it to be as easy as possible to get started with Laravel. There are a variety of options for developing and running a Laravel project on your own computer. While you may wish to explore these options at a later time, Laravel provides Sail, a built-in solution for running your Laravel project using Docker.

Docker is a tool for running applications and services in small, light-weight "containers" which do not interfere with your local computer's installed software or configuration. This means you don't have to worry about configuring or setting up complicated development tools such as web servers and databases on your personal computer. To get started, you only need to install Docker Desktop.

Laravel Sail is a light-weight command-line interface for interacting with Laravel's default Docker configuration. Sail provides a great starting point for building a Laravel application using PHP, MySQL, and Redis without requiring prior Docker experience.

{tip} Already a Docker expert? Don't worry! Everything about Sail can be customized using the `docker-compose.yml` file included with Laravel.

Getting Started On macOS

If you're developing on a Mac and Docker Desktop is already installed, you can use a simple terminal command to create a new Laravel project. For example, to create a new Laravel application in a directory named "example-app", you may run the following command in your terminal:

```
curl -s "https://laravel.build/example-app" | bash
```

Of course, you can change "example-app" in this URL to anything you like. The Laravel application's directory will be created within the directory you execute the command from.

After the project has been created, you can navigate to the application directory and start Laravel Sail. Laravel Sail provides a simple command-line interface for interacting with Laravel's default Docker configuration:

```
cd example-app

./vendor/bin/sail up
```

The first time you run the Sail up command, Sail's application containers will be built on your machine. This could take several minutes. **Don't worry, subsequent attempts to start Sail will be much faster.**

Once the application's Docker containers have been started, you can access the application in your web browser at: http://localhost.

{tip} To continue learning more about Laravel Sail, review its complete documentation.

Getting Started On Windows

Before we create a new Laravel application on your Windows machine, make sure to install Docker Desktop. Next, you should ensure that Windows Subsystem for Linux 2 (WSL2) is installed and enabled. WSL allows you to run Linux binary executables natively on Windows 10. Information on how to install and enable WSL2 can be found within Microsoft's developer environment documentation.

{tip} After installing and enabling WSL2, you should ensure that Docker Desktop is configured to use the WSL2 backend.

Next, you are ready to create your first Laravel project. Launch Windows Terminal and begin a new terminal session for your WSL2 Linux operating system. Next, you can use a simple terminal command to create a new Laravel project. For example, to create a new Laravel application in a directory named "example-app", you may run the following command in your terminal:

```
curl -s https://laravel.build/example-app | bash
```

Of course, you can change "example-app" in this URL to anything you like. The Laravel application's directory will be created within the directory you execute the command from.

After the project has been created, you can navigate to the application directory and start Laravel Sail. Laravel Sail provides a simple command-line interface for interacting with Laravel's default Docker configuration:

```
cd example-app

./vendor/bin/sail up
```

The first time you run the Sail up command, Sail's application containers will be built on your machine. This could take several minutes. **Don't worry, subsequent attempts to start Sail will be much faster.**

Once the application's Docker containers have been started, you can access the application in your web browser at: http://localhost.

{tip} To continue learning more about Laravel Sail, review its complete documentation.

Developing Within WSL2

Of course, you will need to be able to modify the Laravel application files that were created within your WSL2 installation. To accomplish this, we recommend using Microsoft's Visual Studio Code editor and their first-party extension for Remote Development.

Once these tools are installed, you may open any Laravel project by executing the `code .` command from your application's root directory using Windows Terminal.

Getting Started On Linux

If you're developing on Linux and Docker is already installed, you can use a simple terminal command to create a new Laravel project. For example, to create a new Laravel application in a directory named "example-app", you may run the following command in your terminal:

```
curl -s https://laravel.build/example-app | bash
```

Of course, you can change "example-app" in this URL to anything you like. The Laravel application's directory will be created within the directory you execute the command from.

After the project has been created, you can navigate to the application directory and start Laravel Sail. Laravel Sail provides a simple command-line interface for interacting with Laravel's default Docker configuration:

```
cd example-app

./vendor/bin/sail up
```

The first time you run the Sail `up` command, Sail's application containers will be built on your machine. This could take several minutes. **Don't worry, subsequent attempts to start Sail will be much faster.**

Once the application's Docker containers have been started, you can access the application in your web browser at: http://localhost.

{tip} To continue learning more about Laravel Sail, review its complete documentation.

Choosing Your Sail Services

When creating a new Laravel application via Sail, you may use the `with` query string variable to choose which services should be configured in your new application's `docker-compose.yml` file. Available services include `mysql`, `pgsql`, `mariadb`, `redis`, `memcached`, `meilisearch`, `minio`, `selenium`, and `mailhog`:

```
curl -s "https://laravel.build/example-app?with=mysql,redis" | bash
```

If you do not specify which services you would like configured, a default stack of `mysql`, `redis`, `meilisearch`, `mailhog`, and `selenium` will be configured.

You may instruct Sail to install a default Devcontainer by adding the `devcontainer` parameter to the URL:

```
curl -s
"https://laravel.build/example-app?with=mysql,redis&devcontainer" | bash
```

Installation Via Composer

If your computer already has PHP and Composer installed, you may create a new Laravel project by using Composer directly. After the application has been created, you may start

Laravel's local development server using the Artisan CLI's `serve` command:

```
composer create-project laravel/laravel example-app

cd example-app

php artisan serve
```

The Laravel Installer

Or, you may install the Laravel Installer as a global Composer dependency:

```
composer global require laravel/installer

laravel new example-app

cd example-app

php artisan serve
```

Make sure to place Composer's system-wide vendor bin directory in your `$PATH` so the `laravel` executable can be located by your system. This directory exists in different locations based on your operating system; however, some common locations include:

- macOS: `$HOME/.composer/vendor/bin` - Windows: `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin` - GNU / Linux Distributions: `$HOME/.config/composer/vendor/bin` or `$HOME/.composer/vendor/bin`

For convenience, the Laravel installer can also create a Git repository for your new project. To indicate that you want a Git repository to be created, pass the `--git` flag when creating a new project:

```
laravel new example-app --git
```

This command will initialize a new Git repository for your project and automatically commit the base Laravel skeleton. The `git` flag assumes you have properly installed and configured Git. You can also use the `--branch` flag to set the initial branch name:

```
laravel new example-app --git --branch="main"
```

Instead of using the `--git` flag, you may also use the `--github` flag to create a Git repository and also create a corresponding private repository on GitHub:

```
laravel new example-app --github
```

The created repository will then be available at `https://github.com/<your-account>/example-app`. The `github` flag assumes you have properly installed the GitHub CLI and are authenticated with GitHub. Additionally, you should have `git` installed and properly configured. If needed, you can pass additional flags that are supported by the GitHub CLI:

```
laravel new example-app --github="--public"
```

You may use the `--organization` flag to create the repository under a specific GitHub organization:

```
laravel new example-app --github="--public" --organization="laravel"
```

Initial Configuration

All of the configuration files for the Laravel framework are stored in the `config` directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

Laravel needs almost no additional configuration out of the box. You are free to get started developing! However, you may wish to review the `config/app.php` file and its documentation. It contains several options such as `timezone` and `locale` that you may wish to change according to your application.

Environment Based Configuration

Since many of Laravel's configuration option values may vary depending on whether your application is running on your local computer or on a production web server, many important configuration values are defined using the `.env` file that exists at the root of your application.

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

{tip} For more information about the `.env` file and environment based configuration, check out the full configuration documentation.

Directory Configuration

Laravel should always be served out of the root of the "web directory" configured for your web server. You should not attempt to serve a Laravel application out of a subdirectory of the "web directory". Attempting to do so could expose sensitive files that exist within your application.

Next Steps

Now that you have created your Laravel project, you may be wondering what to learn next. First, we strongly recommend becoming familiar with how Laravel works by reading the following documentation:

- [Request Lifecycle](#lifecycle) - [Configuration](#configuration) - [Directory Structure](#structure) - [Service Container](#container) - [Facades](#facades)

How you want to use Laravel will also dictate the next steps on your journey. There are a variety of ways to use Laravel, and we'll explore two primary use cases for the framework below.

Laravel The Full Stack Framework

Laravel may serve as a full stack framework. By "full stack" framework we mean that you are going to use Laravel to route requests to your application and render your frontend via Blade templates or using a single-page application hybrid technology like Inertia.js. This is the most common way to use the Laravel framework.

If this is how you plan to use Laravel, you may want to check out our documentation on routing, views, or the Eloquent ORM. In addition, you might be interested in learning about community packages like Livewire and Inertia.js. These packages allow you to use Laravel as a full-stack framework while enjoying many of the UI benefits provided by single-page JavaScript applications.

If you are using Laravel as a full stack framework, we also strongly encourage you to learn how to compile your application's CSS and JavaScript using Laravel Mix.

{tip} If you want to get a head start building your application, check out one of our official application starter kits.

Laravel The API Backend

Laravel may also serve as an API backend to a JavaScript single-page application or mobile application. For example, you might use Laravel as an API backend for your Next.js application. In this context, you may use Laravel to provide authentication and data storage / retrieval for your application, while also taking advantage of Laravel's powerful services such as queues, emails, notifications, and more.

If this is how you plan to use Laravel, you may want to check out our documentation on routing, Laravel Sanctum, and the Eloquent ORM.

{tip} Need a head start scaffolding your Laravel backend and Next.js frontend? Laravel Breeze offers an API stack as well as a Next.js frontend implementation so you can get started in minutes.

Authentication

- Introduction
    - Starter Kits
    - Database Considerations
    - Ecosystem Overview
- Authentication Quickstart
    - Install A Starter Kit
    - Retrieving The Authenticated User
    - Protecting Routes
    - Login Throttling
- Manually Authenticating Users
    - Remembering Users
    - Other Authentication Methods
- HTTP Basic Authentication
    - Stateless HTTP Basic Authentication
- Logging Out
    - Invalidating Sessions On Other Devices
- Password Confirmation
    - Configuration
    - Routing
    - Protecting Routes
- Adding Custom Guards
    - Closure Request Guards
- Adding Custom User Providers
    - The User Provider Contract
    - The Authenticatable Contract
- Social Authentication
- Events

Introduction

Many web applications provide a way for their users to authenticate with the application and "login". Implementing this feature in web applications can be a complex and potentially risky endeavor. For this reason, Laravel strives to give you the tools you need to implement authentication quickly, securely, and easily.

At its core, Laravel's authentication facilities are made up of "guards" and "providers". Guards define how users are authenticated for each request. For example, Laravel ships with a `session` guard which maintains state using session storage and cookies.

Providers define how users are retrieved from your persistent storage. Laravel ships with support for retrieving users using Eloquent and the database query builder. However, you are free to define additional providers as needed for your application.

Your application's authentication configuration file is located at `config/auth.php`. This file contains several well-documented options for tweaking the behavior of Laravel's authentication services.

{tip} Guards and providers should not be confused with "roles" and "permissions". To learn more about authorizing user actions via permissions, please refer to the authorization documentation.

Starter Kits

Want to get started fast? Install a Laravel application starter kit in a fresh Laravel application. After migrating your database, navigate your browser to `/register` or any other URL that is assigned to your application. The starter kits will take care of scaffolding your entire authentication system!

Even if you choose not to use a starter kit in your final Laravel application, installing the Laravel Breeze starter kit can be a wonderful opportunity to learn how to implement all of Laravel's authentication functionality in an actual Laravel project. Since Laravel Breeze creates authentication controllers, routes, and views for you, you can examine the code within these files to learn how Laravel's authentication features may be implemented.

Database Considerations

By default, Laravel includes an `App\Models\User` Eloquent model in your `app/Models` directory. This model may be used with the default Eloquent authentication driver. If your application is not using Eloquent, you may use the `database` authentication provider which uses the Laravel query builder.

When building the database schema for the `App\Models\User` model, make sure the password column is at least 60 characters in length. Of course, the `users` table migration that is included in new Laravel applications already creates a column that exceeds this length.

Also, you should verify that your `users` (or equivalent) table contains a nullable, string `remember_token` column of 100 characters. This column will be used to store a token for users that select the "remember me" option when logging into your application. Again, the default `users` table migration that is included in new Laravel applications already contains this column.

Ecosystem Overview

Laravel offers several packages related to authentication. Before continuing, we'll review the general authentication ecosystem in Laravel and discuss each package's intended purpose.

First, consider how authentication works. When using a web browser, a user will provide their username and password via a login form. If these credentials are correct, the application will store information about the authenticated user in the user's session. A cookie issued to the browser contains the session ID so that subsequent requests to the application can associate the user with the correct session. After the session cookie is received, the application will retrieve the session data based on the session ID, note that the authentication information has been stored in the session, and will consider the user as "authenticated".

When a remote service needs to authenticate to access an API, cookies are not typically used for authentication because there is no web browser. Instead, the remote service sends an API token to the API on each request. The application may validate the incoming token against a table of valid API tokens and "authenticate" the request as being performed by the user associated with that API token.

Laravel's Built-in Browser Authentication Services

Laravel includes built-in authentication and session services which are typically accessed via the `Auth` and `Session` facades. These features provide cookie-based authentication for

requests that are initiated from web browsers. They provide methods that allow you to verify a user's credentials and authenticate the user. In addition, these services will automatically store the proper authentication data in the user's session and issue the user's session cookie. A discussion of how to use these services is contained within this documentation.

Application Starter Kits

As discussed in this documentation, you can interact with these authentication services manually to build your application's own authentication layer. However, to help you get started more quickly, we have released free packages that provide robust, modern scaffolding of the entire authentication layer. These packages are Laravel Breeze, Laravel Jetstream, and Laravel Fortify.

Laravel Breeze is a simple, minimal implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is comprised of simple Blade templates styled with Tailwind CSS. To get started, check out the documentation on Laravel's application starter kits.

Laravel Fortify is a headless authentication backend for Laravel that implements many of the features found in this documentation, including cookie-based authentication as well as other features such as two-factor authentication and email verification. Fortify provides the authentication backend for Laravel Jetstream or may be used independently in combination with Laravel Sanctum to provide authentication for an SPA that needs to authenticate with Laravel.

Laravel Jetstream is a robust application starter kit that consumes and exposes Laravel Fortify's authentication services with a beautiful, modern UI powered by Tailwind CSS, Livewire, and / or Inertia.js. Laravel Jetstream includes optional support for two-factor authentication, team support, browser session management, profile management, and built-in integration with Laravel Sanctum to offer API token authentication. Laravel's API authentication offerings are discussed below.

Laravel's API Authentication Services

Laravel provides two optional packages to assist you in managing API tokens and authenticating requests made with API tokens: Passport and Sanctum. Please note that these libraries and Laravel's built-in cookie based authentication libraries are not mutually exclusive. These libraries primarily focus on API token authentication while the built-in authentication services focus on cookie based browser authentication. Many applications will use both Laravel's built-in cookie based authentication services and one of Laravel's API authentication packages.

Passport

Passport is an OAuth2 authentication provider, offering a variety of OAuth2 "grant types"

which allow you to issue various types of tokens. In general, this is a robust and complex package for API authentication. However, most applications do not require the complex features offered by the OAuth2 spec, which can be confusing for both users and developers. In addition, developers have been historically confused about how to authenticate SPA applications or mobile applications using OAuth2 authentication providers like Passport.

Sanctum

In response to the complexity of OAuth2 and developer confusion, we set out to build a simpler, more streamlined authentication package that could handle both first-party web requests from a web browser and API requests via tokens. This goal was realized with the release of Laravel Sanctum, which should be considered the preferred and recommended authentication package for applications that will be offering a first-party web UI in addition to an API, or will be powered by a single-page application (SPA) that exists separately from the backend Laravel application, or applications that offer a mobile client.

Laravel Sanctum is a hybrid web / API authentication package that can manage your application's entire authentication process. This is possible because when Sanctum based applications receive a request, Sanctum will first determine if the request includes a session cookie that references an authenticated session. Sanctum accomplishes this by calling Laravel's built-in authentication services which we discussed earlier. If the request is not being authenticated via a session cookie, Sanctum will inspect the request for an API token. If an API token is present, Sanctum will authenticate the request using that token. To learn more about this process, please consult Sanctum's "how it works" documentation.

Laravel Sanctum is the API package we have chosen to include with the Laravel Jetstream application starter kit because we believe it is the best fit for the majority of web application's authentication needs.

Summary & Choosing Your Stack

In summary, if your application will be accessed using a browser and you are building a monolithic Laravel application, your application will use Laravel's built-in authentication services.

Next, if your application offers an API that will be consumed by third parties, you will choose between Passport or Sanctum to provide API token authentication for your application. In general, Sanctum should be preferred when possible since it is a simple, complete solution for API authentication, SPA authentication, and mobile authentication, including support for "scopes" or "abilities".

If you are building a single-page application (SPA) that will be powered by a Laravel backend, you should use Laravel Sanctum. When using Sanctum, you will either need to manually implement your own backend authentication routes or utilize Laravel Fortify as a headless authentication backend service that provides routes and controllers for features such as registration, password reset, email verification, and more.

Passport may be chosen when your application absolutely needs all of the features provided by the OAuth2 specification.

And, if you would like to get started quickly, we are pleased to recommend Laravel Jetstream as a quick way to start a new Laravel application that already uses our preferred authentication stack of Laravel's built-in authentication services and Laravel Sanctum.

Authentication Quickstart

{note} This portion of the documentation discusses authenticating users via the Laravel application starter kits, which includes UI scaffolding to help you get started quickly. If you would like to integrate with Laravel's authentication systems directly, check out the documentation on manually authenticating users.

Install A Starter Kit

First, you should install a Laravel application starter kit. Our current starter kits, Laravel Breeze and Laravel Jetstream, offer beautifully designed starting points for incorporating authentication into your fresh Laravel application.

Laravel Breeze is a minimal, simple implementation of all of Laravel's authentication features, including login, registration, password reset, email verification, and password confirmation. Laravel Breeze's view layer is made up of simple Blade templates styled with Tailwind CSS. Breeze also offers an Inertia based scaffolding option using Vue or React.

Laravel Jetstream is a more robust application starter kit that includes support for scaffolding your application with Livewire or Inertia.js and Vue. In addition, Jetstream features optional support for two-factor authentication, teams, profile management, browser session management, API support via Laravel Sanctum, account deletion, and more.

Retrieving The Authenticated User

After installing an authentication starter kit and allowing users to register and authenticate with your application, you will often need to interact with the currently authenticated user. While handling an incoming request, you may access the authenticated user via the `Auth` facade's `user` method:

```
use Illuminate\Support\Facades\Auth;

// Retrieve the currently authenticated user...
$user = Auth::user();

// Retrieve the currently authenticated user's ID...
$id = Auth::id();
```

Alternatively, once a user is authenticated, you may access the authenticated user via an `Illuminate\Http\Request` instance. Remember, type-hinted classes will automatically be injected into your controller methods. By type-hinting the `Illuminate\Http\Request` object, you may gain convenient access to the authenticated user from any controller method in your application via the request's `user` method:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Update the flight information for an existing flight.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        // $request->user()
    }
}
```

Determining If The Current User Is Authenticated

To determine if the user making the incoming HTTP request is authenticated, you may use the `check` method on the `Auth` facade. This method will return `true` if the user is authenticated:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // The user is logged in...
}
```

{tip} Even though it is possible to determine if a user is authenticated using the `check` method, you will typically use a middleware to verify that the user is authenticated before allowing the user access to certain routes / controllers. To learn more about this, check out the documentation on protecting routes.

Protecting Routes

Route middleware can be used to only allow authenticated users to access a given route. Laravel ships with an `auth` middleware, which references the `Illuminate\Auth\Middleware\Authenticate` class. Since this middleware is already registered in your application's HTTP kernel, all you need to do is attach the middleware to a route definition:

```
Route::get('/flights', function () {
    // Only authenticated users may access this route...
})->middleware('auth');
```

Redirecting Unauthenticated Users

When the `auth` middleware detects an unauthenticated user, it will redirect the user to the `login` named route. You may modify this behavior by updating the `redirectTo` function in your application's `app/Http/Middleware/Authenticate.php` file:

```
/**
 * Get the path the user should be redirected to.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return string
 */
protected function redirectTo($request)
{
    return route('login');
}
```

Specifying A Guard

When attaching the `auth` middleware to a route, you may also specify which "guard" should be used to authenticate the user. The guard specified should correspond to one of the keys in the `guards` array of your `auth.php` configuration file:

```
Route::get('/flights', function () {
    // Only authenticated users may access this route...
})->middleware('auth:admin');
```

Login Throttling

If you are using the Laravel Breeze or Laravel Jetstream starter kits, rate limiting will automatically be applied to login attempts. By default, the user will not be able to login for one minute if they fail to provide the correct credentials after several attempts. The throttling is unique to the user's username / email address and their IP address.

{tip} If you would like to rate limit other routes in your application, check out the rate limiting documentation.

Manually Authenticating Users

You are not required to use the authentication scaffolding included with Laravel's application starter kits. If you choose not to use this scaffolding, you will need to manage user authentication using the Laravel authentication classes directly. Don't worry, it's a cinch!

We will access Laravel's authentication services via the `Auth` facade, so we'll need to make sure to import the `Auth` facade at the top of the class. Next, let's check out the `attempt` method. The `attempt` method is normally used to handle authentication attempts from your application's "login" form. If authentication is successful, you should regenerate the user's session to prevent session fixation:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class LoginController extends Controller
{
    /**
     * Handle an authentication attempt.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function authenticate(Request $request)
    {
        $credentials = $request->validate([
            'email' => ['required', 'email'],
            'password' => ['required'],
        ]);

        if (Auth::attempt($credentials)) {
            $request->session()->regenerate();

            return redirect()->intended('dashboard');
        }

        return back()->withErrors([
            'email' => 'The provided credentials do not match our
records.',
        ]);
    }
}
```

The `attempt` method accepts an array of key / value pairs as its first argument. The values in the array will be used to find the user in your database table. So, in the example above, the user will be retrieved by the value of the `email` column. If the user is found, the hashed password stored in the database will be compared with the `password` value passed to the method via the array. You should not hash the incoming request's `password` value, since the framework will automatically hash the value before comparing it to the hashed password in the database. An authenticated session will be started for the user if the two hashed passwords match.

Remember, Laravel's authentication services will retrieve users from your database based on your authentication guard's "provider" configuration. In the default `config/auth.php` configuration file, the Eloquent user provider is specified and it is instructed to use the `App\Models\User` model when retrieving users. You may change these values within your configuration file based on the needs of your application.

The `attempt` method will return `true` if authentication was successful. Otherwise, `false` will be returned.

The `intended` method provided by Laravel's redirector will redirect the user to the URL they were attempting to access before being intercepted by the authentication middleware. A fallback URI may be given to this method in case the intended destination is not available.

Specifying Additional Conditions

If you wish, you may also add extra query conditions to the authentication query in addition to the user's email and password. To accomplish this, we may simply add the query conditions to the array passed to the `attempt` method. For example, we may verify that the user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active'
=> 1])) {
    // Authentication was successful...
}
```

{note} In these examples, `email` is not a required option, it is merely used as an example. You should use whatever column name corresponds to a "username" in your database table.

Accessing Specific Guard Instances

Via the `Auth` facade's `guard` method, you may specify which guard instance you would like to utilize when authenticating the user. This allows you to manage authentication for separate parts of your application using entirely separate authenticatable models or user tables.

The guard name passed to the `guard` method should correspond to one of the guards configured in your `auth.php` configuration file:

```
if (Auth::guard('admin')->attempt($credentials)) {
    // ...
}
```

Remembering Users

Many web applications provide a "remember me" checkbox on their login form. If you would like to provide "remember me" functionality in your application, you may pass a boolean value as the second argument to the `attempt` method.

When this value is `true`, Laravel will keep the user authenticated indefinitely or until they manually logout. Your `users` table must include the string `remember_token` column, which will be used to store the "remember me" token. The `users` table migration included with new Laravel applications already includes this column:

```
use Illuminate\Support\Facades\Auth;

if (Auth::attempt(['email' => $email, 'password' => $password],
$remember)) {
    // The user is being remembered...
}
```

Other Authentication Methods

Authenticate A User Instance

If you need to set an existing user instance as the currently authenticated user, you may pass the user instance to the `Auth` facade's `login` method. The given user instance must be an implementation of the `Illuminate\Contracts\Auth\Authenticatable` contract. The `App\Models\User` model included with Laravel already implements this interface. This method of authentication is useful when you already have a valid user instance, such as directly after a user registers with your application:

```
use Illuminate\Support\Facades\Auth;

Auth::login($user);
```

You may pass a boolean value as the second argument to the `login` method. This value indicates if "remember me" functionality is desired for the authenticated session. Remember, this means that the session will be authenticated indefinitely or until the user manually logs out of the application:

```
Auth::login($user, $remember = true);
```

If needed, you may specify an authentication guard before calling the `login` method:

```
Auth::guard('admin')->login($user);
```

Authenticate A User By ID

To authenticate a user using their database record's primary key, you may use the `loginUsingId` method. This method accepts the primary key of the user you wish to authenticate:

```
Auth::loginUsingId(1);
```

You may pass a boolean value as the second argument to the `loginUsingId` method. This value indicates if "remember me" functionality is desired for the authenticated session. Remember, this means that the session will be authenticated indefinitely or until the user manually logs out of the application:

```
Auth::loginUsingId(1, $remember = true);
```

Authenticate A User Once

You may use the `once` method to authenticate a user with the application for a single request. No sessions or cookies will be utilized when calling this method:

```
if (Auth::once($credentials)) {
    //
}
```

HTTP Basic Authentication

HTTP Basic Authentication provides a quick way to authenticate users of your application without setting up a dedicated "login" page. To get started, attach the `auth.basic` middleware to a route. The `auth.basic` middleware is included with the Laravel framework, so you do not need to define it:

```
Route::get('/profile', function () {
    // Only authenticated users may access this route...
})->middleware('auth.basic');
```

Once the middleware has been attached to the route, you will automatically be prompted for credentials when accessing the route in your browser. By default, the `auth.basic` middleware will assume the `email` column on your `users` database table is the user's "username".

A Note On FastCGI

If you are using PHP FastCGI and Apache to serve your Laravel application, HTTP Basic authentication may not work correctly. To correct these problems, the following lines may be added to your application's `.htaccess` file:

```
RewriteCond %{HTTP:Authorization} ^(.+)$
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Stateless HTTP Basic Authentication

You may also use HTTP Basic Authentication without setting a user identifier cookie in the session. This is primarily helpful if you choose to use HTTP Authentication to authenticate requests to your application's API. To accomplish this, define a middleware that calls the `onceBasic` method. If no response is returned by the `onceBasic` method, the request may be passed further into the application:

```php
<?php

namespace App\Http\Middleware;

use Illuminate\Support\Facades\Auth;

class AuthenticateOnceWithBasicAuth
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }

}
```

Next, register the route middleware and attach it to a route:

```php
Route::get('/api/user', function () {
    // Only authenticated users may access this route...
})->middleware('auth.basic.once');
```

Logging Out

To manually log users out of your application, you may use the `logout` method provided by the `Auth` facade. This will remove the authentication information from the user's session so that subsequent requests are not authenticated.

In addition to calling the `logout` method, it is recommended that you invalidate the user's session and regenerate their CSRF token. After logging the user out, you would typically redirect the user to the root of your application:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Log the user out of the application.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function logout(Request $request)
{
    Auth::logout();

    $request->session()->invalidate();

    $request->session()->regenerateToken();

    return redirect('/');
}
```

Invalidating Sessions On Other Devices

Laravel also provides a mechanism for invalidating and "logging out" a user's sessions that are active on other devices without invalidating the session on their current device. This feature is typically utilized when a user is changing or updating their password and you would like to invalidate sessions on other devices while keeping the current device authenticated.

Before getting started, you should make sure that the

`Illuminate\Session\Middleware\AuthenticateSession` middleware is present and un-commented in your `App\Http\Kernel` class' `web` middleware group:

```
'web' => [
    // ...
    \Illuminate\Session\Middleware\AuthenticateSession::class,
    // ...
],
```

Then, you may use the `logoutOtherDevices` method provided by the `Auth` facade. This method requires the user to confirm their current password, which your application should accept through an input form:

```
use Illuminate\Support\Facades\Auth;

Auth::logoutOtherDevices($currentPassword);
```

When the `logoutOtherDevices` method is invoked, the user's other sessions will be invalidated entirely, meaning they will be "logged out" of all guards they were previously authenticated by.

Password Confirmation

While building your application, you may occasionally have actions that should require the user to confirm their password before the action is performed or before the user is redirected to a sensitive area of the application. Laravel includes built-in middleware to make this process a breeze. Implementing this feature will require you to define two routes: one route to display a view asking the user to confirm their password and another route to confirm that the password is valid and redirect the user to their intended destination.

{tip} The following documentation discusses how to integrate with Laravel's password confirmation features directly; however, if you would like to get started more quickly, the Laravel application starter kits include support for this feature!

Configuration

After confirming their password, a user will not be asked to confirm their password again for three hours. However, you may configure the length of time before the user is re-prompted for their password by changing the value of the `password_timeout` configuration value within your application's `config/auth.php` configuration file.

Routing

The Password Confirmation Form

First, we will define a route to display a view that requests the user to confirm their password:

```
Route::get('/confirm-password', function () {
    return view('auth.confirm-password');
})->middleware('auth')->name('password.confirm');
```

As you might expect, the view that is returned by this route should have a form containing a `password` field. In addition, feel free to include text within the view that explains that the user is entering a protected area of the application and must confirm their password.

Confirming The Password

Next, we will define a route that will handle the form request from the "confirm password" view. This route will be responsible for validating the password and redirecting the user to their intended destination:

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Redirect;

Route::post('/confirm-password', function (Request $request) {
    if (! Hash::check($request->password, $request->user()->password)) {
        return back()->withErrors([
            'password' => ['The provided password does not match our
records.']
        ]);
    }

    $request->session()->passwordConfirmed();

    return redirect()->intended();
})->middleware(['auth', 'throttle:6,1'])->name('password.confirm');
```

Before moving on, let's examine this route in more detail. First, the request's `password` field is determined to actually match the authenticated user's password. If the password is valid, we need to inform Laravel's session that the user has confirmed their password. The `passwordConfirmed` method will set a timestamp in the user's session that Laravel can use to determine when the user last confirmed their password. Finally, we can redirect the user to their intended destination.

Protecting Routes

You should ensure that any route that performs an action which requires recent password confirmation is assigned the `password.confirm` middleware. This middleware is included with the default installation of Laravel and will automatically store the user's intended destination in the session so that the user may be redirected to that location after confirming their password. After storing the user's intended destination in the session, the middleware will redirect the user to the `password.confirm` named route:

```
Route::get('/settings', function () {
    // ...
})->middleware(['password.confirm']);

Route::post('/settings', function () {
    // ...
})->middleware(['password.confirm']);
```

Adding Custom Guards

You may define your own authentication guards using the `extend` method on the `Auth` facade. You should place your call to the `extend` method within a service provider. Since Laravel already ships with an `AuthServiceProvider`, we can place the code in that provider:

```
<?php

namespace App\Providers;

use App\Services\Auth\JwtGuard;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::extend('jwt', function ($app, $name, array $config) {
            // Return an instance of Illuminate\Contracts\Auth\Guard...

            return new
JwtGuard(Auth::createUserProvider($config['provider']));
        });
    }
}
```

As you can see in the example above, the callback passed to the `extend` method should return an implementation of `Illuminate\Contracts\Auth\Guard`. This interface contains a few methods you will need to implement to define a custom guard. Once your custom guard has been defined, you may reference the guard in the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

Closure Request Guards

The simplest way to implement a custom, HTTP request based authentication system is by using the `Auth::viaRequest` method. This method allows you to quickly define your authentication process using a single closure.

To get started, call the `Auth::viaRequest` method within the `boot` method of your `AuthServiceProvider`. The `viaRequest` method accepts an authentication driver name as its first argument. This name can be any string that describes your custom guard. The second argument passed to the method should be a closure that receives the incoming HTTP request and returns a user instance or, if authentication fails, `null`:

```
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

/**
 * Register any application authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Auth::viaRequest('custom-token', function (Request $request) {
        return User::where('token', $request->token)->first();
    });
}
```

Once your custom authentication driver has been defined, you may configure it as a driver within the `guards` configuration of your `auth.php` configuration file:

```
'guards' => [
    'api' => [
        'driver' => 'custom-token',
    ],
],
```

Adding Custom User Providers

If you are not using a traditional relational database to store your users, you will need to extend Laravel with your own authentication user provider. We will use the `provider` method on the `Auth` facade to define a custom user provider. The user provider resolver should return an implementation of `Illuminate\Contracts\Auth\UserProvider`:

```
<?php

namespace App\Providers;

use App\Extensions\MongoUserProvider;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
ServiceProvider;
use Illuminate\Support\Facades\Auth;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        Auth::provider('mongo', function ($app, array $config) {
            // Return an instance of
Illuminate\Contracts\Auth\UserProvider...

            return new
MongoUserProvider($app->make('mongo.connection'));
        });
    }
}
```

After you have registered the provider using the `provider` method, you may switch to the new user provider in your `auth.php` configuration file. First, define a `provider` that uses your new driver:

```
'providers' => [
    'users' => [
        'driver' => 'mongo',
    ],
],
```

Finally, you may reference this provider in your `guards` configuration:

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],
],
```

The User Provider Contract

`Illuminate\Contracts\Auth\UserProvider` implementations are responsible for fetching an `Illuminate\Contracts\Auth\Authenticatable` implementation out of a persistent storage system, such as MySQL, MongoDB, etc. These two interfaces allow the Laravel authentication mechanisms to continue functioning regardless of how the user data is stored or what type of class is used to represent the authenticated user:

Let's take a look at the `Illuminate\Contracts\Auth\UserProvider` contract:

```php
<?php

namespace Illuminate\Contracts\Auth;

interface UserProvider
{
    public function retrieveById($identifier);
    public function retrieveByToken($identifier, $token);
    public function updateRememberToken(Authenticatable $user, $token);
    public function retrieveByCredentials(array $credentials);
    public function validateCredentials(Authenticatable $user, array
$credentials);
}
```

The `retrieveById` function typically receives a key representing the user, such as an auto-incrementing ID from a MySQL database. The `Authenticatable` implementation matching the ID should be retrieved and returned by the method.

The `retrieveByToken` function retrieves a user by their unique `$identifier` and "remember me" `$token`, typically stored in a database column like `remember_token`. As with the previous method, the `Authenticatable` implementation with a matching token value should be returned by this method.

The `updateRememberToken` method updates the `$user` instance's `remember_token` with the new `$token`. A fresh token is assigned to users on a successful "remember me" authentication attempt or when the user is logging out.

The `retrieveByCredentials` method receives the array of credentials passed to the `Auth::attempt` method when attempting to authenticate with an application. The method should then "query" the underlying persistent storage for the user matching those credentials. Typically, this method will run a query with a "where" condition that searches for a user record with a "username" matching the value of `$credentials['username']`. The method should return an implementation of `Authenticatable`. **This method should not attempt to do any password validation or authentication.**

The `validateCredentials` method should compare the given `$user` with the `$credentials` to authenticate the user. For example, this method will typically use the `Hash::check` method to compare the value of `$user->getAuthPassword()` to the value of `$credentials['password']`. This method should return `true` or `false` indicating whether the password is valid.

The Authenticatable Contract

Now that we have explored each of the methods on the `UserProvider`, let's take a look at the `Authenticatable` contract. Remember, user providers should return implementations of this interface from the `retrieveById`, `retrieveByToken`, and `retrieveByCredentials` methods:

```
<?php

namespace Illuminate\Contracts\Auth;

interface Authenticatable
{
    public function getAuthIdentifierName();
    public function getAuthIdentifier();
    public function getAuthPassword();
    public function getRememberToken();
    public function setRememberToken($value);
    public function getRememberTokenName();
}
```

This interface is simple. The `getAuthIdentifierName` method should return the name of the "primary key" field of the user and the `getAuthIdentifier` method should return the "primary key" of the user. When using a MySQL back-end, this would likely be the auto-incrementing primary key assigned to the user record. The `getAuthPassword` method should return the user's hashed password.

This interface allows the authentication system to work with any "user" class, regardless of what ORM or storage abstraction layer you are using. By default, Laravel includes a `App\Models\User` class in the `app/Models` directory which implements this interface.

Events

Laravel dispatches a variety of events during the authentication process. You may attach listeners to these events in your `EventServiceProvider`:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Auth\Events\Registered' => [
        'App\Listeners\LogRegisteredUser',
    ],

    'Illuminate\Auth\Events\Attempting' => [
        'App\Listeners\LogAuthenticationAttempt',
    ],

    'Illuminate\Auth\Events\Authenticated' => [
        'App\Listeners\LogAuthenticated',
    ],

    'Illuminate\Auth\Events\Login' => [
        'App\Listeners\LogSuccessfulLogin',
    ],

    'Illuminate\Auth\Events\Failed' => [
        'App\Listeners\LogFailedLogin',
    ],

    'Illuminate\Auth\Events\Validated' => [
        'App\Listeners\LogValidated',
    ],

    'Illuminate\Auth\Events\Verified' => [
        'App\Listeners\LogVerified',
    ],

    'Illuminate\Auth\Events\Logout' => [
        'App\Listeners\LogSuccessfulLogout',
    ],
```

```
    'Illuminate\Auth\Events\CurrentDeviceLogout' => [
        'App\Listeners\LogCurrentDeviceLogout',
    ],

    'Illuminate\Auth\Events\OtherDeviceLogout' => [
        'App\Listeners\LogOtherDeviceLogout',
    ],

    'Illuminate\Auth\Events\Lockout' => [
        'App\Listeners\LogLockout',
    ],

    'Illuminate\Auth\Events\PasswordReset' => [
        'App\Listeners\LogPasswordReset',
    ],
];
```

Authorization

- Introduction
- Gates
    - Writing Gates
    - Authorizing Actions
    - Gate Responses
    - Intercepting Gate Checks
    - Inline Authorization
- Creating Policies
    - Generating Policies
    - Registering Policies
- Writing Policies
    - Policy Methods
    - Policy Responses
    - Methods Without Models
    - Guest Users
    - Policy Filters
- Authorizing Actions Using Policies
    - Via The User Model
    - Via Controller Helpers
    - Via Middleware
    - Via Blade Templates
    - Supplying Additional Context

Introduction

In addition to providing built-in authentication services, Laravel also provides a simple way to authorize user actions against a given resource. For example, even though a user is authenticated, they may not be authorized to update or delete certain Eloquent models or database records managed by your application. Laravel's authorization features provide an easy, organized way of managing these types of authorization checks.

Laravel provides two primary ways of authorizing actions: gates and policies. Think of gates and policies like routes and controllers. Gates provide a simple, closure-based approach to authorization while policies, like controllers, group logic around a particular model or resource. In this documentation, we'll explore gates first and then examine policies.

You do not need to choose between exclusively using gates or exclusively using policies when building an application. Most applications will most likely contain some mixture of gates and policies, and that is perfectly fine! Gates are most applicable to actions which are not related to any model or resource, such as viewing an administrator dashboard. In contrast, policies should be used when you wish to authorize an action for a particular model or resource.

Gates

Writing Gates

{note} Gates are a great way to learn the basics of Laravel's authorization features; however, when building robust Laravel applications you should consider using policies to organize your authorization rules.

Gates are simply closures that determine if a user is authorized to perform a given action. Typically, gates are defined within the `boot` method of the `App\Providers\AuthServiceProvider` class using the `Gate` facade. Gates always receive a user instance as their first argument and may optionally receive additional arguments such as a relevant Eloquent model.

In this example, we'll define a gate to determine if a user can update a given `App\Models\Post` model. The gate will accomplish this by comparing the user's `id` against the `user_id` of the user that created the post:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', function (User $user, Post $post) {
        return $user->id === $post->user_id;
    });
}
```

Like controllers, gates may also be defined using a class callback array:

```
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    Gate::define('update-post', [PostPolicy::class, 'update']);
}
```

Authorizing Actions

To authorize an action using gates, you should use the `allows` or `denies` methods provided by the `Gate` facade. Note that you are not required to pass the currently authenticated user to these methods. Laravel will automatically take care of passing the user into the gate closure. It is typical to call the gate authorization methods within your application's controllers before performing an action that requires authorization:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Post  $post
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Post $post)
    {
        if (! Gate::allows('update-post', $post)) {
            abort(403);
        }

        // Update the post...
    }
}
```

If you would like to determine if a user other than the currently authenticated user is authorized to perform an action, you may use the `forUser` method on the `Gate` facade:

```php
if (Gate::forUser($user)->allows('update-post', $post)) {
    // The user can update the post...
}

if (Gate::forUser($user)->denies('update-post', $post)) {
    // The user can't update the post...
}
```

You may authorize multiple actions at a time using the `any` or `none` methods:

```
if (Gate::any(['update-post', 'delete-post'], $post)) {
    // The user can update or delete the post...
}

if (Gate::none(['update-post', 'delete-post'], $post)) {
    // The user can't update or delete the post...
}
```

Authorizing Or Throwing Exceptions

If you would like to attempt to authorize an action and automatically throw an `Illuminate\Auth\Access\AuthorizationException` if the user is not allowed to perform the given action, you may use the `Gate` facade's `authorize` method. Instances of `AuthorizationException` are automatically converted to a 403 HTTP response by Laravel's exception handler:

```
Gate::authorize('update-post', $post);

// The action is authorized...
```

Supplying Additional Context

The gate methods for authorizing abilities (`allows`, `denies`, `check`, `any`, `none`, `authorize`, `can`, `cannot`) and the authorization Blade directives (`@can`, `@cannot`, `@canany`) can receive an array as their second argument. These array elements are passed as parameters to the gate closure, and can be used for additional context when making authorization decisions:

```
use App\Models\Category;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

Gate::define('create-post', function (User $user, Category $category,
$pinned) {
    if (! $user->canPublishToGroup($category->group)) {
        return false;
    } elseif ($pinned && ! $user->canPinPosts()) {
        return false;
    }

    return true;
});

if (Gate::check('create-post', [$category, $pinned])) {
    // The user can create the post...
}
```

Gate Responses

So far, we have only examined gates that return simple boolean values. However, sometimes you may wish to return a more detailed response, including an error message. To do so, you may return an `Illuminate\Auth\Access\Response` from your gate:

```
use App\Models\User;
use Illuminate\Auth\Access\Response;
use Illuminate\Support\Facades\Gate;

Gate::define('edit-settings', function (User $user) {
    return $user->isAdmin
                ? Response::allow()
                : Response::deny('You must be an administrator.');
});
```

Even when you return an authorization response from your gate, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
$response = Gate::inspect('edit-settings');

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

When using the `Gate::authorize` method, which throws an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('edit-settings');

// The action is authorized...
```

Intercepting Gate Checks

Sometimes, you may wish to grant all abilities to a specific user. You may use the `before` method to define a closure that is run before all other authorization checks:

```
use Illuminate\Support\Facades\Gate;

Gate::before(function ($user, $ability) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

If the `before` closure returns a non-null result that result will be considered the result of the authorization check.

You may use the `after` method to define a closure to be executed after all other authorization checks:

```
Gate::after(function ($user, $ability, $result, $arguments) {
    if ($user->isAdministrator()) {
        return true;
    }
});
```

Similar to the `before` method, if the `after` closure returns a non-null result that result will be considered the result of the authorization check.

Inline Authorization

Occasionally, you may wish to determine if the currently authenticated user is authorized to perform a given action without writing a dedicate gate that corresponds to the action. Laravel allows you to perform these types of "inline" authorization checks via the `Gate::allowIf` and `Gate::denyIf` methods:

```
use Illuminate\Support\Facades\Auth;

Gate::allowIf(fn ($user) => $user->isAdministrator());

Gate::denyIf(fn ($user) => $user->banned());
```

If the action is not authorized or if no user is currently authenticated, Laravel will automatically throw an `Illuminate\Auth\Access\AuthorizationException` exception. Instances of `AuthorizationException` are automatically converted to a 403 HTTP response by Laravel's exception handler:

Creating Policies

Generating Policies

Policies are classes that organize authorization logic around a particular model or resource. For example, if your application is a blog, you may have a `App\Models\Post` model and a corresponding `App\Policies\PostPolicy` to authorize user actions such as creating or updating posts.

You may generate a policy using the `make:policy` Artisan command. The generated policy will be placed in the `app/Policies` directory. If this directory does not exist in your application, Laravel will create it for you:

```
php artisan make:policy PostPolicy
```

The `make:policy` command will generate an empty policy class. If you would like to generate a class with example policy methods related to viewing, creating, updating, and deleting the resource, you may provide a `--model` option when executing the command:

```
php artisan make:policy PostPolicy --model=Post
```

Registering Policies

Once the policy class has been created, it needs to be registered. Registering policies is how we can inform Laravel which policy to use when authorizing actions against a given model type.

The `App\Providers\AuthServiceProvider` included with fresh Laravel applications contains a `policies` property which maps your Eloquent models to their corresponding policies. Registering a policy will instruct Laravel which policy to utilize when authorizing actions against a given Eloquent model:

```php
<?php

namespace App\Providers;

use App\Models\Post;
use App\Policies\PostPolicy;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as
ServiceProvider;
use Illuminate\Support\Facades\Gate;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        Post::class => PostPolicy::class,
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @return void
     */
    public function boot()
    {
        $this->registerPolicies();

        //
    }
}
```

Policy Auto-Discovery

Instead of manually registering model policies, Laravel can automatically discover policies as long as the model and policy follow standard Laravel naming conventions. Specifically, the policies must be in a `Policies` directory at or above the directory that contains your models. So, for example, the models may be placed in the `app/Models` directory while the policies may be placed in the `app/Policies` directory. In this situation, Laravel will check for policies in `app/Models/Policies` then `app/Policies`. In addition, the policy name must match the model name and have a `Policy` suffix. So, a `User` model would correspond

to a `UserPolicy` policy class.

If you would like to define your own policy discovery logic, you may register a custom policy discovery callback using the `Gate::guessPolicyNamesUsing` method. Typically, this method should be called from the `boot` method of your application's `AuthServiceProvider`:

```
use Illuminate\Support\Facades\Gate;

Gate::guessPolicyNamesUsing(function ($modelClass) {
    // Return the name of the policy class for the given model...
});
```

{note} Any policies that are explicitly mapped in your `AuthServiceProvider` will take precedence over any potentially auto-discovered policies.

Writing Policies

Policy Methods

Once the policy class has been registered, you may add methods for each action it authorizes. For example, let's define an `update` method on our `PostPolicy` which determines if a given `App\Models\User` can update a given `App\Models\Post` instance.

The `update` method will receive a `User` and a `Post` instance as its arguments, and should return `true` or `false` indicating whether the user is authorized to update the given `Post`. So, in this example, we will verify that the user's `id` matches the `user_id` on the post:

```
<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Post  $post
     * @return bool
     */
    public function update(User $user, Post $post)
    {
        return $user->id === $post->user_id;
    }
}
```

You may continue to define additional methods on the policy as needed for the various actions it authorizes. For example, you might define `view` or `delete` methods to authorize various `Post` related actions, but remember you are free to give your policy methods any name you like.

If you used the `--model` option when generating your policy via the Artisan console, it will already contain methods for the `viewAny`, `view`, `create`, `update`, `delete`, `restore`, and `forceDelete` actions.

{tip} All policies are resolved via the Laravel service container, allowing you to type-hint any needed dependencies in the policy's constructor to have them automatically injected.

Policy Responses

So far, we have only examined policy methods that return simple boolean values. However, sometimes you may wish to return a more detailed response, including an error message. To do so, you may return an `Illuminate\Auth\Access\Response` instance from your policy method:

```
use App\Models\Post;
use App\Models\User;
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\Models\User  $user
 * @param  \App\Models\Post  $post
 * @return \Illuminate\Auth\Access\Response
 */
public function update(User $user, Post $post)
{
    return $user->id === $post->user_id
                ? Response::allow()
                : Response::deny('You do not own this post.');
}
```

When returning an authorization response from your policy, the `Gate::allows` method will still return a simple boolean value; however, you may use the `Gate::inspect` method to get the full authorization response returned by the gate:

```
use Illuminate\Support\Facades\Gate;

$response = Gate::inspect('update', $post);

if ($response->allowed()) {
    // The action is authorized...
} else {
    echo $response->message();
}
```

When using the `Gate::authorize` method, which throws an `AuthorizationException` if the action is not authorized, the error message provided by the authorization response will be propagated to the HTTP response:

```
Gate::authorize('update', $post);

// The action is authorized...
```

Methods Without Models

Some policy methods only receive an instance of the currently authenticated user. This situation is most common when authorizing `create` actions. For example, if you are creating a blog, you may wish to determine if a user is authorized to create any posts at all. In these situations, your policy method should only expect to receive a user instance:

```
/**
 * Determine if the given user can create posts.
 *
 * @param  \App\Models\User  $user
 * @return bool
 */
public function create(User $user)
{
    return $user->role == 'writer';
}
```

Guest Users

By default, all gates and policies automatically return `false` if the incoming HTTP request was not initiated by an authenticated user. However, you may allow these authorization checks to pass through to your gates and policies by declaring an "optional" type-hint or supplying a `null` default value for the user argument definition:

```
<?php

namespace App\Policies;

use App\Models\Post;
use App\Models\User;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\Models\User  $user
     * @param  \App\Models\Post  $post
     * @return bool
     */
    public function update(?User $user, Post $post)
    {
        return optional($user)->id === $post->user_id;
    }
}
```

Policy Filters

For certain users, you may wish to authorize all actions within a given policy. To accomplish this, define a `before` method on the policy. The `before` method will be executed before any other methods on the policy, giving you an opportunity to authorize the action before the intended policy method is actually called. This feature is most commonly used for authorizing application administrators to perform any action:

```
use App\Models\User;

/**
 * Perform pre-authorization checks.
 *
 * @param  \App\Models\User  $user
 * @param  string  $ability
 * @return void|bool
 */
public function before(User $user, $ability)
{
    if ($user->isAdministrator()) {
        return true;
    }
}
```

If you would like to deny all authorization checks for a particular type of user then you may return `false` from the `before` method. If `null` is returned, the authorization check will fall through to the policy method.

{note} The `before` method of a policy class will not be called if the class doesn't contain a method with a name matching the name of the ability being checked.

Authorizing Actions Using Policies

Via The User Model

The `App\Models\User` model that is included with your Laravel application includes two helpful methods for authorizing actions: `can` and `cannot`. The `can` and `cannot` methods receive the name of the action you wish to authorize and the relevant model. For example, let's determine if a user is authorized to update a given `App\Models\Post` model. Typically, this will be done within a controller method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Update the given post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Post  $post
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Post $post)
    {
        if ($request->user()->cannot('update', $post)) {
            abort(403);
        }

        // Update the post...
    }
}
```

If a policy is registered for the given model, the `can` method will automatically call the appropriate policy and return the boolean result. If no policy is registered for the model, the `can` method will attempt to call the closure-based Gate matching the given action name.

Actions That Don't Require Models

Remember, some actions may correspond to policy methods like `create` that do not require a model instance. In these situations, you may pass a class name to the `can` method. The class name will be used to determine which policy to use when authorizing the action:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Create a post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        if ($request->user()->cannot('create', Post::class)) {
            abort(403);
        }

        // Create the post...
    }
}
```

Via Controller Helpers

In addition to helpful methods provided to the `App\Models\User` model, Laravel provides a helpful `authorize` method to any of your controllers which extend the `App\Http\Controllers\Controller` base class.

Like the `can` method, this method accepts the name of the action you wish to authorize and the relevant model. If the action is not authorized, the `authorize` method will throw an `Illuminate\Auth\Access\AuthorizationException` exception which the Laravel exception handler will automatically convert to an HTTP response with a 403 status code:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Post  $post
     * @return \Illuminate\Http\Response
     *
     * @throws \Illuminate\Auth\Access\AuthorizationException
     */
    public function update(Request $request, Post $post)
    {
        $this->authorize('update', $post);

        // The current user can update the blog post...
    }
}
```

Actions That Don't Require Models

As previously discussed, some policy methods like `create` do not require a model instance. In these situations, you should pass a class name to the `authorize` method. The class name will be used to determine which policy to use when authorizing the action:

```
use App\Models\Post;
use Illuminate\Http\Request;

/**
 * Create a new blog post.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function create(Request $request)
{
    $this->authorize('create', Post::class);

    // The current user can create blog posts...
}
```

Authorizing Resource Controllers

If you are utilizing resource controllers, you may make use of the `authorizeResource` method in your controller's constructor. This method will attach the appropriate `can` middleware definitions to the resource controller's methods.

The `authorizeResource` method accepts the model's class name as its first argument, and the name of the route / request parameter that will contain the model's ID as its second argument. You should ensure your resource controller is created using the `--model` flag so that it has the required method signatures and type hints:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Create the controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->authorizeResource(Post::class, 'post');
    }
}
```

The following controller methods will be mapped to their corresponding policy method. When requests are routed to the given controller method, the corresponding policy method will automatically be invoked before the controller method is executed:

Controller Method	Policy Method
index	viewAny
show	view
create	create
store	create
edit	update
update	update
destroy	delete

{tip} You may use the `make:policy` command with the `--model` option to quickly generate a policy class for a given model: `php artisan make:policy PostPolicy --model=Post`.

Via Middleware

Laravel includes a middleware that can authorize actions before the incoming request even reaches your routes or controllers. By default, the `Illuminate\Auth\Middleware\Authorize` middleware is assigned the `can` key in your `App\Http\Kernel` class. Let's explore an example of using the `can` middleware to authorize that a user can update a post:

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->middleware('can:update,post');
```

In this example, we're passing the `can` middleware two arguments. The first is the name of the action we wish to authorize and the second is the route parameter we wish to pass to the policy method. In this case, since we are using implicit model binding, a `App\Models\Post` model will be passed to the policy method. If the user is not authorized to perform the given action, an HTTP response with a 403 status code will be returned by the middleware.

For convenience, you may also attach the `can` middleware to your route using the `can` method:

```
use App\Models\Post;

Route::put('/post/{post}', function (Post $post) {
    // The current user may update the post...
})->can('update', 'post');
```

Actions That Don't Require Models

Again, some policy methods like `create` do not require a model instance. In these situations, you may pass a class name to the middleware. The class name will be used to determine which policy to use when authorizing the action:

```
Route::post('/post', function () {
    // The current user may create posts...
})->middleware('can:create,App\Models\Post');
```

Specifying the entire class name within a string middleware definition can become cumbersome. For that reason, you may choose to attach the `can` middleware to your route using the `can` method:

```
use App\Models\Post;

Route::post('/post', function () {
    // The current user may create posts...
})->can('create', Post::class);
```

Via Blade Templates

When writing Blade templates, you may wish to display a portion of the page only if the user is authorized to perform a given action. For example, you may wish to show an update form for a blog post only if the user can actually update the post. In this situation, you may use the `@can` and `@cannot` directives:

```
@can('update', $post)
    <!-- The current user can update the post... -->
@elsecan('create', App\Models\Post::class)
    <!-- The current user can create new posts... -->
@else
    <!-- ... -->
@endcan

@cannot('update', $post)
    <!-- The current user cannot update the post... -->
@elsecannot('create', App\Models\Post::class)
    <!-- The current user cannot create new posts... -->
@endcannot
```

These directives are convenient shortcuts for writing `@if` and `@unless` statements. The `@can` and `@cannot` statements above are equivalent to the following statements:

```
@if (Auth::user()->can('update', $post))
    <!-- The current user can update the post... -->
@endif

@unless (Auth::user()->can('update', $post))
    <!-- The current user cannot update the post... -->
@endunless
```

You may also determine if a user is authorized to perform any action from a given array of actions. To accomplish this, use the `@canany` directive:

```
@canany(['update', 'view', 'delete'], $post)
    <!-- The current user can update, view, or delete the post... -->
@elsecanany(['create'], \App\Models\Post::class)
    <!-- The current user can create a post... -->
@endcanany
```

Actions That Don't Require Models

Like most of the other authorization methods, you may pass a class name to the `@can` and `@cannot` directives if the action does not require a model instance:

```
@can('create', App\Models\Post::class)
    <!-- The current user can create posts... -->
@endcan

@cannot('create', App\Models\Post::class)
    <!-- The current user can't create posts... -->
@endcannot
```

Supplying Additional Context

When authorizing actions using policies, you may pass an array as the second argument to the various authorization functions and helpers. The first element in the array will be used to determine which policy should be invoked, while the rest of the array elements are passed as parameters to the policy method and can be used for additional context when making authorization decisions. For example, consider the following `PostPolicy` method definition

which contains an additional `$category` parameter:

```
/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\Models\User  $user
 * @param  \App\Models\Post  $post
 * @param  int  $category
 * @return bool
 */
public function update(User $user, Post $post, int $category)
{
    return $user->id === $post->user_id &&
           $user->canUpdateCategory($category);
}
```

When attempting to determine if the authenticated user can update a given post, we can invoke this policy method like so:

```
/**
 * Update the given blog post.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \App\Models\Post  $post
 * @return \Illuminate\Http\Response
 *
 * @throws \Illuminate\Auth\Access\AuthorizationException
 */
public function update(Request $request, Post $post)
{
    $this->authorize('update', [$post, $request->category]);

    // The current user can update the blog post...
}
```

Blade Templates

- Introduction
- Displaying Data
    - HTML Entity Encoding

    - Blade & JavaScript Frameworks
- Blade Directives
    - If Statements
    - Switch Statements
    - Loops
    - The Loop Variable
    - Conditional Classes
    - Including Subviews
    - The `@once` Directive
    - Raw PHP
    - Comments
- Components
    - Rendering Components
    - Passing Data To Components
    - Component Attributes
    - Reserved Keywords
    - Slots
    - Inline Component Views
    - Anonymous Components
    - Dynamic Components
    - Manually Registering Components
- Building Layouts
    - Layouts Using Components
    - Layouts Using Template Inheritance
- Forms
    - CSRF Field
    - Method Field
    - Validation Errors
- Stacks
- Service Injection
- Extending Blade
    - Custom Echo Handlers
    - Custom If Statements

Introduction

Blade is the simple, yet powerful templating engine that is included with Laravel. Unlike some PHP templating engines, Blade does not restrict you from using plain PHP code in your templates. In fact, all Blade templates are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade template files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Blade views may be returned from routes or controller using the global `view` helper. Of course, as mentioned in the documentation on views, data may be passed to the Blade view using the `view` helper's second argument:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'Finn']);
});
```

{tip} Want to take your Blade templates to the next level and build dynamic interfaces with ease? Check out Laravel Livewire.

Displaying Data

You may display data that is passed to your Blade views by wrapping the variable in curly braces. For example, given the following route:

```
Route::get('/', function () {
    return view('welcome', ['name' => 'Samantha']);
});
```

You may display the contents of the `name` variable like so:

```
Hello, {{ $name }}.
```

{tip} Blade's `{{ }}` echo statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

You are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

```
The current UNIX timestamp is {{ time() }}.
```

HTML Entity Encoding

By default, Blade (and the Laravel `e` helper) will double encode HTML entities. If you would like to disable double encoding, call the `Blade::withoutDoubleEncoding` method from the `boot` method of your `AppServiceProvider`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::withoutDoubleEncoding();
    }
}
```

Displaying Unescaped Data

By default, Blade `{{ }}` statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

```
Hello, {!! $name !!}.
```

{note} Be very careful when echoing content that is supplied by users of your application. You should typically use the escaped, double curly brace syntax to prevent XSS attacks when displaying user supplied data.

Blade & JavaScript Frameworks

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the `@` symbol to inform the Blade rendering engine an expression should remain untouched. For example:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

In this example, the `@` symbol will be removed by Blade; however, `{{ name }}` expression will remain untouched by the Blade engine, allowing it to be rendered by your JavaScript framework.

The `@` symbol may also be used to escape Blade directives:

```
{{-- Blade template --}}
@@if()

<!-- HTML output -->
@if()
```

Rendering JSON

Sometimes you may pass an array to your view with the intention of rendering it as JSON in order to initialize a JavaScript variable. For example:

```
<script>
    var app = <?php echo json_encode($array); ?>;
</script>
```

However, instead of manually calling `json_encode`, you may use the `Illuminate\Support\Js::from` method directive. The `from` method accepts the same arguments as PHP's `json_encode` function; however, it will ensure that the resulting JSON is properly escaped for inclusion within HTML quotes. The `from` method will return a string `JSON.parse` JavaScript statement that will convert the given object or array into a valid JavaScript object:

```
<script>
    var app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

The latest versions of the Laravel application skeleton include a `Js` facade, which provides

convenient access to this functionality within your Blade templates:

```
<script>
    var app = {{ Js::from($array) }};
</script>
```

{note} You should only use the `Js::from` method to render existing variables as JSON. The Blade templating is based on regular expressions and attempts to pass a complex expression to the directive may cause unexpected failures.

The `@verbatim` Directive

If you are displaying JavaScript variables in a large portion of your template, you may wrap the HTML in the `@verbatim` directive so that you do not have to prefix each Blade echo statement with an `@` symbol:

```
@verbatim
    <div class="container">
        Hello, {{ name }}.
    </div>
@endverbatim
```

Blade Directives

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures while also remaining familiar to their PHP counterparts.

If Statements

You may construct `if` statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

For convenience, Blade also provides an `@unless` directive:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

In addition to the conditional directives already discussed, the `@isset` and `@empty` directives may be used as convenient shortcuts for their respective PHP functions:

```
@isset($records)
    // $records is defined and is not null...
@endisset

@empty($records)
    // $records is "empty"...
@endempty
```

Authentication Directives

The `@auth` and `@guest` directives may be used to quickly determine if the current user is authenticated or is a guest:

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

If needed, you may specify the authentication guard that should be checked when using the `@auth` and `@guest` directives:

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

Environment Directives

You may check if the application is running in the production environment using the `@production` directive:

```
@production
    // Production specific content...
@endproduction
```

Or, you may determine if the application is running in a specific environment using the `@env` directive:

```
@env('staging')
    // The application is running in "staging"...
@endenv

@env(['staging', 'production'])
    // The application is running in "staging" or "production"...
@endenv
```

Section Directives

You may determine if a template inheritance section has content using the `@hasSection` directive:

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

You may use the `sectionMissing` directive to determine if a section does not have content:

```
@sectionMissing('navigation')
    <div class="pull-right">
        @include('default-navigation')
    </div>
@endif
```

Switch Statements

Switch statements can be constructed using the `@switch`, `@case`, `@break`, `@default` and `@endswitch` directives:

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

{tip} When looping, you may use the loop variable to gain valuable information about the loop, such as whether you are in the first or last iteration through the loop.

When using loops you may also end the loop or skip the current iteration using the `@continue` and `@break` directives:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

You may also include the continuation or break condition within the directive declaration:

```
@foreach ($users as $user)
    @continue($user->type == 1)

    <li>{{ $user->name }}</li>

    @break($user->number == 5)
@endforeach
```

The Loop Variable

When looping, a `$loop` variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop:

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

If you are in a nested loop, you may access the parent loop's `$loop` variable via the `parent` property:

```
@foreach ($users as $user)
    @foreach ($user->posts as $post)
        @if ($loop->parent->first)
            This is the first iteration of the parent loop.
        @endif
    @endforeach
@endforeach
```

The `$loop` variable also contains a variety of other useful properties:

Property	Description
`$loop->index`	The index of the current loop iteration (starts at 0).
`$loop->iteration`	The current loop iteration (starts at 1).
`$loop->remaining`	The iterations remaining in the loop.
`$loop->count`	The total number of items in the array being iterated.
`$loop->first`	Whether this is the first iteration through the loop.
`$loop->last`	Whether this is the last iteration through the loop.
`$loop->even`	Whether this is an even iteration through the loop.
`$loop->odd`	Whether this is an odd iteration through the loop.
`$loop->depth`	The nesting level of the current loop.
`$loop->parent`	When in a nested loop, the parent's loop variable.

Conditional Classes

The `@class` directive conditionally compiles a CSS class string. The directive accepts an array of classes where the array key contains the class or classes you wish to add, while the value is a boolean expression. If the array element has a numeric key, it will always be included in the rendered class list:

```
@php
    $isActive = false;
    $hasError = true;
@endphp

<span @class([
    'p-4',
    'font-bold' => $isActive,
    'text-gray-500' => ! $isActive,
    'bg-red' => $hasError,
])></span>

<span class="p-4 text-gray-500 bg-red"></span>
```

Including Subviews

{tip} While you're free to use the `@include` directive, Blade components provide similar functionality and offer several benefits over the `@include` directive such as data and attribute binding.

Blade's `@include` directive allows you to include a Blade view from within another view. All variables that are available to the parent view will be made available to the included view:

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of additional data that should be made available to the included view:

```
@include('view.name', ['status' => 'complete'])
```

If you attempt to `@include` a view which does not exist, Laravel will throw an error. If you would like to include a view that may or may not be present, you should use the `@includeIf` directive:

```
@includeIf('view.name', ['status' => 'complete'])
```

If you would like to `@include` a view if a given boolean expression evaluates to `true` or `false`, you may use the `@includeWhen` and `@includeUnless` directives:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])

@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

To include the first view that exists from a given array of views, you may use the `includeFirst` directive:

```
@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])
```

{note} You should avoid using the `__DIR__` and `__FILE__` constants in your Blade views, since they will refer to the location of the cached, compiled view.

Rendering Views For Collections

You may combine loops and includes into one line with Blade's `@each` directive:

```
@each('view.name', $jobs, 'job')
```

The `@each` directive's first argument is the view to render for each element in the array or collection. The second argument is the array or collection you wish to iterate over, while the third argument is the variable name that will be assigned to the current iteration within the view. So, for example, if you are iterating over an array of `jobs`, typically you will want to access each job as a `job` variable within the view. The array key for the current iteration will be available as the `key` variable within the view.

You may also pass a fourth argument to the `@each` directive. This argument determines the view that will be rendered if the given array is empty.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

{note} Views rendered via `@each` do not inherit the variables from the parent view. If the child view requires these variables, you should use the `@foreach` and `@include` directives instead.

The `@once` Directive

The `@once` directive allows you to define a portion of the template that will only be evaluated once per rendering cycle. This may be useful for pushing a given piece of JavaScript into the page's header using stacks. For example, if you are rendering a given component within a loop, you may wish to only push the JavaScript to the header the first time the component is rendered:

```
@once
    @push('scripts')
        <script>
            // Your custom JavaScript...
        </script>
    @endpush
@endonce
```

Raw PHP

In some situations, it's useful to embed PHP code into your views. You can use the Blade @php directive to execute a block of plain PHP within your template:

```
@php
    $counter = 1;
@endphp
```

Comments

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Components

Components and slots provide similar benefits to sections, layouts, and includes; however, some may find the mental model of components and slots easier to understand. There are two approaches to writing components: class based components and anonymous components.

To create a class based component, you may use the `make:component` Artisan command. To illustrate how to use components, we will create a simple `Alert` component. The `make:component` command will place the component in the `App\View\Components` directory:

```
php artisan make:component Alert
```

The `make:component` command will also create a view template for the component. The view will be placed in the `resources/views/components` directory. When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory, so no further component registration is typically required.

You may also create components within subdirectories:

```
php artisan make:component Forms/Input
```

The command above will create an `Input` component in the `App\View\Components\Forms` directory and the view will be placed in the `resources/views/components/forms` directory.

Manually Registering Package Components

When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory.

However, if you are building a package that utilizes Blade components, you will need to manually register your component class and its HTML tag alias. You should typically register your components in the `boot` method of your package's service provider:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap your package's services.
 */
public function boot()
{
    Blade::component('package-alert', Alert::class);
}
```

Once your component has been registered, it may be rendered using its tag alias:

```
<x-package-alert/>
```

Alternatively, you may use the `componentNamespace` method to autoload component classes by convention. For example, a `Nightshade` package might have `Calendar` and `ColorPicker` components that reside within the `Package\Views\Components` namespace:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::componentNamespace('Nightshade\\Views\\Components',
'nightshade');
}
```

This will allow the usage of package components by their vendor namespace using the `package-name::` syntax:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade will automatically detect the class that's linked to this component by pascal-casing the component name. Subdirectories are also supported using "dot" notation.

Rendering Components

To display a component, you may use a Blade component tag within one of your Blade templates. Blade component tags start with the string `x-` followed by the kebab case name of the component class:

```
<x-alert/>

<x-user-profile/>
```

If the component class is nested deeper within the `App\View\Components` directory, you may use the `.` character to indicate directory nesting. For example, if we assume a component is located at `App\View\Components\Inputs\Button.php`, we may render it like so:

```
<x-inputs.button/>
```

Passing Data To Components

You may pass data to Blade components using HTML attributes. Hard-coded, primitive values may be passed to the component using simple HTML attribute strings. PHP expressions and variables should be passed to the component via attributes that use the `:` character as a prefix:

```
<x-alert type="error" :message="$message"/>
```

You should define the component's required data in its class constructor. All public properties on a component will automatically be made available to the component's view. It is not necessary to pass the data to the view from the component's `render` method:

```
<?php
```

```
namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component
{
    /**
     * The alert type.
     *
     * @var string
     */
    public $type;

    /**
     * The alert message.
     *
     * @var string
     */
    public $message;

    /**
     * Create the component instance.
     *
     * @param  string  $type
     * @param  string  $message
     * @return void
     */
    public function __construct($type, $message)
    {
        $this->type = $type;
        $this->message = $message;
    }

    /**
     * Get the view / contents that represent the component.
     *
     * @return \Illuminate\View\View|\Closure|string
     */
    public function render()
    {
        return view('components.alert');
    }
}
```

When your component is rendered, you may display the contents of your component's

public variables by echoing the variables by name:

```
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

Casing

Component constructor arguments should be specified using `camelCase`, while `kebab-case` should be used when referencing the argument names in your HTML attributes. For example, given the following component constructor:

```
/**
 * Create the component instance.
 *
 * @param  string  $alertType
 * @return void
 */
public function __construct($alertType)
{
    $this->alertType = $alertType;
}
```

The `$alertType` argument may be provided to the component like so:

```
<x-alert alert-type="danger" />
```

Escaping Attribute Rendering

Since some JavaScript frameworks such as Alpine.js also use colon-prefixed attributes, you may use a double colon (`::`) prefix to inform Blade that the attribute is not a PHP expression. For example, given the following component:

```
<x-button ::class="{ danger: isDeleting }">
    Submit
</x-button>
```

The following HTML will be rendered by Blade:

```
<button :class="{ danger: isDeleting }">
    Submit
</button>
```

Component Methods

In addition to public variables being available to your component template, any public methods on the component may be invoked. For example, imagine a component that has an `isSelected` method:

```
/**
 * Determine if the given option is the currently selected option.
 *
 * @param  string  $option
 * @return bool
 */
public function isSelected($option)
{
    return $option === $this->selected;
}
```

You may execute this method from your component template by invoking the variable matching the name of the method:

```
<option {{ $isSelected($value) ? 'selected="selected"' : '' }} value="{{
$value }}">
    {{ $label }}
</option>
```

Accessing Attributes & Slots Within Component Classes

Blade components also allow you to access the component name, attributes, and slot inside the class's render method. However, in order to access this data, you should return a closure from your component's `render` method. The closure will receive a `$data` array as its only argument. This array will contain several elements that provide information about the component:

```
/**
 * Get the view / contents that represent the component.
 *
 * @return \Illuminate\View\View|\Closure|string
 */
public function render()
{
    return function (array $data) {
        // $data['componentName'];
        // $data['attributes'];
        // $data['slot'];

        return '<div>Components content</div>';
    };
}
```

The `componentName` is equal to the name used in the HTML tag after the `x-` prefix. So `<x-alert />`'s `componentName` will be `alert`. The `attributes` element will contain all of the attributes that were present on the HTML tag. The `slot` element is an `Illuminate\Support\HtmlString` instance with the contents of the component's slot.

The closure should return a string. If the returned string corresponds to an existing view, that view will be rendered; otherwise, the returned string will be evaluated as an inline Blade view.

Additional Dependencies

If your component requires dependencies from Laravel's service container, you may list them before any of the component's data attributes and they will automatically be injected by the container:

```
use App\Services\AlertCreator

/**
 * Create the component instance.
 *
 * @param  \App\Services\AlertCreator  $creator
 * @param  string  $type
 * @param  string  $message
 * @return void
 */
public function __construct(AlertCreator $creator, $type, $message)
{
    $this->creator = $creator;
    $this->type = $type;
    $this->message = $message;
}
```

Hiding Attributes / Methods

If you would like to prevent some public methods or properties from being exposed as variables to your component template, you may add them to an `$except` array property on your component:

```php
<?php

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component
{
    /**
     * The alert type.
     *
     * @var string
     */
    public $type;

    /**
     * The properties / methods that should not be exposed to the
component template.
     *
     * @var array
     */
    protected $except = ['type'];
}
```

Component Attributes

We've already examined how to pass data attributes to a component; however, sometimes you may need to specify additional HTML attributes, such as `class`, that are not part of the data required for a component to function. Typically, you want to pass these additional attributes down to the root element of the component template. For example, imagine we want to render an `alert` component like so:

```
<x-alert type="error" :message="$message" class="mt-4"/>
```

All of the attributes that are not part of the component's constructor will automatically be added to the component's "attribute bag". This attribute bag is automatically made available to the component via the `$attributes` variable. All of the attributes may be rendered within the component by echoing this variable:

```
<div {{ $attributes }}>
    <!-- Component content -->
</div>
```

{note} Using directives such as `@env` within component tags is not supported at this time. For example, `<x-alert :live="@env('production')"/>` will not be compiled.

Default / Merged Attributes

Sometimes you may need to specify default values for attributes or merge additional values into some of the component's attributes. To accomplish this, you may use the attribute bag's `merge` method. This method is particularly useful for defining a set of default CSS classes that should always be applied to a component:

```
<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
    {{ $message }}
</div>
```

If we assume this component is utilized like so:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

The final, rendered HTML of the component will appear like the following:

```
<div class="alert alert-error mb-4">
    <!-- Contents of the $message variable -->
</div>
```

Conditionally Merge Classes

Sometimes you may wish to merge classes if a given condition is `true`. You can accomplish this via the `class` method, which accepts an array of classes where the array key contains the class or classes you wish to add, while the value is a boolean expression. If the array element has a numeric key, it will always be included in the rendered class list:

```
<div {{ $attributes->class(['p-4', 'bg-red' => $hasError]) }}>
    {{ $message }}
</div>
```

If you need to merge other attributes onto your component, you can chain the `merge` method onto the `class` method:

```
<button {{ $attributes->class(['p-4'])->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

{tip} If you need to conditionally compile classes on other HTML elements that shouldn't receive merged attributes, you can use the `@class` directive.

Non-Class Attribute Merging

When merging attributes that are not `class` attributes, the values provided to the `merge` method will be considered the "default" values of the attribute. However, unlike the `class` attribute, these attributes will not be merged with injected attribute values. Instead, they will be overwritten. For example, a `button` component's implementation may look like the following:

```
<button {{ $attributes->merge(['type' => 'button']) }}>
    {{ $slot }}
</button>
```

To render the button component with a custom `type`, it may be specified when consuming the component. If no type is specified, the `button` type will be used:

```
<x-button type="submit">
    Submit
</x-button>
```

The rendered HTML of the `button` component in this example would be:

```
<button type="submit">
    Submit
</button>
```

If you would like an attribute other than `class` to have its default value and injected values joined together, you may use the `prepends` method. In this example, the `data-controller` attribute will always begin with `profile-controller` and any additional injected `data-controller` values will be placed after this default value:

```
<div {{ $attributes->merge(['data-controller' =>
$attributes->prepends('profile-controller')]) }}>
    {{ $slot }}
</div>
```

Retrieving & Filtering Attributes

You may filter attributes using the `filter` method. This method accepts a closure which should return `true` if you wish to retain the attribute in the attribute bag:

```
{{ $attributes->filter(fn ($value, $key) => $key == 'foo') }}
```

For convenience, you may use the `whereStartsWith` method to retrieve all attributes whose keys begin with a given string:

```
{{ $attributes->whereStartsWith('wire:model') }}
```

Conversely, the `whereDoesntStartWith` method may be used to exclude all attributes whose keys begin with a given string:

```
{{ $attributes->whereDoesntStartWith('wire:model') }}
```

Using the `first` method, you may render the first attribute in a given attribute bag:

```
{{ $attributes->whereStartsWith('wire:model')->first() }}
```

If you would like to check if an attribute is present on the component, you may use the `has` method. This method accepts the attribute name as its only argument and returns a boolean indicating whether or not the attribute is present:

```
@if ($attributes->has('class'))
    <div>Class attribute is present</div>
@endif
```

You may retrieve a specific attribute's value using the `get` method:

```
{{ $attributes->get('class') }}
```

Reserved Keywords

By default, some keywords are reserved for Blade's internal use in order to render components. The following keywords cannot be defined as public properties or method names within your components:

- `data` - `render` - `resolveView` - `shouldRender` - `view` - `withAttributes` - `withName`

Slots

You will often need to pass additional content to your component via "slots". Component slots are rendered by echoing the `$slot` variable. To explore this concept, let's imagine that an `alert` component has the following markup:

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

We may pass content to the `slot` by injecting content into the component:

```
<x-alert>
    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Sometimes a component may need to render multiple different slots in different locations within the component. Let's modify our alert component to allow for the injection of a "title" slot:

```
<!-- /resources/views/components/alert.blade.php -->

<span class="alert-title">{{ $title }}</span>

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

You may define the content of the named slot using the `x-slot` tag. Any content not within an explicit `x-slot` tag will be passed to the component in the `$slot` variable:

```
<x-alert>
    <x-slot name="title">
        Server Error
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Scoped Slots

If you have used a JavaScript framework such as Vue, you may be familiar with "scoped slots", which allow you to access data or methods from the component within your slot. You may achieve similar behavior in Laravel by defining public methods or properties on your component and accessing the component within your slot via the `$component` variable. In this example, we will assume that the `x-alert` component has a public `formatAlert` method defined on its component class:

```
<x-alert>
    <x-slot name="title">
        {{ $component->formatAlert('Server Error') }}
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Slot Attributes

Like Blade components, you may assign additional attributes to slots such as CSS class names:

```
<x-card class="shadow-sm">
    <x-slot name="heading" class="font-bold">
        Heading
    </x-slot>

    Content

    <x-slot name="footer" class="text-sm">
        Footer
    </x-slot>
</x-card>
```

To interact with slot attributes, you may access the `attributes` property of the slot's variable. For more information on how to interact with attributes, please consult the documentation on component attributes:

```
@props([
    'heading',
    'footer',
])

<div {{ $attributes->class(['border']) }}>
    <h1 {{ $heading->attributes->class(['text-lg']) }}>
        {{ $heading }}
    </h1>

    {{ $slot }}

    <footer {{ $footer->attributes->class(['text-gray-700']) }}>
        {{ $footer }}
    </footer>
</div>
```

Inline Component Views

For very small components, it may feel cumbersome to manage both the component class and the component's view template. For this reason, you may return the component's markup directly from the `render` method:

```
/**
 * Get the view / contents that represent the component.
 *
 * @return \Illuminate\View\View|\Closure|string
 */
public function render()
{
    return <<<'blade'
        <div class="alert alert-danger">
            {{ $slot }}
        </div>
    blade;
}
```

Generating Inline View Components

To create a component that renders an inline view, you may use the `inline` option when executing the `make:component` command:

```
php artisan make:component Alert --inline
```

Anonymous Components

Similar to inline components, anonymous components provide a mechanism for managing a component via a single file. However, anonymous components utilize a single view file and have no associated class. To define an anonymous component, you only need to place a Blade template within your `resources/views/components` directory. For example, assuming you have defined a component at `resources/views/components/alert.blade.php`, you may simply render it like so:

```
<x-alert/>
```

You may use the `.` character to indicate if a component is nested deeper inside the `components` directory. For example, assuming the component is defined at `resources/views/components/inputs/button.blade.php`, you may render it like so:

```
<x-inputs.button/>
```

Anonymous Index Components

Sometimes, when a component is made up of many Blade templates, you may wish to group the given component's templates within a single directory. For example, imagine an "accordion" component with the following directory structure:

```
/resources/views/components/accordion.blade.php
/resources/views/components/accordion/item.blade.php
```

This directory structure allows you to render the accordion component and its item like so:

```
<x-accordion>
    <x-accordion.item>
        ...
    </x-accordion.item>
</x-accordion>
```

However, in order to render the accordion component via `x-accordion`, we were forced to place the "index" accordion component template in the `resources/views/components` directory instead of nesting it within the `accordion` directory with the other accordion related templates.

Thankfully, Blade allows you to place an `index.blade.php` file within a component's template directory. When an `index.blade.php` template exists for the component, it will be rendered as the "root" node of the component. So, we can continue to use the same Blade syntax given in the example above; however, we will adjust our directory structure like so:

```
/resources/views/components/accordion/index.blade.php
/resources/views/components/accordion/item.blade.php
```

Data Properties / Attributes

Since anonymous components do not have any associated class, you may wonder how you may differentiate which data should be passed to the component as variables and which attributes should be placed in the component's attribute bag.

You may specify which attributes should be considered data variables using the `@props` directive at the top of your component's Blade template. All other attributes on the component will be available via the component's attribute bag. If you wish to give a data variable a default value, you may specify the variable's name as the array key and the default value as the array value:

```
<!-- /resources/views/components/alert.blade.php -->

@props(['type' => 'info', 'message'])

<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
    {{ $message }}
</div>
```

Given the component definition above, we may render the component like so:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

Accessing Parent Data

Sometimes you may want to access data from a parent component inside a child component. In these cases, you may use the `@aware` directive. For example, imagine we are building a complex menu component consisting of a parent `<x-menu>` and child `<x-menu.item>`:

```
<x-menu color="purple">
    <x-menu.item>...</x-menu.item>
    <x-menu.item>...</x-menu.item>
</x-menu>
```

The `<x-menu>` component may have an implementation like the following:

```
<!-- /resources/views/components/menu/index.blade.php -->

@props(['color' => 'gray'])

<ul {{ $attributes->merge(['class' => 'bg-'.$color.'-200']) }}>
    {{ $slot }}
</ul>
```

Because the `color` prop was only passed into the parent (`<x-menu>`), it won't be available inside `<x-menu.item>`. However, if we use the `@aware` directive, we can make it available inside `<x-menu.item>` as well:

```
<!-- /resources/views/components/menu/item.blade.php -->

@aware(['color' => 'gray'])

<li {{ $attributes->merge(['class' => 'text-'.$color.'-800']) }}>
    {{ $slot }}
</li>
```

Dynamic Components

Sometimes you may need to render a component but not know which component should be rendered until runtime. In this situation, you may use Laravel's built-in `dynamic-component` component to render the component based on a runtime value or variable:

```
<x-dynamic-component :component="$componentName" class="mt-4" />
```

Manually Registering Components

{note} The following documentation on manually registering components is primarily applicable to those who are writing Laravel packages that include view components. If you are not writing a package, this portion of the component documentation may not be relevant to you.

When writing components for your own application, components are automatically discovered within the `app/View/Components` directory and `resources/views/components` directory.

However, if you are building a package that utilizes Blade components or placing components in non-conventional directories, you will need to manually register your component class and its HTML tag alias so that Laravel knows where to find the component. You should typically register your components in the `boot` method of your package's service provider:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::component('package-alert', AlertComponent::class);
}
```

Once your component has been registered, it may be rendered using its tag alias:

```
<x-package-alert/>
```

Autoloading Package Components

Alternatively, you may use the `componentNamespace` method to autoload component classes by convention. For example, a `Nightshade` package might have `Calendar` and `ColorPicker` components that reside within the `Package\Views\Components` namespace:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::componentNamespace('Nightshade\\Views\\Components',
'nightshade');
}
```

This will allow the usage of package components by their vendor namespace using the `package-name::` syntax:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade will automatically detect the class that's linked to this component by pascal-casing the component name. Subdirectories are also supported using "dot" notation.

Building Layouts

Layouts Using Components

Most web applications maintain the same general layout across various pages. It would be incredibly cumbersome and hard to maintain our application if we had to repeat the entire layout HTML in every view we create. Thankfully, it's convenient to define this layout as a single Blade component and then use it throughout our application.

Defining The Layout Component

For example, imagine we are building a "todo" list application. We might define a `layout` component that looks like the following:

```
<!-- resources/views/components/layout.blade.php -->

<html>
    <head>
        <title>{{ $title ?? 'Todo Manager' }}</title>
    </head>
    <body>
        <h1>Todos</h1>
        <hr/>
        {{ $slot }}
    </body>
</html>
```

Applying The Layout Component

Once the `layout` component has been defined, we may create a Blade view that utilizes the component. In this example, we will define a simple view that displays our task list:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

Remember, content that is injected into a component will be supplied to the default `$slot` variable within our `layout` component. As you may have noticed, our `layout` also respects a `$title` slot if one is provided; otherwise, a default title is shown. We may inject a custom title from our task list view using the standard slot syntax discussed in the component documentation:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
    <x-slot name="title">
        Custom Title
    </x-slot>

    @foreach ($tasks as $task)
        {{ $task }}
    @endforeach
</x-layout>
```

Now that we have defined our layout and task list views, we just need to return the `task` view from a route:

```
use App\Models\Task;

Route::get('/tasks', function () {
    return view('tasks', ['tasks' => Task::all()]);
});
```

Layouts Using Template Inheritance

Defining A Layout

Layouts may also be created via "template inheritance". This was the primary way of building applications prior to the introduction of components.

To get started, let's take a look at a simple example. First, we will examine a page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

```
<!-- resources/views/layouts/app.blade.php -->

<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('sidebar')
            This is the master sidebar.
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

As you can see, this file contains typical HTML mark-up. However, take note of the `@section` and `@yield` directives. The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

Extending A Layout

When defining a child view, use the `@extends` Blade directive to specify which layout the child view should "inherit". Views which extend a Blade layout may inject content into the layout's sections using `@section` directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using `@yield`:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
    @@parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

In this example, the `sidebar` section is utilizing the `@@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@@parent` directive will be replaced by the content of the layout when the view is rendered.

{tip} Contrary to the previous example, this `sidebar` section ends with `@endsection` instead of `@show`. The `@endsection` directive will only define a section while `@show` will define and **immediately yield** the section.

The `@yield` directive also accepts a default value as its second parameter. This value will be rendered if the section being yielded is undefined:

```
@yield('content', 'Default content')
```

Forms

CSRF Field

Anytime you define an HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the `@csrf` Blade directive to generate the token field:

```
<form method="POST" action="/profile">
    @csrf

    ...
</form>
```

Method Field

Since HTML forms can't make `PUT`, `PATCH`, or `DELETE` requests, you will need to add a hidden `_method` field to spoof these HTTP verbs. The `@method` Blade directive can create this field for you:

```
<form action="/foo/bar" method="POST">
    @method('PUT')

    ...
</form>
```

Validation Errors

The `@error` directive may be used to quickly check if validation error messages exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid
@enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Since the `@error` directive compiles to an "if" statement, you may use the `@else` directive to render content when there is not an error for an attribute:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email') is-invalid @else
is-valid @enderror">
```

You may pass the name of a specific error bag as the second parameter to the `@error` directive to retrieve validation error messages on pages containing multiple forms:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email', 'login') is-
invalid @enderror">

@error('email', 'login')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Stacks

Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout. This can be particularly useful for specifying any JavaScript libraries required by your child views:

```
@push('scripts')
    <script src="/example.js"></script>
@endpush
```

You may push to a stack as many times as needed. To render the complete stack contents, pass the name of the stack to the `@stack` directive:

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

If you would like to prepend content onto the beginning of a stack, you should use the `@prepend` directive:

```
@push('scripts')
    This will be second...
@endpush

// Later...

@prepend('scripts')
    This will be first...
@endprepend
```

Service Injection

The `@inject` directive may be used to retrieve a service from the Laravel service container. The first argument passed to `@inject` is the name of the variable the service will be placed into, while the second argument is the class or interface name of the service you wish to resolve:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
    Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

Extending Blade

Blade allows you to define your own custom directives using the `directive` method. When the Blade compiler encounters the custom directive, it will call the provided callback with the expression that the directive contains.

The following example creates a `@datetime($var)` directive which formats a given `$var`, which should be an instance of `DateTime`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

As you can see, we will chain the `format` method onto whatever expression is passed into the directive. So, in this example, the final PHP generated by this directive will be:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

{note} After updating the logic of a Blade directive, you will need to delete all of the cached Blade views. The cached Blade views may be removed using the `view:clear` Artisan command.

Custom Echo Handlers

If you attempt to "echo" an object using Blade, the object's `__toString` method will be invoked. The `__toString` method is one of PHP's built-in "magic methods". However, sometimes you may not have control over the `__toString` method of a given class, such as when the class that you are interacting with belongs to a third-party library.

In these cases, Blade allows you to register a custom echo handler for that particular type of object. To accomplish this, you should invoke Blade's `stringable` method. The `stringable` method accepts a closure. This closure should type-hint the type of object that it is responsible for rendering. Typically, the `stringable` method should be invoked within the `boot` method of your application's `AppServiceProvider` class:

```
use Illuminate\Support\Facades\Blade;
use Money\Money;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Blade::stringable(function (Money $money) {
        return $money->formatTo('en_GB');
    });
}
```

Once your custom echo handler has been defined, you may simply echo the object in your Blade template:

```
Cost: {{ $money }}
```

Custom If Statements

Programming a custom directive is sometimes more complex than necessary when defining simple, custom conditional statements. For that reason, Blade provides a `Blade::if` method which allows you to quickly define custom conditional directives using closures. For example, let's define a custom conditional that checks the configured default "disk" for the application. We may do this in the `boot` method of our `AppServiceProvider`:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Blade::if('disk', function ($value) {
        return config('filesystems.default') === $value;
    });
}
```

Once the custom conditional has been defined, you can use it within your templates:

```
@disk('local')
    <!-- The application is using the local disk... -->
@elsedisk('s3')
    <!-- The application is using the s3 disk... -->
@else
    <!-- The application is using some other disk... -->
@enddisk

@unlessdisk('local')
    <!-- The application is not using the local disk... -->
@enddisk
```

Cache

- Introduction
- Configuration
    - Driver Prerequisites
- Cache Usage
    - Obtaining A Cache Instance
    - Retrieving Items From The Cache
    - Storing Items In The Cache
    - Removing Items From The Cache
    - The Cache Helper
- Cache Tags
    - Storing Tagged Cache Items
    - Accessing Tagged Cache Items
    - Removing Tagged Cache Items
- Atomic Locks
    - Driver Prerequisites
    - Managing Locks
    - Managing Locks Across Processes
- Adding Custom Cache Drivers
    - Writing The Driver
    - Registering The Driver
- Events

Introduction

Some of the data retrieval or processing tasks performed by your application could be CPU intensive or take several seconds to complete. When this is the case, it is common to cache the retrieved data for a time so it can be retrieved quickly on subsequent requests for the same data. The cached data is usually stored in a very fast data store such as Memcached or Redis.

Thankfully, Laravel provides an expressive, unified API for various cache backends, allowing you to take advantage of their blazing fast data retrieval and speed up your web application.

Configuration

Your application's cache configuration file is located at `config/cache.php`. In this file, you may specify which cache driver you would like to be used by default throughout your application. Laravel supports popular caching backends like Memcached, Redis, DynamoDB, and relational databases out of the box. In addition, a file based cache driver is available, while `array` and "null" cache drivers provide convenient cache backends for your automated tests.

The cache configuration file also contains various other options, which are documented within the file, so make sure to read over these options. By default, Laravel is configured to use the `file` cache driver, which stores the serialized, cached objects on the server's filesystem. For larger applications, it is recommended that you use a more robust driver such as Memcached or Redis. You may even configure multiple cache configurations for the same driver.

Driver Prerequisites

Database

When using the `database` cache driver, you will need to setup a table to contain the cache items. You'll find an example `Schema` declaration for the table below:

```
Schema::create('cache', function ($table) {
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

{tip} You may also use the `php artisan cache:table` Artisan command to generate a migration with the proper schema.

Memcached

Using the Memcached driver requires the Memcached PECL package to be installed. You may list all of your Memcached servers in the `config/cache.php` configuration file. This file already contains a `memcached.servers` entry to get you started:

```
'memcached' => [
    'servers' => [
        [
            'host' => env('MEMCACHED_HOST', '127.0.0.1'),
            'port' => env('MEMCACHED_PORT', 11211),
            'weight' => 100,
        ],
    ],
],
```

If needed, you may set the `host` option to a UNIX socket path. If you do this, the `port` option should be set to `0`:

```
'memcached' => [
    [
        'host' => '/var/run/memcached/memcached.sock',
        'port' => 0,
        'weight' => 100
    ],
],
```

Redis

Before using a Redis cache with Laravel, you will need to either install the PhpRedis PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer. Laravel Sail already includes this extension. In addition, official Laravel deployment platforms such as Laravel Forge and Laravel Vapor have the PhpRedis extension installed by default.

For more information on configuring Redis, consult its Laravel documentation page.

DynamoDB

Before using the DynamoDB cache driver, you must create a DynamoDB table to store all of the cached data. Typically, this table should be named `cache`. However, you should name the table based on the value of the `stores.dynamodb.table` configuration value within your application's `cache` configuration file.

This table should also have a string partition key with a name that corresponds to the value of the `stores.dynamodb.attributes.key` configuration item within your application's `cache` configuration file. By default, the partition key should be named `key`.

Cache Usage

Obtaining A Cache Instance

To obtain a cache store instance, you may use the `Cache` facade, which is what we will use throughout this documentation. The `Cache` facade provides convenient, terse access to the underlying implementations of the Laravel cache contracts:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\Cache;

class UserController extends Controller
{
    /**
     * Show a list of all users of the application.
     *
     * @return Response
     */
    public function index()
    {
        $value = Cache::get('key');

        //
    }
}
```

Accessing Multiple Cache Stores

Using the `Cache` facade, you may access various cache stores via the `store` method. The key passed to the `store` method should correspond to one of the stores listed in the `stores` configuration array in your `cache` configuration file:

```
$value = Cache::store('file')->get('foo');

Cache::store('redis')->put('bar', 'baz', 600); // 10 Minutes
```

Retrieving Items From The Cache

The `Cache` facade's `get` method is used to retrieve items from the cache. If the item does not exist in the cache, `null` will be returned. If you wish, you may pass a second argument to the `get` method specifying the default value you wish to be returned if the item doesn't exist:

```
$value = Cache::get('key');

$value = Cache::get('key', 'default');
```

You may even pass a closure as the default value. The result of the closure will be returned if the specified item does not exist in the cache. Passing a closure allows you to defer the retrieval of default values from a database or other external service:

```
$value = Cache::get('key', function () {
    return DB::table(...)->get();
});
```

Checking For Item Existence

The `has` method may be used to determine if an item exists in the cache. This method will also return `false` if the item exists but its value is `null`:

```
if (Cache::has('key')) {
    //
}
```

Incrementing / Decrementing Values

The `increment` and `decrement` methods may be used to adjust the value of integer items in the cache. Both of these methods accept an optional second argument indicating the amount by which to increment or decrement the item's value:

```
Cache::increment('key');
Cache::increment('key', $amount);
Cache::decrement('key');
Cache::decrement('key', $amount);
```

Retrieve & Store

Sometimes you may wish to retrieve an item from the cache, but also store a default value if the requested item doesn't exist. For example, you may wish to retrieve all users from the cache or, if they don't exist, retrieve them from the database and add them to the cache. You may do this using the `Cache::remember` method:

```
$value = Cache::remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

If the item does not exist in the cache, the closure passed to the `remember` method will be executed and its result will be placed in the cache.

You may use the `rememberForever` method to retrieve an item from the cache or store it forever if it does not exist:

```
$value = Cache::rememberForever('users', function () {
    return DB::table('users')->get();
});
```

Retrieve & Delete

If you need to retrieve an item from the cache and then delete the item, you may use the `pull` method. Like the `get` method, `null` will be returned if the item does not exist in the cache:

```
$value = Cache::pull('key');
```

Storing Items In The Cache

You may use the `put` method on the `Cache` facade to store items in the cache:

```
Cache::put('key', 'value', $seconds = 10);
```

If the storage time is not passed to the `put` method, the item will be stored indefinitely:

```
Cache::put('key', 'value');
```

Instead of passing the number of seconds as an integer, you may also pass a `DateTime` instance representing the desired expiration time of the cached item:

```
Cache::put('key', 'value', now()->addMinutes(10));
```

Store If Not Present

The `add` method will only add the item to the cache if it does not already exist in the cache store. The method will return `true` if the item is actually added to the cache. Otherwise, the method will return `false`. The `add` method is an atomic operation:

```
Cache::add('key', 'value', $seconds);
```

Storing Items Forever

The `forever` method may be used to store an item in the cache permanently. Since these items will not expire, they must be manually removed from the cache using the `forget` method:

```
Cache::forever('key', 'value');
```

{tip} If you are using the Memcached driver, items that are stored "forever" may be removed when the cache reaches its size limit.

Removing Items From The Cache

You may remove items from the cache using the `forget` method:

```
Cache::forget('key');
```

You may also remove items by providing a zero or negative number of expiration seconds:

```
Cache::put('key', 'value', 0);

Cache::put('key', 'value', -5);
```

You may clear the entire cache using the `flush` method:

```
Cache::flush();
```

{note} Flushing the cache does not respect your configured cache "prefix" and will remove all entries from the cache. Consider this carefully when clearing a cache which is shared by other applications.

The Cache Helper

In addition to using the `Cache` facade, you may also use the global `cache` function to retrieve and store data via the cache. When the `cache` function is called with a single, string argument, it will return the value of the given key:

```
$value = cache('key');
```

If you provide an array of key / value pairs and an expiration time to the function, it will store values in the cache for the specified duration:

```
cache(['key' => 'value'], $seconds);

cache(['key' => 'value'], now()->addMinutes(10));
```

When the `cache` function is called without any arguments, it returns an instance of the `Illuminate\Contracts\Cache\Factory` implementation, allowing you to call other caching methods:

```
cache()->remember('users', $seconds, function () {
    return DB::table('users')->get();
});
```

{tip} When testing call to the global `cache` function, you may use the `Cache::shouldReceive` method just as if you were testing the facade.

Cache Tags

{note} Cache tags are not supported when using the `file`, `dynamodb`, or `database` cache drivers. Furthermore, when using multiple tags with caches that are stored "forever", performance will be best with a driver such as `memcached`, which automatically purges stale records.

Storing Tagged Cache Items

Cache tags allow you to tag related items in the cache and then flush all cached values that have been assigned a given tag. You may access a tagged cache by passing in an ordered array of tag names. For example, let's access a tagged cache and `put` a value into the cache:

```
Cache::tags(['people', 'artists'])->put('John', $john, $seconds);

Cache::tags(['people', 'authors'])->put('Anne', $anne, $seconds);
```

Accessing Tagged Cache Items

To retrieve a tagged cache item, pass the same ordered list of tags to the `tags` method and then call the `get` method with the key you wish to retrieve:

```
$john = Cache::tags(['people', 'artists'])->get('John');

$anne = Cache::tags(['people', 'authors'])->get('Anne');
```

Removing Tagged Cache Items

You may flush all items that are assigned a tag or list of tags. For example, this statement would remove all caches tagged with either `people`, `authors`, or both. So, both `Anne` and `John` would be removed from the cache:

```
Cache::tags(['people', 'authors'])->flush();
```

In contrast, this statement would remove only cached values tagged with `authors`, so `Anne` would be removed, but not `John`:

```
Cache::tags('authors')->flush();
```

Atomic Locks

{note} To utilize this feature, your application must be using the `memcached`, `redis`, `dynamodb`, `database`, `file`, or `array` cache driver as your application's default cache driver. In addition, all servers must be communicating with the same central cache server.

Driver Prerequisites

Database

When using the `database` cache driver, you will need to setup a table to contain your application's cache locks. You'll find an example `Schema` declaration for the table below:

```
Schema::create('cache_locks', function ($table) {
    $table->string('key')->primary();
    $table->string('owner');
    $table->integer('expiration');
});
```

Managing Locks

Atomic locks allow for the manipulation of distributed locks without worrying about race conditions. For example, Laravel Forge uses atomic locks to ensure that only one remote task is being executed on a server at a time. You may create and manage locks using the `Cache::lock` method:

```
use Illuminate\Support\Facades\Cache;

$lock = Cache::lock('foo', 10);

if ($lock->get()) {
    // Lock acquired for 10 seconds...

    $lock->release();
}
```

The `get` method also accepts a closure. After the closure is executed, Laravel will automatically release the lock:

```
Cache::lock('foo')->get(function () {
    // Lock acquired indefinitely and automatically released...
});
```

If the lock is not available at the moment you request it, you may instruct Laravel to wait for a specified number of seconds. If the lock can not be acquired within the specified time limit, an `Illuminate\Contracts\Cache\LockTimeoutException` will be thrown:

```
use Illuminate\Contracts\Cache\LockTimeoutException;

$lock = Cache::lock('foo', 10);

try {
    $lock->block(5);

    // Lock acquired after waiting a maximum of 5 seconds...
} catch (LockTimeoutException $e) {
    // Unable to acquire lock...
} finally {
    optional($lock)->release();
}
```

The example above may be simplified by passing a closure to the `block` method. When a closure is passed to this method, Laravel will attempt to acquire the lock for the specified number of seconds and will automatically release the lock once the closure has been executed:

```
Cache::lock('foo', 10)->block(5, function () {
    // Lock acquired after waiting a maximum of 5 seconds...
});
```

Managing Locks Across Processes

Sometimes, you may wish to acquire a lock in one process and release it in another process. For example, you may acquire a lock during a web request and wish to release the lock at the end of a queued job that is triggered by that request. In this scenario, you should pass the lock's scoped "owner token" to the queued job so that the job can re-instantiate the lock using the given token.

In the example below, we will dispatch a queued job if a lock is successfully acquired. In addition, we will pass the lock's owner token to the queued job via the lock's `owner` method:

```
$podcast = Podcast::find($id);

$lock = Cache::lock('processing', 120);

if ($lock->get()) {
    ProcessPodcast::dispatch($podcast, $lock->owner());
}
```

Within our application's `ProcessPodcast` job, we can restore and release the lock using the owner token:

```
Cache::restoreLock('processing', $this->owner)->release();
```

If you would like to release a lock without respecting its current owner, you may use the `forceRelease` method:

```
Cache::lock('processing')->forceRelease();
```

Adding Custom Cache Drivers

Writing The Driver

To create our custom cache driver, we first need to implement the `Illuminate\Contracts\Cache\Store` contract. So, a MongoDB cache implementation might look something like this:

```
<?php

namespace App\Extensions;

use Illuminate\Contracts\Cache\Store;

class MongoStore implements Store
{
    public function get($key) {}
    public function many(array $keys) {}
    public function put($key, $value, $seconds) {}
    public function putMany(array $values, $seconds) {}
    public function increment($key, $value = 1) {}
    public function decrement($key, $value = 1) {}
    public function forever($key, $value) {}
    public function forget($key) {}
    public function flush() {}
    public function getPrefix() {}
}
```

We just need to implement each of these methods using a MongoDB connection. For an example of how to implement each of these methods, take a look at the `Illuminate\Cache\MemcachedStore` in the Laravel framework source code. Once our implementation is complete, we can finish our custom driver registration by calling the `Cache` facade's `extend` method:

```
Cache::extend('mongo', function ($app) {
    return Cache::repository(new MongoStore);
});
```

{tip} If you're wondering where to put your custom cache driver code, you could create an `Extensions` namespace within your `app` directory. However, keep in mind that Laravel does not have a rigid application structure and you are free to organize your application according to your preferences.

Registering The Driver

To register the custom cache driver with Laravel, we will use the `extend` method on the `Cache` facade. Since other service providers may attempt to read cached values within their `boot` method, we will register our custom driver within a `booting` callback. By using the `booting` callback, we can ensure that the custom driver is registered just before the `boot` method is called on our application's service providers but after the `register` method is called on all of the service providers. We will register our `booting` callback within the `register` method of our application's `App\Providers\AppServiceProvider` class:

```php
<?php

namespace App\Providers;

use App\Extensions\MongoStore;
use Illuminate\Support\Facades\Cache;
use Illuminate\Support\ServiceProvider;

class CacheServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        $this->app->booting(function () {
             Cache::extend('mongo', function ($app) {
                 return Cache::repository(new MongoStore);
             });
         });
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        //
    }
}
```

The first argument passed to the `extend` method is the name of the driver. This will correspond to your `driver` option in the `config/cache.php` configuration file. The second argument is a closure that should return an `Illuminate\Cache\Repository` instance. The closure will be passed an `$app` instance, which is an instance of the service container.

Once your extension is registered, update your `config/cache.php` configuration file's `driver` option to the name of your extension.

Events

To execute code on every cache operation, you may listen for the events fired by the cache. Typically, you should place these event listeners within your application's `App\Providers\EventServiceProvider` class:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Cache\Events\CacheHit' => [
        'App\Listeners\LogCacheHit',
    ],

    'Illuminate\Cache\Events\CacheMissed' => [
        'App\Listeners\LogCacheMissed',
    ],

    'Illuminate\Cache\Events\KeyForgotten' => [
        'App\Listeners\LogKeyForgotten',
    ],

    'Illuminate\Cache\Events\KeyWritten' => [
        'App\Listeners\LogKeyWritten',
    ],
];
```

Collections

- Introduction
    - Creating Collections
    - Extending Collections
- Available Methods
- Higher Order Messages
- Lazy Collections
    - Introduction
    - Creating Lazy Collections

- The Enumerable Contract
- Lazy Collection Methods

Introduction

The `Illuminate\Support\Collection` class provides a fluent, convenient wrapper for working with arrays of data. For example, check out the following code. We'll use the `collect` helper to create a new collection instance from the array, run the `strtoupper` function on each element, and then remove all empty elements:

```
$collection = collect(['taylor', 'abigail', null])->map(function ($name)
{
    return strtoupper($name);
})->reject(function ($name) {
    return empty($name);
});
```

As you can see, the `Collection` class allows you to chain its methods to perform fluent mapping and reducing of the underlying array. In general, collections are immutable, meaning every `Collection` method returns an entirely new `Collection` instance.

Creating Collections

As mentioned above, the `collect` helper returns a new `Illuminate\Support\Collection` instance for the given array. So, creating a collection is as simple as:

```
$collection = collect([1, 2, 3]);
```

{tip} The results of Eloquent queries are always returned as `Collection` instances.

Extending Collections

Collections are "macroable", which allows you to add additional methods to the `Collection` class at run time. The `Illuminate\Support\Collection` class' `macro` method accepts a closure that will be executed when your macro is called. The macro closure may access the collection's other methods via `$this`, just as if it were a real method of the collection class. For example, the following code adds a `toUpper` method to the

`Collection` class:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Str;

Collection::macro('toUpper', function () {
    return $this->map(function ($value) {
        return Str::upper($value);
    });
});

$collection = collect(['first', 'second']);

$upper = $collection->toUpper();

// ['FIRST', 'SECOND']
```

Typically, you should declare collection macros in the `boot` method of a service provider.

Macro Arguments

If necessary, you may define macros that accept additional arguments:

```
use Illuminate\Support\Collection;
use Illuminate\Support\Facades\Lang;

Collection::macro('toLocale', function ($locale) {
    return $this->map(function ($value) use ($locale) {
        return Lang::get($value, [], $locale);
    });
});

$collection = collect(['first', 'second']);

$translated = $collection->toLocale('es');
```

Available Methods

For the majority of the remaining collection documentation, we'll discuss each method available on the `Collection` class. Remember, all of these methods may be chained to fluently manipulate the underlying array. Furthermore, almost every method returns a new `Collection` instance, allowing you to preserve the original copy of the collection when necessary:

all average avg chunk chunkWhile collapse collect combine concat contains containsStrict count countBy crossJoin dd diff diffAssoc diffKeys dump duplicates duplicatesStrict each eachSpread every except filter first firstWhere flatMap flatten flip forget forPage get groupBy has implode intersect intersectByKeys isEmpty isNotEmpty join keyBy keys last macro make map mapInto mapSpread mapToGroups mapWithKeys max median merge mergeRecursive min mode nth only pad partition pipe pipeInto pluck pop prepend pull push put random range reduce reduceMany reduceSpread reject replace replaceRecursive reverse search shift shuffle sliding skip skipUntil skipWhile slice sole some sort sortBy sortByDesc sortDesc sortKeys sortKeysDesc splice split splitIn sum take takeUntil takeWhile tap times toArray toJson transform union unique uniqueStrict unless unlessEmpty unlessNotEmpty unwrap values when whenEmpty whenNotEmpty where whereStrict whereBetween whereIn whereInStrict whereInstanceOf whereNotBetween whereNotIn whereNotInStrict whereNotNull whereNull wrap zip

Method Listing

`all()` {.collection-method .first-collection-method}

The `all` method returns the underlying array represented by the collection:

```
collect([1, 2, 3])->all();

// [1, 2, 3]
```

`average()` {.collection-method}

Alias for the avg method.

`avg()` {.collection-method}

The avg method returns the average value of a given key:

```
$average = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->avg('foo');

// 20

$average = collect([1, 1, 2, 4])->avg();

// 2
```

`chunk()` {.collection-method}

The chunk method breaks the collection into multiple, smaller collections of a given size:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7]);

$chunks = $collection->chunk(4);

$chunks->all();

// [[1, 2, 3, 4], [5, 6, 7]]
```

This method is especially useful in views when working with a grid system such as Bootstrap. For example, imagine you have a collection of Eloquent models you want to display in a grid:

```
@foreach ($products->chunk(3) as $chunk)
    <div class="row">
        @foreach ($chunk as $product)
            <div class="col-xs-4">{{ $product->name }}</div>
        @endforeach
    </div>
@endforeach
```

`chunkWhile()` {.collection-method}

The `chunkWhile` method breaks the collection into multiple, smaller collections based on the evaluation of the given callback. The `$chunk` variable passed to the closure may be used to inspect the previous element:

```
$collection = collect(str_split('AABBCCCD'));

$chunks = $collection->chunkWhile(function ($value, $key, $chunk) {
    return $value === $chunk->last();
});

$chunks->all();

// [['A', 'A'], ['B', 'B'], ['C', 'C', 'C'], ['D']]
```

`collapse()` {.collection-method}

The `collapse` method collapses a collection of arrays into a single, flat collection:

```
$collection = collect([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
]);

$collapsed = $collection->collapse();

$collapsed->all();

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`collect()` {.collection-method}

The `collect` method returns a new `Collection` instance with the items currently in the collection:

```
$collectionA = collect([1, 2, 3]);

$collectionB = $collectionA->collect();

$collectionB->all();

// [1, 2, 3]
```

The `collect` method is primarily useful for converting lazy collections into standard `Collection` instances:

```
$lazyCollection = LazyCollection::make(function () {
    yield 1;
    yield 2;
    yield 3;
});

$collection = $lazyCollection->collect();

get_class($collection);

// 'Illuminate\Support\Collection'

$collection->all();

// [1, 2, 3]
```

{tip} The `collect` method is especially useful when you have an instance of `Enumerable` and need a non-lazy collection instance. Since `collect()` is part of the `Enumerable` contract, you can safely use it to get a `Collection` instance.

`combine()` {.collection-method}

The `combine` method combines the values of the collection, as keys, with the values of another array or collection:

```
$collection = collect(['name', 'age']);

$combined = $collection->combine(['George', 29]);

$combined->all();

// ['name' => 'George', 'age' => 29]
```

`concat()` {.collection-method}

The `concat` method appends the given `array` or collection's values onto the end of another collection:

```
$collection = collect(['John Doe']);

$concatenated = $collection->concat(['Jane Doe'])->concat(['name' =>
'Johnny Doe']);

$concatenated->all();

// ['John Doe', 'Jane Doe', 'Johnny Doe']
```

`contains()` {.collection-method}

The `contains` method determines whether the collection contains a given item. You may pass a closure to the `contains` method to determine if an element exists in the collection matching a given truth test:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->contains(function ($value, $key) {
    return $value > 5;
});

// false
```

Alternatively, you may pass a string to the `contains` method to determine whether the collection contains a given item value:

```
$collection = collect(['name' => 'Desk', 'price' => 100]);

$collection->contains('Desk');

// true

$collection->contains('New York');

// false
```

You may also pass a key / value pair to the `contains` method, which will determine if the given pair exists in the collection:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->contains('product', 'Bookcase');

// false
```

The `contains` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `containsStrict` method to filter using "strict" comparisons.

`containsStrict()` {.collection-method}

This method has the same signature as the `contains` method; however, all values are compared using "strict" comparisons.

{tip} This method's behavior is modified when using Eloquent Collections.

`count()` {.collection-method}

The `count` method returns the total number of items in the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->count();

// 4
```

`countBy()` {.collection-method}

The `countBy` method counts the occurrences of values in the collection. By default, the method counts the occurrences of every element, allowing you to count certain "types" of elements in the collection:

```
$collection = collect([1, 2, 2, 2, 3]);

$counted = $collection->countBy();

$counted->all();

// [1 => 1, 2 => 3, 3 => 1]
```

You pass a closure to the `countBy` method to count all items by a custom value:

```
$collection = collect(['alice@gmail.com', 'bob@yahoo.com',
'carlos@gmail.com']);

$counted = $collection->countBy(function ($email) {
    return substr(strrchr($email, "@"), 1);
});

$counted->all();

// ['gmail.com' => 2, 'yahoo.com' => 1]
```

`crossJoin()` {.collection-method}

The `crossJoin` method cross joins the collection's values among the given arrays or collections, returning a Cartesian product with all possible permutations:

```
$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b']);

$matrix->all();

/*
    [
        [1, 'a'],
        [1, 'b'],
        [2, 'a'],
        [2, 'b'],
    ]
*/

$collection = collect([1, 2]);

$matrix = $collection->crossJoin(['a', 'b'], ['I', 'II']);

$matrix->all();

/*
    [
        [1, 'a', 'I'],
        [1, 'a', 'II'],
        [1, 'b', 'I'],
        [1, 'b', 'II'],
        [2, 'a', 'I'],
        [2, 'a', 'II'],
        [2, 'b', 'I'],
        [2, 'b', 'II'],
    ]
*/
```

`dd()` {.collection-method}

The `dd` method dumps the collection's items and ends execution of the script:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dd();

/*
    Collection {
        #items: array:2 [
            0 => "John Doe"
            1 => "Jane Doe"
        ]
    }
*/
```

If you do not want to stop executing the script, use the dump method instead.

diff() {.collection-method}

The diff method compares the collection against another collection or a plain PHP array based on its values. This method will return the values in the original collection that are not present in the given collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$diff = $collection->diff([2, 4, 6, 8]);

$diff->all();

// [1, 3, 5]
```

{tip} This method's behavior is modified when using Eloquent Collections.

diffAssoc() {.collection-method}

The diffAssoc method compares the collection against another collection or a plain PHP array based on its keys and values. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'color' => 'orange',
    'type' => 'fruit',
    'remain' => 6,
]);

$diff = $collection->diffAssoc([
    'color' => 'yellow',
    'type' => 'fruit',
    'remain' => 3,
    'used' => 6,
]);

$diff->all();

// ['color' => 'orange', 'remain' => 6]
```

`diffKeys()` {.collection-method}

The `diffKeys` method compares the collection against another collection or a plain PHP `array` based on its keys. This method will return the key / value pairs in the original collection that are not present in the given collection:

```
$collection = collect([
    'one' => 10,
    'two' => 20,
    'three' => 30,
    'four' => 40,
    'five' => 50,
]);

$diff = $collection->diffKeys([
    'two' => 2,
    'four' => 4,
    'six' => 6,
    'eight' => 8,
]);

$diff->all();

// ['one' => 10, 'three' => 30, 'five' => 50]
```

`dump()` {.collection-method}

The `dump` method dumps the collection's items:

```
$collection = collect(['John Doe', 'Jane Doe']);

$collection->dump();

/*
    Collection {
        #items: array:2 [
            0 => "John Doe"
            1 => "Jane Doe"
        ]
    }
*/
```

If you want to stop executing the script after dumping the collection, use the dd method instead.

`duplicates()` {.collection-method}

The `duplicates` method retrieves and returns duplicate values from the collection:

```
$collection = collect(['a', 'b', 'a', 'c', 'b']);

$collection->duplicates();

// [2 => 'a', 4 => 'b']
```

If the collection contains arrays or objects, you can pass the key of the attributes that you wish to check for duplicate values:

```
$employees = collect([
    ['email' => 'abigail@example.com', 'position' => 'Developer'],
    ['email' => 'james@example.com', 'position' => 'Designer'],
    ['email' => 'victoria@example.com', 'position' => 'Developer'],
]);

$employees->duplicates('position');

// [2 => 'Developer']
```

`duplicatesStrict()` {.collection-method}

This method has the same signature as the `duplicates` method; however, all values are compared using "strict" comparisons.

`each()` {.collection-method}

The `each` method iterates over the items in the collection and passes each item to a closure:

```
$collection->each(function ($item, $key) {
    //
});
```

If you would like to stop iterating through the items, you may return `false` from your closure:

```
$collection->each(function ($item, $key) {
    if (/* condition */) {
        return false;
    }
});
```

`eachSpread()` {.collection-method}

The `eachSpread` method iterates over the collection's items, passing each nested item value into the given callback:

```
$collection = collect([['John Doe', 35], ['Jane Doe', 33]]);

$collection->eachSpread(function ($name, $age) {
    //
});
```

You may stop iterating through the items by returning `false` from the callback:

```
$collection->eachSpread(function ($name, $age) {
    return false;
});
```

`every()` {.collection-method}

The `every` method may be used to verify that all elements of a collection pass a given truth test:

```
collect([1, 2, 3, 4])->every(function ($value, $key) {
    return $value > 2;
});

// false
```

If the collection is empty, the `every` method will return true:

```
$collection = collect([]);

$collection->every(function ($value, $key) {
    return $value > 2;
});

// true
```

`except()` {.collection-method}

The `except` method returns all items in the collection except for those with the specified keys:

```
$collection = collect(['product_id' => 1, 'price' => 100, 'discount' =>
false]);

$filtered = $collection->except(['price', 'discount']);

$filtered->all();

// ['product_id' => 1]
```

For the inverse of `except`, see the only method.

{tip} This method's behavior is modified when using Eloquent Collections.

`filter()` {.collection-method}

The `filter` method filters the collection using the given callback, keeping only those items that pass a given truth test:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->filter(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [3, 4]
```

If no callback is supplied, all entries of the collection that are equivalent to `false` will be removed:

```
$collection = collect([1, 2, 3, null, false, '', 0, []]);

$collection->filter()->all();

// [1, 2, 3]
```

For the inverse of `filter`, see the reject method.

`first()` {.collection-method}

The `first` method returns the first element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->first(function ($value, $key) {
    return $value > 2;
});

// 3
```

You may also call the `first` method with no arguments to get the first element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->first();

// 1
```

`firstWhere()` {.collection-method}

The `firstWhere` method returns the first element in the collection with the given key / value pair:

```
$collection = collect([
    ['name' => 'Regena', 'age' => null],
    ['name' => 'Linda', 'age' => 14],
    ['name' => 'Diego', 'age' => 23],
    ['name' => 'Linda', 'age' => 84],
]);

$collection->firstWhere('name', 'Linda');

// ['name' => 'Linda', 'age' => 14]
```

You may also call the `firstWhere` method with a comparison operator:

```
$collection->firstWhere('age', '>=', 18);

// ['name' => 'Diego', 'age' => 23]
```

Like the where method, you may pass one argument to the `firstWhere` method. In this scenario, the `firstWhere` method will return the first item where the given item key's value is "truthy":

```
$collection->firstWhere('age');

// ['name' => 'Linda', 'age' => 14]
```

`flatMap()` {.collection-method}

The `flatMap` method iterates through the collection and passes each value to the given

closure. The closure is free to modify the item and return it, thus forming a new collection of modified items. Then, the array is flattened by one level:

```
$collection = collect([
    ['name' => 'Sally'],
    ['school' => 'Arkansas'],
    ['age' => 28]
]);

$flattened = $collection->flatMap(function ($values) {
    return array_map('strtoupper', $values);
});

$flattened->all();

// ['name' => 'SALLY', 'school' => 'ARKANSAS', 'age' => '28'];
```

`flatten()` {.collection-method}

The `flatten` method flattens a multi-dimensional collection into a single dimension:

```
$collection = collect([
    'name' => 'taylor',
    'languages' => [
        'php', 'javascript'
    ]
]);

$flattened = $collection->flatten();

$flattened->all();

// ['taylor', 'php', 'javascript'];
```

If necessary, you may pass the `flatten` method a "depth" argument:

```
$collection = collect([
    'Apple' => [
        [
            'name' => 'iPhone 6S',
            'brand' => 'Apple'
        ],
    ],
    'Samsung' => [
        [
            'name' => 'Galaxy S7',
            'brand' => 'Samsung'
        ],
    ],
]);

$products = $collection->flatten(1);

$products->values()->all();

/*
    [
        ['name' => 'iPhone 6S', 'brand' => 'Apple'],
        ['name' => 'Galaxy S7', 'brand' => 'Samsung'],
    ]
*/
```

In this example, calling `flatten` without providing the depth would have also flattened the nested arrays, resulting in `['iPhone 6S', 'Apple', 'Galaxy S7', 'Samsung']`. Providing a depth allows you to specify the number of levels nested arrays will be flattened.

`flip()` {.collection-method}

The `flip` method swaps the collection's keys with their corresponding values:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$flipped = $collection->flip();

$flipped->all();

// ['taylor' => 'name', 'laravel' => 'framework']
```

`forget()` {.collection-method}

The `forget` method removes an item from the collection by its key:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$collection->forget('name');

$collection->all();

// ['framework' => 'laravel']
```

{note} Unlike most other collection methods, `forget` does not return a new modified collection; it modifies the collection it is called on.

`forPage()` {.collection-method}

The `forPage` method returns a new collection containing the items that would be present on a given page number. The method accepts the page number as its first argument and the number of items to show per page as its second argument:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunk = $collection->forPage(2, 3);

$chunk->all();

// [4, 5, 6]
```

`get()` {.collection-method}

The `get` method returns the item at a given key. If the key does not exist, `null` is returned:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('name');

// taylor
```

You may optionally pass a default value as the second argument:

```
$collection = collect(['name' => 'taylor', 'framework' => 'laravel']);

$value = $collection->get('age', 34);

// 34
```

You may even pass a callback as the method's default value. The result of the callback will be returned if the specified key does not exist:

```
$collection->get('email', function () {
    return 'taylor@example.com';
});

// taylor@example.com
```

`groupBy()` {.collection-method}

The `groupBy` method groups the collection's items by a given key:

```
$collection = collect([
    ['account_id' => 'account-x10', 'product' => 'Chair'],
    ['account_id' => 'account-x10', 'product' => 'Bookcase'],
    ['account_id' => 'account-x11', 'product' => 'Desk'],
]);

$grouped = $collection->groupBy('account_id');

$grouped->all();

/*
    [
        'account-x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'account-x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

Instead of passing a string key, you may pass a callback. The callback should return the value you wish to key the group by:

```
$grouped = $collection->groupBy(function ($item, $key) {
    return substr($item['account_id'], -3);
});

$grouped->all();

/*
    [
        'x10' => [
            ['account_id' => 'account-x10', 'product' => 'Chair'],
            ['account_id' => 'account-x10', 'product' => 'Bookcase'],
        ],
        'x11' => [
            ['account_id' => 'account-x11', 'product' => 'Desk'],
        ],
    ]
*/
```

Multiple grouping criteria may be passed as an array. Each array element will be applied to the corresponding level within a multi-dimensional array:

```
$data = new Collection([
    10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1', 'Role_3']],
    20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1', 'Role_2']],
    30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
    40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
]);

$result = $data->groupBy(['skill', function ($item) {
    return $item['roles'];
}], $preserveKeys = true);

/*
[
    1 => [
        'Role_1' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1',
'Role_3']],
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1',
'Role_2']],
        ],
        'Role_2' => [
            20 => ['user' => 2, 'skill' => 1, 'roles' => ['Role_1',
'Role_2']],
        ],
        'Role_3' => [
            10 => ['user' => 1, 'skill' => 1, 'roles' => ['Role_1',
'Role_3']],
        ],
    ],
    2 => [
        'Role_1' => [
            30 => ['user' => 3, 'skill' => 2, 'roles' => ['Role_1']],
        ],
        'Role_2' => [
            40 => ['user' => 4, 'skill' => 2, 'roles' => ['Role_2']],
        ],
    ],
];
*/
```

`has()` {.collection-method}

The `has` method determines if a given key exists in the collection:

```
$collection = collect(['account_id' => 1, 'product' => 'Desk', 'amount'
=> 5]);

$collection->has('product');

// true

$collection->has(['product', 'amount']);

// true

$collection->has(['amount', 'price']);

// false
```

`implode()` {.collection-method}

The `implode` method joins items in a collection. Its arguments depend on the type of items in the collection. If the collection contains arrays or objects, you should pass the key of the attributes you wish to join, and the "glue" string you wish to place between the values:

```
$collection = collect([
    ['account_id' => 1, 'product' => 'Desk'],
    ['account_id' => 2, 'product' => 'Chair'],
]);

$collection->implode('product', ', ');

// Desk, Chair
```

If the collection contains simple strings or numeric values, you should pass the "glue" as the only argument to the method:

```
collect([1, 2, 3, 4, 5])->implode('-');

// '1-2-3-4-5'
```

`intersect()` {.collection-method}

The `intersect` method removes any values from the original collection that are not present in the given `array` or collection. The resulting collection will preserve the original collection's keys:

```
$collection = collect(['Desk', 'Sofa', 'Chair']);

$intersect = $collection->intersect(['Desk', 'Chair', 'Bookcase']);

$intersect->all();

// [0 => 'Desk', 2 => 'Chair']
```

{tip} This method's behavior is modified when using Eloquent Collections.

`intersectByKeys()` {.collection-method}

The `intersectByKeys` method removes any keys and their corresponding values from the original collection that are not present in the given `array` or collection:

```
$collection = collect([
    'serial' => 'UX301', 'type' => 'screen', 'year' => 2009,
]);

$intersect = $collection->intersectByKeys([
    'reference' => 'UX404', 'type' => 'tab', 'year' => 2011,
]);

$intersect->all();

// ['type' => 'screen', 'year' => 2009]
```

`isEmpty()` {.collection-method}

The `isEmpty` method returns `true` if the collection is empty; otherwise, `false` is returned:

```
collect([])->isEmpty();

// true
```

`isNotEmpty()` {.collection-method}

The `isNotEmpty` method returns `true` if the collection is not empty; otherwise, `false` is returned:

```
collect([])->isNotEmpty();

// false
```

`join()` {.collection-method}

The `join` method joins the collection's values with a string. Using this method's second argument, you may also specify how the final element should be appended to the string:

```
collect(['a', 'b', 'c'])->join(', '); // 'a, b, c'
collect(['a', 'b', 'c'])->join(', ', ', and '); // 'a, b, and c'
collect(['a', 'b'])->join(', ', ' and '); // 'a and b'
collect(['a'])->join(', ', ' and '); // 'a'
collect([])->join(', ', ' and '); // ''
```

`keyBy()` {.collection-method}

The `keyBy` method keys the collection by the given key. If multiple items have the same key, only the last one will appear in the new collection:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keyed = $collection->keyBy('product_id');

$keyed->all();

/*
    [
        'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

You may also pass a callback to the method. The callback should return the value to key the collection by:

```
$keyed = $collection->keyBy(function ($item) {
    return strtoupper($item['product_id']);
});

$keyed->all();

/*
    [
        'PROD-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
        'PROD-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
    ]
*/
```

`keys()` {.collection-method}

The `keys` method returns all of the collection's keys:

```
$collection = collect([
    'prod-100' => ['product_id' => 'prod-100', 'name' => 'Desk'],
    'prod-200' => ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$keys = $collection->keys();

$keys->all();

// ['prod-100', 'prod-200']
```

`last()` {.collection-method}

The `last` method returns the last element in the collection that passes a given truth test:

```
collect([1, 2, 3, 4])->last(function ($value, $key) {
    return $value < 3;
});

// 2
```

You may also call the `last` method with no arguments to get the last element in the collection. If the collection is empty, `null` is returned:

```
collect([1, 2, 3, 4])->last();

// 4
```

`macro()` {.collection-method}

The static `macro` method allows you to add methods to the `Collection` class at run time. Refer to the documentation on extending collections for more information.

`make()` {.collection-method}

The static `make` method creates a new collection instance. See the Creating Collections section.

`map()` {.collection-method}

The `map` method iterates through the collection and passes each value to the given callback. The callback is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([1, 2, 3, 4, 5]);

$multiplied = $collection->map(function ($item, $key) {
    return $item * 2;
});

$multiplied->all();

// [2, 4, 6, 8, 10]
```

{note} Like most other collection methods, `map` returns a new collection instance; it does not modify the collection it is called on. If you want to transform the original collection, use the `transform` method.

`mapInto()` {.collection-method}

The `mapInto()` method iterates over the collection, creating a new instance of the given class by passing the value into the constructor:

```
class Currency
{
    /**
     * Create a new currency instance.
     *
     * @param  string  $code
     * @return void
     */
    function __construct(string $code)
    {
        $this->code = $code;
    }
}

$collection = collect(['USD', 'EUR', 'GBP']);

$currencies = $collection->mapInto(Currency::class);

$currencies->all();

// [Currency('USD'), Currency('EUR'), Currency('GBP')]
```

`mapSpread()` {.collection-method}

The `mapSpread` method iterates over the collection's items, passing each nested item value into the given closure. The closure is free to modify the item and return it, thus forming a new collection of modified items:

```
$collection = collect([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]);

$chunks = $collection->chunk(2);

$sequence = $chunks->mapSpread(function ($even, $odd) {
    return $even + $odd;
});

$sequence->all();

// [1, 5, 9, 13, 17]
```

`mapToGroups()` {.collection-method}

The `mapToGroups` method groups the collection's items by the given closure. The closure should return an associative array containing a single key / value pair, thus forming a new collection of grouped values:

```
$collection = collect([
    [
        'name' => 'John Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Jane Doe',
        'department' => 'Sales',
    ],
    [
        'name' => 'Johnny Doe',
        'department' => 'Marketing',
    ]
]);

$grouped = $collection->mapToGroups(function ($item, $key) {
    return [$item['department'] => $item['name']];
});

$grouped->all();

/*
    [
        'Sales' => ['John Doe', 'Jane Doe'],
        'Marketing' => ['Johnny Doe'],
    ]
*/

$grouped->get('Sales')->all();

// ['John Doe', 'Jane Doe']
```

`mapWithKeys()` {.collection-method}

The `mapWithKeys` method iterates through the collection and passes each value to the given callback. The callback should return an associative array containing a single key / value pair:

```
$collection = collect([
    [
        'name' => 'John',
        'department' => 'Sales',
        'email' => 'john@example.com',
    ],
    [
        'name' => 'Jane',
        'department' => 'Marketing',
        'email' => 'jane@example.com',
    ]
]);

$keyed = $collection->mapWithKeys(function ($item, $key) {
    return [$item['email'] => $item['name']];
});

$keyed->all();

/*
    [
        'john@example.com' => 'John',
        'jane@example.com' => 'Jane',
    ]
*/
```

`max()` {.collection-method}

The `max` method returns the maximum value of a given key:

```
$max = collect([
    ['foo' => 10],
    ['foo' => 20]
])->max('foo');

// 20

$max = collect([1, 2, 3, 4, 5])->max();

// 5
```

`median()` {.collection-method}

The `median` method returns the median value of a given key:

```
$median = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->median('foo');

// 15

$median = collect([1, 1, 2, 4])->median();

// 1.5
```

`merge()` {.collection-method}

The `merge` method merges the given array or collection with the original collection. If a string key in the given items matches a string key in the original collection, the given items's value will overwrite the value in the original collection:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->merge(['price' => 200, 'discount' => false]);

$merged->all();

// ['product_id' => 1, 'price' => 200, 'discount' => false]
```

If the given items's keys are numeric, the values will be appended to the end of the collection:

```
$collection = collect(['Desk', 'Chair']);

$merged = $collection->merge(['Bookcase', 'Door']);

$merged->all();

// ['Desk', 'Chair', 'Bookcase', 'Door']
```

`mergeRecursive()` {.collection-method}

The `mergeRecursive` method merges the given array or collection recursively with the original collection. If a string key in the given items matches a string key in the original collection, then the values for these keys are merged together into an array, and this is done recursively:

```
$collection = collect(['product_id' => 1, 'price' => 100]);

$merged = $collection->mergeRecursive([
    'product_id' => 2,
    'price' => 200,
    'discount' => false
]);

$merged->all();

// ['product_id' => [1, 2], 'price' => [100, 200], 'discount' => false]
```

`min()` {.collection-method}

The `min` method returns the minimum value of a given key:

```
$min = collect([['foo' => 10], ['foo' => 20]])->min('foo');

// 10

$min = collect([1, 2, 3, 4, 5])->min();

// 1
```

`mode()` {.collection-method}

The `mode` method returns the mode value of a given key:

```
$mode = collect([
    ['foo' => 10],
    ['foo' => 10],
    ['foo' => 20],
    ['foo' => 40]
])->mode('foo');

// [10]

$mode = collect([1, 1, 2, 4])->mode();

// [1]

$mode = collect([1, 1, 2, 2])->mode();

// [1, 2]
```

`nth()` {.collection-method}

The `nth` method creates a new collection consisting of every n-th element:

```
$collection = collect(['a', 'b', 'c', 'd', 'e', 'f']);

$collection->nth(4);

// ['a', 'e']
```

You may optionally pass a starting offset as the second argument:

```
$collection->nth(4, 1);

// ['b', 'f']
```

`only()` {.collection-method}

The `only` method returns the items in the collection with the specified keys:

```
$collection = collect([
    'product_id' => 1,
    'name' => 'Desk',
    'price' => 100,
    'discount' => false
]);

$filtered = $collection->only(['product_id', 'name']);

$filtered->all();

// ['product_id' => 1, 'name' => 'Desk']
```

For the inverse of `only`, see the except method.

{tip} This method's behavior is modified when using Eloquent Collections.

`pad()` {.collection-method}

The `pad` method will fill the array with the given value until the array reaches the specified size. This method behaves like the array_pad PHP function.

To pad to the left, you should specify a negative size. No padding will take place if the absolute value of the given size is less than or equal to the length of the array:

```
$collection = collect(['A', 'B', 'C']);

$filtered = $collection->pad(5, 0);

$filtered->all();

// ['A', 'B', 'C', 0, 0]

$filtered = $collection->pad(-5, 0);

$filtered->all();

// [0, 0, 'A', 'B', 'C']
```

`partition()` {.collection-method}

The `partition` method may be combined with PHP array destructuring to separate elements that pass a given truth test from those that do not:

```
$collection = collect([1, 2, 3, 4, 5, 6]);

[$underThree, $equalOrAboveThree] = $collection->partition(function ($i)
{
    return $i < 3;
});

$underThree->all();

// [1, 2]

$equalOrAboveThree->all();

// [3, 4, 5, 6]
```

`pipe()` {.collection-method}

The `pipe` method passes the collection to the given closure and returns the result of the executed closure:

```
$collection = collect([1, 2, 3]);

$piped = $collection->pipe(function ($collection) {
    return $collection->sum();
});

// 6
```

`pipeInto()` {.collection-method}

The `pipeInto` method creates a new instance of the given class and passes the collection into the constructor:

```
class ResourceCollection
{
    /**
     * The Collection instance.
     */
    public $collection;

    /**
     * Create a new ResourceCollection instance.
     *
     * @param  Collection  $collection
     * @return void
     */
    public function __construct(Collection $collection)
    {
        $this->collection = $collection;
    }
}

$collection = collect([1, 2, 3]);

$resource = $collection->pipeInto(ResourceCollection::class);

$resource->collection->all();

// [1, 2, 3]
```

`pluck()` {.collection-method}

The `pluck` method retrieves all of the values for a given key:

```
$collection = collect([
    ['product_id' => 'prod-100', 'name' => 'Desk'],
    ['product_id' => 'prod-200', 'name' => 'Chair'],
]);

$plucked = $collection->pluck('name');

$plucked->all();

// ['Desk', 'Chair']
```

You may also specify how you wish the resulting collection to be keyed:

```
$plucked = $collection->pluck('name', 'product_id');

$plucked->all();

// ['prod-100' => 'Desk', 'prod-200' => 'Chair']
```

The `pluck` method also supports retrieving nested values using "dot" notation:

```
$collection = collect([
    [
        'speakers' => [
            'first_day' => ['Rosa', 'Judith'],
            'second_day' => ['Angela', 'Kathleen'],
        ],
    ],
]);

$plucked = $collection->pluck('speakers.first_day');

$plucked->all();

// ['Rosa', 'Judith']
```

If duplicate keys exist, the last matching element will be inserted into the plucked collection:

```
$collection = collect([
    ['brand' => 'Tesla',  'color' => 'red'],
    ['brand' => 'Pagani', 'color' => 'white'],
    ['brand' => 'Tesla',  'color' => 'black'],
    ['brand' => 'Pagani', 'color' => 'orange'],
]);

$plucked = $collection->pluck('color', 'brand');

$plucked->all();

// ['Tesla' => 'black', 'Pagani' => 'orange']
```

`pop()` {.collection-method}

The pop method removes and returns the last item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop();

// 5

$collection->all();

// [1, 2, 3, 4]
```

You may pass an integer to the pop method to remove and return multiple items from the end of a collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->pop(3);

// collect([5, 4, 3])

$collection->all();

// [1, 2]
```

`prepend()` {.collection-method}

The `prepend` method adds an item to the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->prepend(0);

$collection->all();

// [0, 1, 2, 3, 4, 5]
```

You may also pass a second argument to specify the key of the prepended item:

```
$collection = collect(['one' => 1, 'two' => 2]);

$collection->prepend(0, 'zero');

$collection->all();

// ['zero' => 0, 'one' => 1, 'two' => 2]
```

`pull()` {.collection-method}

The `pull` method removes and returns an item from the collection by its key:

```
$collection = collect(['product_id' => 'prod-100', 'name' => 'Desk']);

$collection->pull('name');

// 'Desk'

$collection->all();

// ['product_id' => 'prod-100']
```

`push()` {.collection-method}

The `push` method appends an item to the end of the collection:

```
$collection = collect([1, 2, 3, 4]);

$collection->push(5);

$collection->all();

// [1, 2, 3, 4, 5]
```

`put()` {.collection-method}

The `put` method sets the given key and value in the collection:

```
$collection = collect(['product_id' => 1, 'name' => 'Desk']);

$collection->put('price', 100);

$collection->all();

// ['product_id' => 1, 'name' => 'Desk', 'price' => 100]
```

`random()` {.collection-method}

The `random` method returns a random item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->random();

// 4 - (retrieved randomly)
```

You may pass an integer to `random` to specify how many items you would like to randomly retrieve. A collection of items is always returned when explicitly passing the number of items you wish to receive:

```
$random = $collection->random(3);

$random->all();

// [2, 4, 5] - (retrieved randomly)
```

If the collection instance has fewer items than requested, the `random` method will throw an `InvalidArgumentException`.

`range()` {.collection-method}

The `range` method returns a collection containing integers between the specified range:

```
$collection = collect()->range(3, 6);

$collection->all();

// [3, 4, 5, 6]
```

`reduce()` {.collection-method}

The `reduce` method reduces the collection to a single value, passing the result of each iteration into the subsequent iteration:

```
$collection = collect([1, 2, 3]);

$total = $collection->reduce(function ($carry, $item) {
    return $carry + $item;
});

// 6
```

The value for `$carry` on the first iteration is `null`; however, you may specify its initial value by passing a second argument to `reduce`:

```
$collection->reduce(function ($carry, $item) {
    return $carry + $item;
}, 4);

// 10
```

The `reduce` method also passes array keys in associative collections to the given callback:

```
$collection = collect([
    'usd' => 1400,
    'gbp' => 1200,
    'eur' => 1000,
]);

$ratio = [
    'usd' => 1,
    'gbp' => 1.37,
    'eur' => 1.22,
];

$collection->reduce(function ($carry, $value, $key) use ($ratio) {
    return $carry + ($value * $ratio[$key]);
});

// 4264
```

`reduceMany()` {#collection-method}

The `reduceMany` method reduces the collection to an array of values, passing the results of each iteration into the subsequent iteration. This method is similar to the `reduce` method; however, it can accept multiple initial values:

```
[$creditsRemaining, $batch] = Image::where('status', 'unprocessed')
    ->get()
    ->reduceMany(function ($creditsRemaining, $batch, $image) {
        if ($creditsRemaining >= $image->creditsRequired()) {
            $batch->push($image);

            $creditsRemaining -= $image->creditsRequired();
        }

        return [$creditsRemaining, $batch];
    }, $creditsAvailable, collect());
```

`reduceSpread()` {.collection-method}

The `reduceSpread` method reduces the collection to an array of values, passing the results of each iteration into the subsequent iteration. This method is similar to the `reduce` method; however, it can accept multiple initial values:

```
[$creditsRemaining, $batch] = Image::where('status', 'unprocessed')
    ->get()
    ->reduceSpread(function ($creditsRemaining, $batch, $image) {
        if ($creditsRemaining >= $image->creditsRequired()) {
            $batch->push($image);

            $creditsRemaining -= $image->creditsRequired();
        }

        return [$creditsRemaining, $batch];
    }, $creditsAvailable, collect());
```

`reject()` {.collection-method}

The `reject` method filters the collection using the given closure. The closure should return `true` if the item should be removed from the resulting collection:

```
$collection = collect([1, 2, 3, 4]);

$filtered = $collection->reject(function ($value, $key) {
    return $value > 2;
});

$filtered->all();

// [1, 2]
```

For the inverse of the `reject` method, see the `filter` method.

`replace()` {.collection-method}

The `replace` method behaves similarly to `merge`; however, in addition to overwriting matching items that have string keys, the `replace` method will also overwrite items in the collection that have matching numeric keys:

```
$collection = collect(['Taylor', 'Abigail', 'James']);

$replaced = $collection->replace([1 => 'Victoria', 3 => 'Finn']);

$replaced->all();

// ['Taylor', 'Victoria', 'James', 'Finn']
```

`replaceRecursive()` {.collection-method}

This method works like `replace`, but it will recur into arrays and apply the same replacement process to the inner values:

```
$collection = collect([
    'Taylor',
    'Abigail',
    [
        'James',
        'Victoria',
        'Finn'
    ]
]);

$replaced = $collection->replaceRecursive([
    'Charlie',
    2 => [1 => 'King']
]);

$replaced->all();

// ['Charlie', 'Abigail', ['James', 'King', 'Finn']]
```

`reverse()` {.collection-method}

The `reverse` method reverses the order of the collection's items, preserving the original keys:

```
$collection = collect(['a', 'b', 'c', 'd', 'e']);

$reversed = $collection->reverse();

$reversed->all();

/*
    [
        4 => 'e',
        3 => 'd',
        2 => 'c',
        1 => 'b',
        0 => 'a',
    ]
*/
```

`search()` {.collection-method}

The `search` method searches the collection for the given value and returns its key if found. If the item is not found, `false` is returned:

```
$collection = collect([2, 4, 6, 8]);

$collection->search(4);

// 1
```

The search is done using a "loose" comparison, meaning a string with an integer value will be considered equal to an integer of the same value. To use "strict" comparison, pass `true` as the second argument to the method:

```
collect([2, 4, 6, 8])->search('4', $strict = true);

// false
```

Alternatively, you may provide your own closure to search for the first item that passes a given truth test:

```
collect([2, 4, 6, 8])->search(function ($item, $key) {
    return $item > 5;
});

// 2
```

`shift()` {.collection-method}

The `shift` method removes and returns the first item from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift();

// 1

$collection->all();

// [2, 3, 4, 5]
```

You may pass an integer to the `shift` method to remove and return multiple items from the beginning of a collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->shift(3);

// collect([1, 2, 3])

$collection->all();

// [4, 5]
```

`shuffle()` {.collection-method}

The `shuffle` method randomly shuffles the items in the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$shuffled = $collection->shuffle();

$shuffled->all();

// [3, 2, 5, 1, 4] - (generated randomly)
```

`sliding()` {.collection-method}

The `sliding` method returns a new collection of chunks representing a "sliding window" view of the items in the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(2);

$chunks->toArray();

// [[1, 2], [2, 3], [3, 4], [4, 5]]
```

This is especially useful in conjunction with the `eachSpread` method:

```
$transactions->sliding(2)->eachSpread(function ($previous, $current) {
    $current->total = $previous->total + $current->amount;
});
```

You may optionally pass a second "step" value, which determines the distance between the first item of every chunk:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunks = $collection->sliding(3, step: 2);

$chunks->toArray();

// [[1, 2, 3], [3, 4, 5]]
```

`skip()` {.collection-method}

The `skip` method returns a new collection, with the given number of elements removed from the beginning of the collection:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$collection = $collection->skip(4);

$collection->all();

// [5, 6, 7, 8, 9, 10]
```

`skipUntil()` {.collection-method}

The `skipUntil` method skips over items from the collection until the given callback returns `true` and then returns the remaining items in the collection as a new collection instance:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(function ($item) {
    return $item >= 3;
});

$subset->all();

// [3, 4]
```

You may also pass a simple value to the `skipUntil` method to skip all items until the given value is found:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipUntil(3);

$subset->all();

// [3, 4]
```

{note} If the given value is not found or the callback never returns `true`, the `skipUntil` method will return an empty collection.

`skipWhile()` {.collection-method}

The `skipWhile` method skips over items from the collection while the given callback returns `true` and then returns the remaining items in the collection as a new collection:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->skipWhile(function ($item) {
    return $item <= 3;
});

$subset->all();

// [4]
```

{note} If the callback never returns `false`, the `skipWhile` method will return an empty collection.

`slice()` {.collection-method}

The `slice` method returns a slice of the collection starting at the given index:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$slice = $collection->slice(4);

$slice->all();

// [5, 6, 7, 8, 9, 10]
```

If you would like to limit the size of the returned slice, pass the desired size as the second argument to the method:

```
$slice = $collection->slice(4, 2);

$slice->all();

// [5, 6]
```

The returned slice will preserve keys by default. If you do not wish to preserve the original keys, you can use the `values` method to reindex them.

`sole()` {.collection-method}

The `sole` method returns the first element in the collection that passes a given truth test, but only if the truth test matches exactly one element:

```
collect([1, 2, 3, 4])->sole(function ($value, $key) {
    return $value === 2;
});

// 2
```

You may also pass a key / value pair to the `sole` method, which will return the first element in the collection that matches the given pair, but only if it exactly one element matches:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
]);

$collection->sole('product', 'Chair');

// ['product' => 'Chair', 'price' => 100]
```

Alternatively, you may also call the `sole` method with no argument to get the first element in the collection if there is only one element:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
]);

$collection->sole();

// ['product' => 'Desk', 'price' => 200]
```

If there are no elements in the collection that should be returned by the `sole` method, an `\Illuminate\Collections\ItemNotFoundException` exception will be thrown. If

there is more than one element that should be returned, an `\Illuminate\Collections\MultipleItemsFoundException` will be thrown.

`some()` {.collection-method}

Alias for the `contains` method.

`sort()` {.collection-method}

The `sort` method sorts the collection. The sorted collection keeps the original array keys, so in the following example we will use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sort();

$sorted->values()->all();

// [1, 2, 3, 4, 5]
```

If your sorting needs are more advanced, you may pass a callback to `sort` with your own algorithm. Refer to the PHP documentation on `uasort`, which is what the collection's `sort` method calls utilizes internally.

{tip} If you need to sort a collection of nested arrays or objects, see the `sortBy` and `sortByDesc` methods.

`sortBy()` {.collection-method}

The `sortBy` method sorts the collection by the given key. The sorted collection keeps the original array keys, so in the following example we will use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([
    ['name' => 'Desk', 'price' => 200],
    ['name' => 'Chair', 'price' => 100],
    ['name' => 'Bookcase', 'price' => 150],
]);

$sorted = $collection->sortBy('price');

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'price' => 100],
        ['name' => 'Bookcase', 'price' => 150],
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

The `sort` method accepts sort flags as its second argument:

```
$collection = collect([
    ['title' => 'Item 1'],
    ['title' => 'Item 12'],
    ['title' => 'Item 3'],
]);

$sorted = $collection->sortBy('title', SORT_NATURAL);

$sorted->values()->all();

/*
    [
        ['title' => 'Item 1'],
        ['title' => 'Item 3'],
        ['title' => 'Item 12'],
    ]
*/
```

Alternatively, you may pass your own closure to determine how to sort the collection's values:

```
$collection = collect([
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$sorted = $collection->sortBy(function ($product, $key) {
    return count($product['colors']);
});

$sorted->values()->all();

/*
    [
        ['name' => 'Chair', 'colors' => ['Black']],
        ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
        ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
    ]
*/
```

If you would like to sort your collection by multiple attributes, you may pass an array of sort operations to the `sortBy` method. Each sort operation should be an array consisting of the attribute that you wish to sort by and the direction of the desired sort:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    ['name', 'asc'],
    ['age', 'desc'],
]);

$sorted->values()->all();

/*
    [
        ['name' => 'Abigail Otwell', 'age' => 32],
        ['name' => 'Abigail Otwell', 'age' => 30],
        ['name' => 'Taylor Otwell', 'age' => 36],
        ['name' => 'Taylor Otwell', 'age' => 34],
    ]
*/
```

When sorting a collection by multiple attributes, you may also provide closures that define each sort operation:

```
$collection = collect([
    ['name' => 'Taylor Otwell', 'age' => 34],
    ['name' => 'Abigail Otwell', 'age' => 30],
    ['name' => 'Taylor Otwell', 'age' => 36],
    ['name' => 'Abigail Otwell', 'age' => 32],
]);

$sorted = $collection->sortBy([
    fn ($a, $b) => $a['name'] <=> $b['name'],
    fn ($a, $b) => $b['age'] <=> $a['age'],
]);

$sorted->values()->all();

/*
    [
        ['name' => 'Abigail Otwell', 'age' => 32],
        ['name' => 'Abigail Otwell', 'age' => 30],
        ['name' => 'Taylor Otwell', 'age' => 36],
        ['name' => 'Taylor Otwell', 'age' => 34],
    ]
*/
```

`sortByDesc()` {.collection-method}

This method has the same signature as the `sortBy` method, but will sort the collection in the opposite order.

`sortDesc()` {.collection-method}

This method will sort the collection in the opposite order as the `sort` method:

```
$collection = collect([5, 3, 1, 2, 4]);

$sorted = $collection->sortDesc();

$sorted->values()->all();

// [5, 4, 3, 2, 1]
```

Unlike `sort`, you may not pass a closure to `sortDesc`. Instead, you should use the `sort`

method and invert your comparison.

`sortKeys()` {.collection-method}

The `sortKeys` method sorts the collection by the keys of the underlying associative array:

```
$collection = collect([
    'id' => 22345,
    'first' => 'John',
    'last' => 'Doe',
]);

$sorted = $collection->sortKeys();

$sorted->all();

/*
    [
        'first' => 'John',
        'id' => 22345,
        'last' => 'Doe',
    ]
*/
```

`sortKeysDesc()` {.collection-method}

This method has the same signature as the `sortKeys` method, but will sort the collection in the opposite order.

`splice()` {.collection-method}

The `splice` method removes and returns a slice of items starting at the specified index:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2);

$chunk->all();

// [3, 4, 5]

$collection->all();

// [1, 2]
```

You may pass a second argument to limit the size of the resulting collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 4, 5]
```

In addition, you may pass a third argument containing the new items to replace the items removed from the collection:

```
$collection = collect([1, 2, 3, 4, 5]);

$chunk = $collection->splice(2, 1, [10, 11]);

$chunk->all();

// [3]

$collection->all();

// [1, 2, 10, 11, 4, 5]
```

`split()` {.collection-method}

The `split` method breaks a collection into the given number of groups:

```
$collection = collect([1, 2, 3, 4, 5]);

$groups = $collection->split(3);

$groups->all();

// [[1, 2], [3, 4], [5]]
```

`splitIn()` {.collection-method}

The `splitIn` method breaks a collection into the given number of groups, filling non-terminal groups completely before allocating the remainder to the final group:

```
$collection = collect([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);

$groups = $collection->splitIn(3);

$groups->all();

// [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10]]
```

`sum()` {.collection-method}

The `sum` method returns the sum of all items in the collection:

```
collect([1, 2, 3, 4, 5])->sum();

// 15
```

If the collection contains nested arrays or objects, you should pass a key that will be used to determine which values to sum:

```
$collection = collect([
    ['name' => 'JavaScript: The Good Parts', 'pages' => 176],
    ['name' => 'JavaScript: The Definitive Guide', 'pages' => 1096],
]);

$collection->sum('pages');

// 1272
```

In addition, you may pass your own closure to determine which values of the collection to sum:

```
$collection = collect([
    ['name' => 'Chair', 'colors' => ['Black']],
    ['name' => 'Desk', 'colors' => ['Black', 'Mahogany']],
    ['name' => 'Bookcase', 'colors' => ['Red', 'Beige', 'Brown']],
]);

$collection->sum(function ($product) {
    return count($product['colors']);
});

// 6
```

`take()` {.collection-method}

The `take` method returns a new collection with the specified number of items:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(3);

$chunk->all();

// [0, 1, 2]
```

You may also pass a negative integer to take the specified number of items from the end of the collection:

```
$collection = collect([0, 1, 2, 3, 4, 5]);

$chunk = $collection->take(-2);

$chunk->all();

// [4, 5]
```

`takeUntil()` {.collection-method}

The `takeUntil` method returns items in the collection until the given callback returns `true`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(function ($item) {
    return $item >= 3;
});

$subset->all();

// [1, 2]
```

You may also pass a simple value to the `takeUntil` method to get the items until the given value is found:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeUntil(3);

$subset->all();

// [1, 2]
```

{note} If the given value is not found or the callback never returns `true`, the `takeUntil` method will return all items in the collection.

`takeWhile()` {.collection-method}

The `takeWhile` method returns items in the collection until the given callback returns `false`:

```
$collection = collect([1, 2, 3, 4]);

$subset = $collection->takeWhile(function ($item) {
    return $item < 3;
});

$subset->all();

// [1, 2]
```

{note} If the callback never returns `false`, the `takeWhile` method will return all items in the collection.

`tap()` {.collection-method}

The `tap` method passes the collection to the given callback, allowing you to "tap" into the collection at a specific point and do something with the items while not affecting the collection itself. The collection is then returned by the `tap` method:

```
collect([2, 4, 3, 1, 5])
    ->sort()
    ->tap(function ($collection) {
        Log::debug('Values after sorting',
$collection->values()->all());
    })
    ->shift();

// 1
```

`times()` {.collection-method}

The static `times` method creates a new collection by invoking the given closure a specified number of times:

```
$collection = Collection::times(10, function ($number) {
    return $number * 9;
});

$collection->all();

// [9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
```

`toArray()` {.collection-method}

The `toArray` method converts the collection into a plain PHP `array`. If the collection's values are Eloquent models, the models will also be converted to arrays:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toArray();

/*
    [
        ['name' => 'Desk', 'price' => 200],
    ]
*/
```

{note} `toArray` also converts all of the collection's nested objects that are an instance of `Arrayable` to an array. If you want to get the raw array underlying the collection, use the `all` method instead.

`toJson()` {.collection-method}

The `toJson` method converts the collection into a JSON serialized string:

```
$collection = collect(['name' => 'Desk', 'price' => 200]);

$collection->toJson();

// '{"name":"Desk", "price":200}'
```

`transform()` {.collection-method}

The `transform` method iterates over the collection and calls the given callback with each item in the collection. The items in the collection will be replaced by the values returned by the callback:

```
$collection = collect([1, 2, 3, 4, 5]);

$collection->transform(function ($item, $key) {
    return $item * 2;
});

$collection->all();

// [2, 4, 6, 8, 10]
```

{note} Unlike most other collection methods, `transform` modifies the collection itself. If you wish to create a new collection instead, use the `map` method.

`union()` {.collection-method}

The `union` method adds the given array to the collection. If the given array contains keys that are already in the original collection, the original collection's values will be preferred:

```
$collection = collect([1 => ['a'], 2 => ['b']]);

$union = $collection->union([3 => ['c'], 1 => ['d']]);

$union->all();

// [1 => ['a'], 2 => ['b'], 3 => ['c']]
```

`unique()` {.collection-method}

The `unique` method returns all of the unique items in the collection. The returned collection keeps the original array keys, so in the following example we will use the `values` method to reset the keys to consecutively numbered indexes:

```
$collection = collect([1, 1, 2, 2, 3, 4, 2]);

$unique = $collection->unique();

$unique->values()->all();

// [1, 2, 3, 4]
```

When dealing with nested arrays or objects, you may specify the key used to determine uniqueness:

```
$collection = collect([
    ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'iPhone 5', 'brand' => 'Apple', 'type' => 'phone'],
    ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' => 'watch'],
    ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' => 'phone'],
    ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' => 'watch'],
]);

$unique = $collection->unique('brand');

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' =>
'phone'],
    ]
*/
```

Finally, you may also pass your own closure to the `unique` method to specify which value should determine an item's uniqueness:

```
$unique = $collection->unique(function ($item) {
    return $item['brand'].$item['type'];
});

$unique->values()->all();

/*
    [
        ['name' => 'iPhone 6', 'brand' => 'Apple', 'type' => 'phone'],
        ['name' => 'Apple Watch', 'brand' => 'Apple', 'type' =>
'watch'],
        ['name' => 'Galaxy S6', 'brand' => 'Samsung', 'type' =>
'phone'],
        ['name' => 'Galaxy Gear', 'brand' => 'Samsung', 'type' =>
'watch'],
    ]
*/
```

The `unique` method uses "loose" comparisons when checking item values, meaning a string

with an integer value will be considered equal to an integer of the same value. Use the `uniqueStrict` method to filter using "strict" comparisons.

{tip} This method's behavior is modified when using Eloquent Collections.

`uniqueStrict()` {.collection-method}

This method has the same signature as the `unique` method; however, all values are compared using "strict" comparisons.

`unless()` {.collection-method}

The `unless` method will execute the given callback unless the first argument given to the method evaluates to `true`:

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
    return $collection->push(4);
});

$collection->unless(false, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

A second callback may be passed to the `unless` method. The second callback will be executed when the first argument given to the `unless` method evaluates to `true`:

```
$collection = collect([1, 2, 3]);

$collection->unless(true, function ($collection) {
    return $collection->push(4);
}, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

For the inverse of `unless`, see the `when` method.

`unlessEmpty()` {.collection-method}

Alias for the `whenNotEmpty` method.

`unlessNotEmpty()` {.collection-method}

Alias for the `whenEmpty` method.

`unwrap()` {.collection-method}

The static `unwrap` method returns the collection's underlying items from the given value when applicable:

```
Collection::unwrap(collect('John Doe'));

// ['John Doe']

Collection::unwrap(['John Doe']);

// ['John Doe']

Collection::unwrap('John Doe');

// 'John Doe'
```

`values()` {.collection-method}

The `values` method returns a new collection with the keys reset to consecutive integers:

```
$collection = collect([
    10 => ['product' => 'Desk', 'price' => 200],
    11 => ['product' => 'Desk', 'price' => 200],
]);

$values = $collection->values();

$values->all();

/*
    [
        0 => ['product' => 'Desk', 'price' => 200],
        1 => ['product' => 'Desk', 'price' => 200],
    ]
*/
```

`when()` {.collection-method}

The `when` method will execute the given callback when the first argument given to the method evaluates to `true`:

```
$collection = collect([1, 2, 3]);

$collection->when(true, function ($collection) {
    return $collection->push(4);
});

$collection->when(false, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 4]
```

A second callback may be passed to the `when` method. The second callback will be executed when the first argument given to the `when` method evaluates to `false`:

```
$collection = collect([1, 2, 3]);

$collection->when(false, function ($collection) {
    return $collection->push(4);
}, function ($collection) {
    return $collection->push(5);
});

$collection->all();

// [1, 2, 3, 5]
```

For the inverse of `when`, see the `unless` method.

`whenEmpty()` {.collection-method}

The `whenEmpty` method will execute the given callback when the collection is empty:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function ($collection) {
    return $collection->push('Adam');
});

$collection->all();

// ['Michael', 'Tom']


$collection = collect();

$collection->whenEmpty(function ($collection) {
    return $collection->push('Adam');
});

$collection->all();

// ['Adam']
```

A second closure may be passed to the `whenEmpty` method that will be executed when the collection is not empty:

```
$collection = collect(['Michael', 'Tom']);

$collection->whenEmpty(function ($collection) {
    return $collection->push('Adam');
}, function ($collection) {
    return $collection->push('Taylor');
});

$collection->all();

// ['Michael', 'Tom', 'Taylor']
```

For the inverse of `whenEmpty`, see the `whenNotEmpty` method.

`whenNotEmpty()` {.collection-method}

The `whenNotEmpty` method will execute the given callback when the collection is not empty:

```
$collection = collect(['michael', 'tom']);

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// ['michael', 'tom', 'adam']


$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
});

$collection->all();

// []
```

A second closure may be passed to the `whenNotEmpty` method that will be executed when

the collection is empty:

```
$collection = collect();

$collection->whenNotEmpty(function ($collection) {
    return $collection->push('adam');
}, function ($collection) {
    return $collection->push('taylor');
});

$collection->all();

// ['taylor']
```

For the inverse of `whenNotEmpty`, see the `whenEmpty` method.

`where()` {.collection-method}

The `where` method filters the collection by a given key / value pair:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->where('price', 100);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `where` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereStrict` method to filter using "strict" comparisons.

Optionally, you may pass a comparison operator as the second parameter.

```
$collection = collect([
    ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
    ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ['name' => 'Sue', 'deleted_at' => null],
]);

$filtered = $collection->where('deleted_at', '!=', null);

$filtered->all();

/*
    [
        ['name' => 'Jim', 'deleted_at' => '2019-01-01 00:00:00'],
        ['name' => 'Sally', 'deleted_at' => '2019-01-02 00:00:00'],
    ]
*/
```

`whereStrict()` {.collection-method}

This method has the same signature as the `where` method; however, all values are compared using "strict" comparisons.

`whereBetween()` {.collection-method}

The `whereBetween` method filters the collection by determining if a specified item value is within a given range:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereBetween('price', [100, 200]);

$filtered->all();

/*
    [
        ['product' => 'Desk', 'price' => 200],
        ['product' => 'Bookcase', 'price' => 150],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

`whereIn()` {.collection-method}

The `whereIn` method removes elements from the collection that do not have a specified item value that is contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Desk', 'price' => 200],
        ['product' => 'Bookcase', 'price' => 150],
    ]
*/
```

The `whereIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereInStrict` method to filter using "strict" comparisons.

`whereInStrict()` {.collection-method}

This method has the same signature as the `whereIn` method; however, all values are compared using "strict" comparisons.

`whereInstanceOf()` {.collection-method}

The `whereInstanceOf` method filters the collection by a given class type:

```
use App\Models\User;
use App\Models\Post;

$collection = collect([
    new User,
    new User,
    new Post,
]);

$filtered = $collection->whereInstanceOf(User::class);

$filtered->all();

// [App\Models\User, App\Models\User]
```

`whereNotBetween()` {.collection-method}

The `whereNotBetween` method filters the collection by determining if a specified item value is outside of a given range:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 80],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Pencil', 'price' => 30],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotBetween('price', [100, 200]);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 80],
        ['product' => 'Pencil', 'price' => 30],
    ]
*/
```

`whereNotIn()` {.collection-method}

The `whereNotIn` method removes elements from the collection that have a specified item value that is contained within the given array:

```
$collection = collect([
    ['product' => 'Desk', 'price' => 200],
    ['product' => 'Chair', 'price' => 100],
    ['product' => 'Bookcase', 'price' => 150],
    ['product' => 'Door', 'price' => 100],
]);

$filtered = $collection->whereNotIn('price', [150, 200]);

$filtered->all();

/*
    [
        ['product' => 'Chair', 'price' => 100],
        ['product' => 'Door', 'price' => 100],
    ]
*/
```

The `whereNotIn` method uses "loose" comparisons when checking item values, meaning a string with an integer value will be considered equal to an integer of the same value. Use the `whereNotInStrict` method to filter using "strict" comparisons.

`whereNotInStrict()` {.collection-method}

This method has the same signature as the `whereNotIn` method; however, all values are compared using "strict" comparisons.

`whereNotNull()` {.collection-method}

The `whereNotNull` method returns items from the collection where the given key is not `null`:

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNotNull('name');

$filtered->all();

/*
    [
        ['name' => 'Desk'],
        ['name' => 'Bookcase'],
    ]
*/
```

`whereNull()` {.collection-method}

The `whereNull` method returns items from the collection where the given key is `null`:

```
$collection = collect([
    ['name' => 'Desk'],
    ['name' => null],
    ['name' => 'Bookcase'],
]);

$filtered = $collection->whereNull('name');

$filtered->all();

/*
    [
        ['name' => null],
    ]
*/
```

`wrap()` {.collection-method}

The static `wrap` method wraps the given value in a collection when applicable:

```
use Illuminate\Support\Collection;

$collection = Collection::wrap('John Doe');

$collection->all();

// ['John Doe']

$collection = Collection::wrap(['John Doe']);

$collection->all();

// ['John Doe']

$collection = Collection::wrap(collect('John Doe'));

$collection->all();

// ['John Doe']
```

`zip()` {.collection-method}

The `zip` method merges together the values of the given array with the values of the original collection at their corresponding index:

```
$collection = collect(['Chair', 'Desk']);

$zipped = $collection->zip([100, 200]);

$zipped->all();

// [['Chair', 100], ['Desk', 200]]
```

Higher Order Messages

Collections also provide support for "higher order messages", which are short-cuts for performing common actions on collections. The collection methods that provide higher order messages are: `average`, `avg`, `contains`, `each`, `every`, `filter`, `first`, `flatMap`, `groupBy`, `keyBy`, `map`, `max`, `min`, `partition`, `reject`, `skipUntil`, `skipWhile`, `some`, `sortBy`, `sortByDesc`, `sum`, `takeUntil`, `takeWhile`, and `unique`.

Each higher order message can be accessed as a dynamic property on a collection instance. For instance, let's use the `each` higher order message to call a method on each object within a collection:

```
use App\Models\User;

$users = User::where('votes', '>', 500)->get();

$users->each->markAsVip();
```

Likewise, we can use the `sum` higher order message to gather the total number of "votes" for a collection of users:

```
$users = User::where('group', 'Development')->get();

return $users->sum->votes;
```

Lazy Collections

Introduction

{note} Before learning more about Laravel's lazy collections, take some time to familiarize yourself with PHP generators.

To supplement the already powerful `Collection` class, the `LazyCollection` class leverages PHP's generators to allow you to work with very large datasets while keeping memory usage low.

For example, imagine your application needs to process a multi-gigabyte log file while taking advantage of Laravel's collection methods to parse the logs. Instead of reading the entire file into memory at once, lazy collections may be used to keep only a small part of the file in memory at a given time:

```
use App\Models\LogEntry;
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
})->chunk(4)->map(function ($lines) {
    return LogEntry::fromLines($lines);
})->each(function (LogEntry $logEntry) {
    // Process the log entry...
});
```

Or, imagine you need to iterate through 10,000 Eloquent models. When using traditional Laravel collections, all 10,000 Eloquent models must be loaded into memory at the same time:

```
use App\Models\User;

$users = User::all()->filter(function ($user) {
    return $user->id > 500;
});
```

However, the query builder's `cursor` method returns a `LazyCollection` instance. This allows you to still only run a single query against the database but also only keep one Eloquent model loaded in memory at a time. In this example, the `filter` callback is not executed until we actually iterate over each user individually, allowing for a drastic reduction in memory usage:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Creating Lazy Collections

To create a lazy collection instance, you should pass a PHP generator function to the collection's `make` method:

```
use Illuminate\Support\LazyCollection;

LazyCollection::make(function () {
    $handle = fopen('log.txt', 'r');

    while (($line = fgets($handle)) !== false) {
        yield $line;
    }
});
```

The Enumerable Contract

Almost all methods available on the `Collection` class are also available on the `LazyCollection` class. Both of these classes implement the `Illuminate\Support\Enumerable` contract, which defines the following methods:

all average avg chunk chunkWhile collapse collect combine concat contains containsStrict count countBy crossJoin dd diff diffAssoc diffKeys dump duplicates duplicatesStrict each eachSpread every except filter first firstWhere flatMap flatten flip forPage get groupBy has implode intersect intersectByKeys isEmpty isNotEmpty join keyBy keys last macro make map mapInto mapSpread mapToGroups mapWithKeys max median merge mergeRecursive min mode nth only pad partition pipe pluck random reduce reject replace replaceRecursive reverse search shuffle skip slice some sort sortBy sortByDesc sortKeys sortKeysDesc split sum take tap times toArray toJson union unique uniqueStrict unless unlessEmpty unlessNotEmpty unwrap values when whenEmpty whenNotEmpty where whereStrict whereBetween whereIn whereInStrict whereInstanceOf whereNotBetween whereNotIn whereNotInStrict wrap zip

{note} Methods that mutate the collection (such as `shift`, `pop`, `prepend` etc.) are **not** available on the `LazyCollection` class.

Lazy Collection Methods

In addition to the methods defined in the `Enumerable` contract, the `LazyCollection` class contains the following methods:

`takeUntilTimeout()` {.collection-method}

The `takeUntilTimeout` method returns a new lazy collection that will enumerate values until the specified time. After that time, the collection will then stop enumerating:

```
$lazyCollection = LazyCollection::times(INF)
    ->takeUntilTimeout(now()->addMinute());

$lazyCollection->each(function ($number) {
    dump($number);

    sleep(1);
});

// 1
// 2
// ...
// 58
// 59
```

To illustrate the usage of this method, imagine an application that submits invoices from the database using a cursor. You could define a scheduled task that runs every 15 minutes and only processes invoices for a maximum of 14 minutes:

```
use App\Models\Invoice;
use Illuminate\Support\Carbon;

Invoice::pending()->cursor()
    ->takeUntilTimeout(
        Carbon::createFromTimestamp(LARAVEL_START)->add(14, 'minutes')
    )
    ->each(fn ($invoice) => $invoice->submit());
```

`tapEach()` {.collection-method}

While the `each` method calls the given callback for each item in the collection right away, the `tapEach` method only calls the given callback as the items are being pulled out of the list one by one:

```
// Nothing has been dumped so far...
$lazyCollection = LazyCollection::times(INF)->tapEach(function ($value)
{
    dump($value);
});

// Three items are dumped...
$array = $lazyCollection->take(3)->all();

// 1
// 2
// 3
```

`remember()` {.collection-method}

The `remember` method returns a new lazy collection that will remember any values that have already been enumerated and will not retrieve them again on subsequent collection enumerations:

```
// No query has been executed yet...
$users = User::cursor()->remember();

// The query is executed...
// The first 5 users are hydrated from the database...
$users->take(5)->all();

// First 5 users come from the collection's cache...
// The rest are hydrated from the database...
$users->take(20)->all();
```

Configuration

- Introduction
- Environment Configuration
    - Environment Variable Types
    - Retrieving Environment Configuration
    - Determining The Current Environment
- Accessing Configuration Values
- Configuration Caching

- Debug Mode
- Maintenance Mode

Introduction

All of the configuration files for the Laravel framework are stored in the `config` directory. Each option is documented, so feel free to look through the files and get familiar with the options available to you.

These configuration files allow you to configure things like your database connection information, your mail server information, as well as various other core configuration values such as your application timezone and encryption key.

Environment Configuration

It is often helpful to have different configuration values based on the environment where the application is running. For example, you may wish to use a different cache driver locally than you do on your production server.

To make this a cinch, Laravel utilizes the DotEnv PHP library. In a fresh Laravel installation, the root directory of your application will contain a `.env.example` file that defines many common environment variables. During the Laravel installation process, this file will automatically be copied to `.env`.

Laravel's default `.env` file contains some common configuration values that may differ based on whether your application is running locally or on a production web server. These values are then retrieved from various Laravel configuration files within the `config` directory using Laravel's `env` function.

If you are developing with a team, you may wish to continue including a `.env.example` file with your application. By putting placeholder values in the example configuration file, other developers on your team can clearly see which environment variables are needed to run your application.

{tip} Any variable in your `.env` file can be overridden by external environment variables such as server-level or system-level environment variables.

Environment File Security

Your `.env` file should not be committed to your application's source control, since each developer / server using your application could require a different environment configuration. Furthermore, this would be a security risk in the event an intruder gains access to your source control repository, since any sensitive credentials would get exposed.

Additional Environment Files

Before loading your application's environment variables, Laravel determines if either the `APP_ENV` environment variable has been externally provided or if the `--env` CLI argument has been specified. If so, Laravel will attempt to load an `.env.[APP_ENV]` file if it exists. If it does not exist, the default `.env` file will be loaded.

Environment Variable Types

All variables in your `.env` files are typically parsed as strings, so some reserved values have been created to allow you to return a wider range of types from the `env()` function:

`.env` Value	`env()` Value
true	(bool) true
(true)	(bool) true
false	(bool) false
(false)	(bool) false
empty	(string) ''
(empty)	(string) ''
null	(null) null
(null)	(null) null

If you need to define an environment variable with a value that contains spaces, you may do so by enclosing the value in double quotes:

```
APP_NAME="My Application"
```

Retrieving Environment Configuration

All of the variables listed in this file will be loaded into the `$_ENV` PHP super-global when your application receives a request. However, you may use the `env` helper to retrieve values from these variables in your configuration files. In fact, if you review the Laravel configuration files, you will notice many of the options are already using this helper:

```
'debug' => env('APP_DEBUG', false),
```

The second value passed to the `env` function is the "default value". This value will be returned if no environment variable exists for the given key.

Determining The Current Environment

The current application environment is determined via the `APP_ENV` variable from your `.env` file. You may access this value via the `environment` method on the `App` facade:

```
use Illuminate\Support\Facades\App;

$environment = App::environment();
```

You may also pass arguments to the `environment` method to determine if the environment matches a given value. The method will return `true` if the environment matches any of the given values:

```
if (App::environment('local')) {
    // The environment is local
}

if (App::environment(['local', 'staging'])) {
    // The environment is either local OR staging...
}
```

{tip} The current application environment detection can be overridden by defining a server-level `APP_ENV` environment variable.

Accessing Configuration Values

You may easily access your configuration values using the global `config` helper function from anywhere in your application. The configuration values may be accessed using "dot" syntax, which includes the name of the file and option you wish to access. A default value may also be specified and will be returned if the configuration option does not exist:

```
$value = config('app.timezone');

// Retrieve a default value if the configuration value does not exist...
$value = config('app.timezone', 'Asia/Seoul');
```

To set configuration values at runtime, pass an array to the `config` helper:

```
config(['app.timezone' => 'America/Chicago']);
```

Configuration Caching

To give your application a speed boost, you should cache all of your configuration files into a single file using the `config:cache` Artisan command. This will combine all of the configuration options for your application into a single file which can be quickly loaded by the framework.

You should typically run the `php artisan config:cache` command as part of your production deployment process. The command should not be run during local development as configuration options will frequently need to be changed during the course of your application's development.

{note} If you execute the `config:cache` command during your deployment process, you should be sure that you are only calling the `env` function from within your configuration files. Once the configuration has been cached, the `.env` file will not be loaded; therefore, the `env` function will only return external, system level environment variables.

Debug Mode

The `debug` option in your `config/app.php` configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the `APP_DEBUG` environment variable, which is stored in your `.env` file.

For local development, you should set the `APP_DEBUG` environment variable to `true`. **In your production environment, this value should always be `false`. If the variable is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.**

Maintenance Mode

When your application is in maintenance mode, a custom view will be displayed for all requests into your application. This makes it easy to "disable" your application while it is updating or when you are performing maintenance. A maintenance mode check is included in the default middleware stack for your application. If the application is in maintenance mode, a `Symfony\Component\HttpKernel\Exception\HttpException` instance will be thrown with a status code of 503.

To enable maintenance mode, execute the `down` Artisan command:

```
php artisan down
```

If you would like the `Refresh` HTTP header to be sent with all maintenance mode responses, you may provide the `refresh` option when invoking the `down` command. The `Refresh` header will instruct the browser to automatically refresh the page after the specified number of seconds:

```
php artisan down --refresh=15
```

You may also provide a `retry` option to the `down` command, which will be set as the `Retry-After` HTTP header's value, although browsers generally ignore this header:

```
php artisan down --retry=60
```

Bypassing Maintenance Mode

Even while in maintenance mode, you may use the `secret` option to specify a maintenance mode bypass token:

```
php artisan down --secret="1630542a-246b-4b66-afa1-dd72a4c43515"
```

After placing the application in maintenance mode, you may navigate to the application URL matching this token and Laravel will issue a maintenance mode bypass cookie to your browser:

```
https://example.com/1630542a-246b-4b66-afa1-dd72a4c43515
```

When accessing this hidden route, you will then be redirected to the `/` route of the application. Once the cookie has been issued to your browser, you will be able to browse the application normally as if it was not in maintenance mode.

Pre-Rendering The Maintenance Mode View

If you utilize the `php artisan down` command during deployment, your users may still occasionally encounter errors if they access the application while your Composer dependencies or other infrastructure components are updating. This occurs because a significant part of the Laravel framework must boot in order to determine your application is in maintenance mode and render the maintenance mode view using the templating engine.

For this reason, Laravel allows you to pre-render a maintenance mode view that will be returned at the very beginning of the request cycle. This view is rendered before any of your application's dependencies have loaded. You may pre-render a template of your choice using the `down` command's `render` option:

```
php artisan down --render="errors::503"
```

Redirecting Maintenance Mode Requests

While in maintenance mode, Laravel will display the maintenance mode view for all application URLs the user attempts to access. If you wish, you may instruct Laravel to redirect all requests to a specific URL. This may be accomplished using the `redirect` option. For example, you may wish to redirect all requests to the `/` URI:

```
php artisan down --redirect=/
```

Disabling Maintenance Mode

To disable maintenance mode, use the `up` command:

```
php artisan up
```

{tip} You may customize the default maintenance mode template by defining your own template at `resources/views/errors/503.blade.php`.

Maintenance Mode & Queues

While your application is in maintenance mode, no queued jobs will be handled. The jobs will continue to be handled as normal once the application is out of maintenance mode.

Alternatives To Maintenance Mode

Since maintenance mode requires your application to have several seconds of downtime, consider alternatives like Laravel Vapor and Envoyer to accomplish zero-downtime deployment with Laravel.

Controllers

- Introduction
- Writing Controllers
    - Basic Controllers
    - Single Action Controllers
- Controller Middleware
- Resource Controllers
    - Partial Resource Routes
    - Nested Resources
    - Naming Resource Routes
    - Naming Resource Route Parameters
    - Scoping Resource Routes
    - Localizing Resource URIs
    - Supplementing Resource Controllers
- Dependency Injection & Controllers

Introduction

Instead of defining all of your request handling logic as closures in your route files, you may wish to organize this behavior using "controller" classes. Controllers can group related request handling logic into a single class. For example, a `UserController` class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the `app/Http/Controllers` directory.

Writing Controllers

Basic Controllers

Let's take a look at an example of a basic controller. Note that the controller extends the base controller class included with Laravel: `App\Http\Controllers\Controller`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param  int  $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

You can define a route to this controller method like so:

```
use App\Http\Controllers\UserController;

Route::get('/user/{id}', [UserController::class, 'show']);
```

When an incoming request matches the specified route URI, the `show` method on the

`App\Http\Controllers\UserController` class will be invoked and the route parameters will be passed to the method.

{tip} Controllers are not **required** to extend a base class. However, you will not have access to convenient features such as the `middleware` and `authorize` methods.

Single Action Controllers

If a controller action is particularly complex, you might find it convenient to dedicate an entire controller class to that single action. To accomplish this, you may define a single `__invoke` method within the controller:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;

class ProvisionServer extends Controller
{
    /**
     * Provision a new web server.
     *
     * @return \Illuminate\Http\Response
     */
    public function __invoke()
    {
        // ...
    }
}
```

When registering routes for single action controllers, you do not need to specify a controller method. Instead, you may simply pass the name of the controller to the router:

```
use App\Http\Controllers\ProvisionServer;

Route::post('/server', ProvisionServer::class);
```

You may generate an invokable controller by using the `--invokable` option of the `make:controller` Artisan command:

```
php artisan make:controller ProvisionServer --invokable
```

{tip} Controller stubs may be customized using stub publishing.

Controller Middleware

Middleware may be assigned to the controller's routes in your route files:

```
Route::get('profile', [UserController::class,
'show'])->middleware('auth');
```

Or, you may find it convenient to specify middleware within your controller's constructor. Using the `middleware` method within your controller's constructor, you can assign middleware to the controller's actions:

```
class UserController extends Controller
{
    /**
     * Instantiate a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log')->only('index');
        $this->middleware('subscribed')->except('store');
    }
}
```

Controllers also allow you to register middleware using a closure. This provides a convenient way to define an inline middleware for a single controller without defining an entire middleware class:

```
$this->middleware(function ($request, $next) {
    return $next($request);
});
```

Resource Controllers

If you think of each Eloquent model in your application as a "resource", it is typical to perform the same sets of actions against each resource in your application. For example, imagine your application contains a `Photo` model and a `Movie` model. It is likely that users can create, read, update, or delete these resources.

Because of this common use case, Laravel resource routing assigns the typical create, read, update, and delete ("CRUD") routes to a controller with a single line of code. To get started, we can use the `make:controller` Artisan command's `--resource` option to quickly create a controller to handle these actions:

```
php artisan make:controller PhotoController --resource
```

This command will generate a controller at `app/Http/Controllers/PhotoController.php`. The controller will contain a method for each of the available resource operations. Next, you may register a resource route that points to the controller:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class);
```

This single route declaration creates multiple routes to handle a variety of actions on the resource. The generated controller will already have methods stubbed for each of these actions. Remember, you can always get a quick overview of your application's routes by running the `route:list` Artisan command.

You may even register many resource controllers at once by passing an array to the `resources` method:

```
Route::resources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	`/photos`	index	photos.index
GET	`/photos/create`	create	photos.create
POST	`/photos`	store	photos.store
GET	`/photos/{photo}`	show	photos.show
GET	`/photos/{photo}/edit`	edit	photos.edit
PUT/PATCH	`/photos/{photo}`	update	photos.update
DELETE	`/photos/{photo}`	destroy	photos.destroy

Customizing Missing Model Behavior

Typically, a 404 HTTP response will be generated if an implicitly bound resource model is not found. However, you may customize this behavior by calling the `missing` method when defining your resource route. The `missing` method accepts a closure that will be invoked if an implicitly bound model can not be found for any of the resource's routes:

```
use App\Http\Controllers\PhotoController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::resource('photos', PhotoController::class)
        ->missing(function (Request $request) {
            return Redirect::route('photos.index');
        });
```

Specifying The Resource Model

If you are using route model binding and would like the resource controller's methods to type-hint a model instance, you may use the `--model` option when generating the controller:

```
php artisan make:controller PhotoController --model=Photo --resource
```

Generating Form Requests

You may provide the `--requests` option when generating a resource controller to instruct Artisan to generate form request classes for the controller's storage and update methods:

```
php artisan make:controller PhotoController --model=Photo --resource --
requests
```

Partial Resource Routes

When declaring a resource route, you may specify a subset of actions the controller should handle instead of the full set of default actions:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('photos', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

API Resource Routes

When declaring resource routes that will be consumed by APIs, you will commonly want to exclude routes that present HTML templates such as `create` and `edit`. For convenience, you may use the `apiResource` method to automatically exclude these two routes:

```
use App\Http\Controllers\PhotoController;

Route::apiResource('photos', PhotoController::class);
```

You may register many API resource controllers at once by passing an array to the `apiResources` method:

```
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\PostController;

Route::apiResources([
    'photos' => PhotoController::class,
    'posts' => PostController::class,
]);
```

To quickly generate an API resource controller that does not include the `create` or `edit` methods, use the `--api` switch when executing the `make:controller` command:

```
php artisan make:controller PhotoController --api
```

Nested Resources

Sometimes you may need to define routes to a nested resource. For example, a photo resource may have multiple comments that may be attached to the photo. To nest the resource controllers, you may use "dot" notation in your route declaration:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments', PhotoCommentController::class);
```

This route will register a nested resource that may be accessed with URIs like the following:

```
/photos/{photo}/comments/{comment}
```

Scoping Nested Resources

Laravel's implicit model binding feature can automatically scope nested bindings such that the resolved child model is confirmed to belong to the parent model. By using the `scoped` method when defining your nested resource, you may enable automatic scoping as well as instruct Laravel which field the child resource should be retrieved by. For more information on how to accomplish this, please see the documentation on scoping resource routes.

Shallow Nesting

Often, it is not entirely necessary to have both the parent and the child IDs within a URI since the child ID is already a unique identifier. When using unique identifiers such as auto-incrementing primary keys to identify your models in URI segments, you may choose to use "shallow nesting":

```
use App\Http\Controllers\CommentController;

Route::resource('photos.comments', CommentController::class)->shallow();
```

This route definition will define the following routes:

Verb	URI	Action	Route Name
GET	`/photos/{photo}/comments`	index	photos.comments.index
GET	`/photos/{photo}/comments/create`	create	photos.comments.create
POST	`/photos/{photo}/comments`	store	photos.comments.store
GET	`/comments/{comment}`	show	comments.show
GET	`/comments/{comment}/edit`	edit	comments.edit
PUT/PATCH	`/comments/{comment}`	update	comments.update
DELETE	`/comments/{comment}`	destroy	comments.destroy

Naming Resource Routes

By default, all resource controller actions have a route name; however, you can override these names by passing a `names` array with your desired route names:

```
use App\Http\Controllers\PhotoController;

Route::resource('photos', PhotoController::class)->names([
    'create' => 'photos.build'
]);
```

Naming Resource Route Parameters

By default, `Route::resource` will create the route parameters for your resource routes based on the "singularized" version of the resource name. You can easily override this on a per resource basis using the `parameters` method. The array passed into the `parameters` method should be an associative array of resource names and parameter names:

```
use App\Http\Controllers\AdminUserController;

Route::resource('users', AdminUserController::class)->parameters([
    'users' => 'admin_user'
]);
```

The example above generates the following URI for the resource's `show` route:

```
/users/{admin_user}
```

Scoping Resource Routes

Laravel's scoped implicit model binding feature can automatically scope nested bindings such that the resolved child model is confirmed to belong to the parent model. By using the `scoped` method when defining your nested resource, you may enable automatic scoping as well as instruct Laravel which field the child resource should be retrieved by:

```
use App\Http\Controllers\PhotoCommentController;

Route::resource('photos.comments',
PhotoCommentController::class)->scoped([
    'comment' => 'slug',
]);
```

This route will register a scoped nested resource that may be accessed with URIs like the following:

```
/photos/{photo}/comments/{comment:slug}
```

When using a custom keyed implicit binding as a nested route parameter, Laravel will automatically scope the query to retrieve the nested model by its parent using conventions to guess the relationship name on the parent. In this case, it will be assumed that the `Photo` model has a relationship named `comments` (the plural of the route parameter name) which can be used to retrieve the `Comment` model.

Localizing Resource URIs

By default, `Route::resource` will create resource URIs using English verbs. If you need to localize the `create` and `edit` action verbs, you may use the `Route::resourceVerbs` method. This may be done at the beginning of the `boot` method within your application's `App\Providers\RouteServiceProvider`:

```
/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::resourceVerbs([
        'create' => 'crear',
        'edit' => 'editar',
    ]);

    // ...
}
```

Once the verbs have been customized, a resource route registration such as `Route::resource('fotos', PhotoController::class)` will produce the following URIs:

```
/fotos/crear

/fotos/{foto}/editar
```

Supplementing Resource Controllers

If you need to add additional routes to a resource controller beyond the default set of resource routes, you should define those routes before your call to the `Route::resource` method; otherwise, the routes defined by the `resource` method may unintentionally take precedence over your supplemental routes:

```
use App\Http\Controller\PhotoController;

Route::get('/photos/popular', [PhotoController::class, 'popular']);
Route::resource('photos', PhotoController::class);
```

{tip} Remember to keep your controllers focused. If you find yourself routinely needing methods outside of the typical set of resource actions, consider splitting your controller into two, smaller controllers.

Dependency Injection & Controllers

Constructor Injection

The Laravel service container is used to resolve all Laravel controllers. As a result, you are able to type-hint any dependencies your controller may need in its constructor. The declared dependencies will automatically be resolved and injected into the controller instance:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  \App\Repositories\UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

Method Injection

In addition to constructor injection, you may also type-hint dependencies on your controller's methods. A common use-case for method injection is injecting the

`Illuminate\Http\Request` instance into your controller methods:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $name = $request->name;

        //
    }
}
```

If your controller method is also expecting input from a route parameter, list your route arguments after your other dependencies. For example, if your route is defined like so:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

You may still type-hint the `Illuminate\Http\Request` and access your `id` parameter by defining your controller method as follows:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the given user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  string  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Database: Getting Started

- Introduction
    - Configuration
    - Read & Write Connections
- Running SQL Queries
    - Using Multiple Database Connections
    - Listening For Query Events
- Database Transactions
- Connecting To The Database CLI

Introduction

Almost every modern web application interacts with a database. Laravel makes interacting with databases extremely simple across a variety of supported databases using raw SQL, a fluent query builder, and the Eloquent ORM. Currently, Laravel provides first-party support for four databases:

- MySQL 5.7+ ([Version Policy](https://en.wikipedia.org/wiki/MySQL#Release_history)) - PostgreSQL 9.6+ ([Version Policy](https://www.postgresql.org/support/versioning/)) - SQLite 3.8.8+ - SQL Server 2017+ ([Version Policy](https://docs.microsoft.com/en-us/lifecycle/products/?products=sql-server))

Configuration

The configuration for Laravel's database services is located in your application's `config/database.php` configuration file. In this file, you may define all of your database connections, as well as specify which connection should be used by default. Most of the configuration options within this file are driven by the values of your application's environment variables. Examples for most of Laravel's supported database systems are provided in this file.

By default, Laravel's sample environment configuration is ready to use with Laravel Sail, which is a Docker configuration for developing Laravel applications on your local machine. However, you are free to modify your database configuration as needed for your local database.

SQLite Configuration

SQLite databases are contained within a single file on your filesystem. You can create a new SQLite database using the `touch` command in your terminal: `touch database/database.sqlite`. After the database has been created, you may easily configure your environment variables to point to this database by placing the absolute path to the database in the `DB_DATABASE` environment variable:

```
DB_CONNECTION=sqlite
DB_DATABASE=/absolute/path/to/database.sqlite
```

To enable foreign key constraints for SQLite connections, you should set the `DB_FOREIGN_KEYS` environment variable to `true`:

```
DB_FOREIGN_KEYS=true
```

Microsoft SQL Server Configuration

To use a Microsoft SQL Server database, you should ensure that you have the `sqlsrv` and `pdo_sqlsrv` PHP extensions installed as well as any dependencies they may require such as the Microsoft SQL ODBC driver.

Configuration Using URLs

Typically, database connections are configured using multiple configuration values such as `host`, `database`, `username`, `password`, etc. Each of these configuration values has its own corresponding environment variable. This means that when configuring your database connection information on a production server, you need to manage several environment variables.

Some managed database providers such as AWS and Heroku provide a single database "URL" that contains all of the connection information for the database in a single string. An example database URL may look something like the following:

```
mysql://root:password@127.0.0.1/forge?charset=UTF-8
```

These URLs typically follow a standard schema convention:

```
driver://username:password@host:port/database?options
```

For convenience, Laravel supports these URLs as an alternative to configuring your database with multiple configuration options. If the `url` (or corresponding `DATABASE_URL` environment variable) configuration option is present, it will be used to extract the database connection and credential information.

Read & Write Connections

Sometimes you may wish to use one database connection for SELECT statements, and another for INSERT, UPDATE, and DELETE statements. Laravel makes this a breeze, and the proper connections will always be used whether you are using raw queries, the query builder, or the Eloquent ORM.

To see how read / write connections should be configured, let's look at this example:

```
'mysql' => [
    'read' => [
        'host' => [
            '192.168.1.1',
            '196.168.1.2',
        ],
    ],
    'write' => [
        'host' => [
            '196.168.1.3',
        ],
    ],
    'sticky' => true,
    'driver' => 'mysql',
    'database' => 'database',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8mb4',
    'collation' => 'utf8mb4_unicode_ci',
    'prefix' => '',
],
```

Note that three keys have been added to the configuration array: `read`, `write` and `sticky`. The `read` and `write` keys have array values containing a single key: `host`. The rest of the database options for the `read` and `write` connections will be merged from the main `mysql` configuration array.

You only need to place items in the `read` and `write` arrays if you wish to override the values from the main `mysql` array. So, in this case, `192.168.1.1` will be used as the host for the "read" connection, while `192.168.1.3` will be used for the "write" connection. The database credentials, prefix, character set, and all other options in the main `mysql` array will be shared across both connections. When multiple values exist in the `host` configuration array, a database host will be randomly chosen for each request.

The `sticky` Option

The `sticky` option is an *optional* value that can be used to allow the immediate reading of records that have been written to the database during the current request cycle. If the `sticky` option is enabled and a "write" operation has been performed against the database during the current request cycle, any further "read" operations will use the "write" connection. This ensures that any data written during the request cycle can be immediately read back from the database during that same request. It is up to you to decide if this is the

desired behavior for your application.

Running SQL Queries

Once you have configured your database connection, you may run queries using the `DB` facade. The `DB` facade provides methods for each type of query: `select`, `update`, `insert`, `delete`, and `statement`.

Running A Select Query

To run a basic SELECT query, you may use the `select` method on the `DB` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = DB::select('select * from users where active = ?',
[1]);

        return view('user.index', ['users' => $users]);
    }
}
```

The first argument passed to the `select` method is the SQL query, while the second argument is any parameter bindings that need to be bound to the query. Typically, these are the values of the `where` clause constraints. Parameter binding provides protection against SQL injection.

The `select` method will always return an `array` of results. Each result within the array will be a PHP `stdClass` object representing a record from the database:

```
use Illuminate\Support\Facades\DB;

$users = DB::select('select * from users');

foreach ($users as $user) {
    echo $user->name;
}
```

Using Named Bindings

Instead of using `?` to represent your parameter bindings, you may execute a query using named bindings:

```
$results = DB::select('select * from users where id = :id', ['id' =>
1]);
```

Running An Insert Statement

To execute an `insert` statement, you may use the `insert` method on the `DB` facade. Like `select`, this method accepts the SQL query as its first argument and bindings as its second argument:

```
use Illuminate\Support\Facades\DB;

DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

Running An Update Statement

The `update` method should be used to update existing records in the database. The number of rows affected by the statement is returned by the method:

```
use Illuminate\Support\Facades\DB;

$affected = DB::update(
    'update users set votes = 100 where name = ?',
    ['Anita']
);
```

Running A Delete Statement

The `delete` method should be used to delete records from the database. Like `update`, the number of rows affected will be returned by the method:

```
use Illuminate\Support\Facades\DB;

$deleted = DB::delete('delete from users');
```

Running A General Statement

Some database statements do not return any value. For these types of operations, you may use the `statement` method on the `DB` facade:

```
DB::statement('drop table users');
```

Running An Unprepared Statement

Sometimes you may want to execute an SQL statement without binding any values. You may use the `DB` facade's `unprepared` method to accomplish this:

```
DB::unprepared('update users set votes = 100 where name = "Dries"');
```

{note} Since unprepared statements do not bind parameters, they may be vulnerable to SQL injection. You should never allow user controlled values within an unprepared statement.

Implicit Commits

When using the `DB` facade's `statement` and `unprepared` methods within transactions you must be careful to avoid statements that cause implicit commits. These statements will cause the database engine to indirectly commit the entire transaction, leaving Laravel unaware of the database's transaction level. An example of such a statement is creating a database table:

```
DB::unprepared('create table a (col varchar(1) null)');
```

Please refer to the MySQL manual for a list of all statements that trigger implicit commits.

Using Multiple Database Connections

If your application defines multiple connections in your `config/database.php` configuration file, you may access each connection via the `connection` method provided by the `DB` facade. The connection name passed to the `connection` method should correspond to one of the connections listed in your `config/database.php` configuration file or configured at runtime using the `config` helper:

```
use Illuminate\Support\Facades\DB;

$users = DB::connection('sqlite')->select(...);
```

You may access the raw, underlying PDO instance of a connection using the `getPdo` method on a connection instance:

```
$pdo = DB::connection()->getPdo();
```

Listening For Query Events

If you would like to specify a closure that is invoked for each SQL query executed by your application, you may use the `DB` facade's `listen` method. This method can be useful for logging queries or debugging. You may register your query listener closure in the `boot` method of a service provider:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\DB;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        DB::listen(function ($query) {
            // $query->sql;
            // $query->bindings;
            // $query->time;
        });
    }
}
```

Database Transactions

You may use the `transaction` method provided by the `DB` facade to run a set of operations within a database transaction. If an exception is thrown within the transaction closure, the transaction will automatically be rolled back and the exception is re-thrown. If the closure executes successfully, the transaction will automatically be committed. You don't need to worry about manually rolling back or committing while using the `transaction` method:

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
});
```

Handling Deadlocks

The `transaction` method accepts an optional second argument which defines the number of times a transaction should be retried when a deadlock occurs. Once these attempts have been exhausted, an exception will be thrown:

```
use Illuminate\Support\Facades\DB;

DB::transaction(function () {
    DB::update('update users set votes = 1');

    DB::delete('delete from posts');
}, 5);
```

Manually Using Transactions

If you would like to begin a transaction manually and have complete control over rollbacks and commits, you may use the `beginTransaction` method provided by the `DB` facade:

```
use Illuminate\Support\Facades\DB;

DB::beginTransaction();
```

You can rollback the transaction via the `rollBack` method:

```
DB::rollBack();
```

Lastly, you can commit a transaction via the `commit` method:

```
DB::commit();
```

{tip} The `DB` facade's transaction methods control the transactions for both the query builder and Eloquent ORM.

Connecting To The Database CLI

If you would like to connect to your database's CLI, you may use the db Artisan command:

```
php artisan db
```

If needed, you may specify a database connection name to connect to a database connection that is not the default connection:

```
php artisan db mysql
```

Eloquent: Collections

- Introduction
- Available Methods
- Custom Collections

Introduction

All Eloquent methods that return more than one model result will return instances of the `Illuminate\Database\Eloquent\Collection` class, including results retrieved via the `get` method or accessed via a relationship. The Eloquent collection object extends Laravel's base collection, so it naturally inherits dozens of methods used to fluently work with the underlying array of Eloquent models. Be sure to review the Laravel collection documentation to learn all about these helpful methods!

All collections also serve as iterators, allowing you to loop over them as if they were simple PHP arrays:

```
use App\Models\User;

$users = User::where('active', 1)->get();

foreach ($users as $user) {
    echo $user->name;
}
```

However, as previously mentioned, collections are much more powerful than arrays and expose a variety of map / reduce operations that may be chained using an intuitive interface. For example, we may remove all inactive models and then gather the first name for each remaining user:

```
$names = User::all()->reject(function ($user) {
    return $user->active === false;
})->map(function ($user) {
    return $user->name;
});
```

Eloquent Collection Conversion

While most Eloquent collection methods return a new instance of an Eloquent collection, the `collapse`, `flatten`, `flip`, `keys`, `pluck`, and `zip` methods return a base collection instance. Likewise, if a `map` operation returns a collection that does not contain any Eloquent models, it will be converted to a base collection instance.

Available Methods

All Eloquent collections extend the base Laravel collection object; therefore, they inherit all of the powerful methods provided by the base collection class.

In addition, the `Illuminate\Database\Eloquent\Collection` class provides a superset of methods to aid with managing your model collections. Most methods return `Illuminate\Database\Eloquent\Collection` instances; however, some methods, like `modelKeys`, return an `Illuminate\Support\Collection` instance.

contains diff except find fresh intersect load loadMissing modelKeys makeVisible makeHidden only toQuery unique

`contains($key, $operator = null, $value = null)` {.collection-method .first-collection-method}

The `contains` method may be used to determine if a given model instance is contained by the collection. This method accepts a primary key or a model instance:

```
$users->contains(1);

$users->contains(User::find(1));
```

`diff($items)` {.collection-method}

The `diff` method returns all of the models that are not present in the given collection:

```
use App\Models\User;

$users = $users->diff(User::whereIn('id', [1, 2, 3])->get());
```

`except($keys)` {.collection-method}

The `except` method returns all of the models that do not have the given primary keys:

```
$users = $users->except([1, 2, 3]);
```

`find($key)` {.collection-method}

The `find` method returns the model that has a primary key matching the given key. If `$key` is a model instance, `find` will attempt to return a model matching the primary key. If `$key` is an array of keys, `find` will return all models which have a primary key in the given array:

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

$user = $users->find(1);
```

`fresh($with = [])` {.collection-method}

The `fresh` method retrieves a fresh instance of each model in the collection from the database. In addition, any specified relationships will be eager loaded:

```
$users = $users->fresh();

$users = $users->fresh('comments');
```

`intersect($items)` {.collection-method}

The `intersect` method returns all of the models that are also present in the given collection:

```
use App\Models\User;

$users = $users->intersect(User::whereIn('id', [1, 2, 3])->get());
```

`load($relations)` {.collection-method}

The `load` method eager loads the given relationships for all models in the collection:

```
$users->load(['comments', 'posts']);

$users->load('comments.author');
```

`loadMissing($relations)` {.collection-method}

The `loadMissing` method eager loads the given relationships for all models in the collection if the relationships are not already loaded:

```
$users->loadMissing(['comments', 'posts']);

$users->loadMissing('comments.author');
```

`modelKeys()` {.collection-method}

The `modelKeys` method returns the primary keys for all models in the collection:

```
$users->modelKeys();

// [1, 2, 3, 4, 5]
```

`makeVisible($attributes)` {.collection-method}

The `makeVisible` method makes attributes visible that are typically "hidden" on each model in the collection:

```
$users = $users->makeVisible(['address', 'phone_number']);
```

`makeHidden($attributes)` {.collection-method}

The `makeHidden` method hides attributes that are typically "visible" on each model in the collection:

```
$users = $users->makeHidden(['address', 'phone_number']);
```

`only($keys)` {.collection-method}

The `only` method returns all of the models that have the given primary keys:

```
$users = $users->only([1, 2, 3]);
```

`toQuery()` {.collection-method}

The `toQuery` method returns an Eloquent query builder instance containing a `whereIn` constraint on the collection model's primary keys:

```
use App\Models\User;

$users = User::where('status', 'VIP')->get();

$users->toQuery()->update([
    'status' => 'Administrator',
]);
```

`unique($key = null, $strict = false)` {.collection-method}

The `unique` method returns all of the unique models in the collection. Any models of the same type with the same primary key as another model in the collection are removed:

```
$users = $users->unique();
```

Custom Collections

If you would like to use a custom `Collection` object when interacting with a given model, you may define a `newCollection` method on your model:

```
<?php

namespace App\Models;

use App\Support\UserCollection;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Create a new Eloquent Collection instance.
     *
     * @param  array  $models
     * @return \Illuminate\Database\Eloquent\Collection
     */
    public function newCollection(array $models = [])
    {
        return new UserCollection($models);
    }
}
```

Once you have defined a `newCollection` method, you will receive an instance of your custom collection anytime Eloquent would normally return an `Illuminate\Database\Eloquent\Collection` instance. If you would like to use a custom collection for every model in your application, you should define the `newCollection` method on a base model class that is extended by all of your application's models.

Eloquent: Mutators & Casting

- Introduction
- Accessors & Mutators
    - Defining An Accessor
    - Defining A Mutator

- Attribute Casting
    - Array & JSON Casting
    - Date Casting
    - Enum Casting
    - Encrypted Casting
    - Query Time Casting
- Custom Casts
    - Value Object Casting
    - Array / JSON Serialization
    - Inbound Casting
    - Cast Parameters
    - Castables

Introduction

Accessors, mutators, and attribute casting allow you to transform Eloquent attribute values when you retrieve or set them on model instances. For example, you may want to use the Laravel encrypter to encrypt a value while it is stored in the database, and then automatically decrypt the attribute when you access it on an Eloquent model. Or, you may want to convert a JSON string that is stored in your database to an array when it is accessed via your Eloquent model.

Accessors & Mutators

Defining An Accessor

An accessor transforms an Eloquent attribute value when it is accessed. To define an accessor, create a `get{Attribute}Attribute` method on your model where `{Attribute}` is the "studly" cased name of the column you wish to access.

In this example, we'll define an accessor for the `first_name` attribute. The accessor will automatically be called by Eloquent when attempting to retrieve the value of the `first_name` attribute:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the user's first name.
     *
     * @param  string  $value
     * @return string
     */
    public function getFirstNameAttribute($value)
    {
        return ucfirst($value);
    }
}
```

As you can see, the original value of the column is passed to the accessor, allowing you to manipulate and return the value. To access the value of the accessor, you may simply access the `first_name` attribute on a model instance:

```
use App\Models\User;

$user = User::find(1);

$firstName = $user->first_name;
```

You are not limited to interacting with a single attribute within your accessor. You may also use accessors to return new, computed values from existing attributes:

```
/**
 * Get the user's full name.
 *
 * @return string
 */
public function getFullNameAttribute()
{
    return "{$this->first_name} {$this->last_name}";
}
```

{tip} If you would like these computed values to be added to the array / JSON representations of your model, you will need to append them.

Defining A Mutator

A mutator transforms an Eloquent attribute value when it is set. To define a mutator, define a `set{Attribute}Attribute` method on your model where `{Attribute}` is the "studly" cased name of the column you wish to access.

Let's define a mutator for the `first_name` attribute. This mutator will be automatically called when we attempt to set the value of the `first_name` attribute on the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Set the user's first name.
     *
     * @param  string  $value
     * @return void
     */
    public function setFirstNameAttribute($value)
    {
        $this->attributes['first_name'] = strtolower($value);
    }
}
```

The mutator will receive the value that is being set on the attribute, allowing you to manipulate the value and set the manipulated value on the Eloquent model's internal `$attributes` property. To use our mutator, we only need to set the `first_name` attribute on an Eloquent model:

```
use App\Models\User;

$user = User::find(1);

$user->first_name = 'Sally';
```

In this example, the `setFirstNameAttribute` function will be called with the value `Sally`. The mutator will then apply the `strtolower` function to the name and set its resulting value in the internal `$attributes` array.

Attribute Casting

Attribute casting provides functionality similar to accessors and mutators without requiring you to define any additional methods on your model. Instead, your model's `$casts` property provides a convenient method of converting attributes to common data types.

The `$casts` property should be an array where the key is the name of the attribute being cast and the value is the type you wish to cast the column to. The supported cast types are:

- `array` - `AsStringable::class` - `boolean` - `collection` - `date` - `datetime` - `immutable_date` - `immutable_datetime` - `decimal:<digits>` - `double` - `encrypted` - `encrypted:array` - `encrypted:collection` - `encrypted:object` - `float` - `integer` - `object` - `real` - `string` - `timestamp`

To demonstrate attribute casting, let's cast the `is_admin` attribute, which is stored in our database as an integer (`0` or `1`) to a boolean value:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'is_admin' => 'boolean',
    ];
}
```

After defining the cast, the `is_admin` attribute will always be cast to a boolean when you access it, even if the underlying value is stored in the database as an integer:

```
$user = App\Models\User::find(1);

if ($user->is_admin) {
    //
}
```

If you need to add a new, temporary cast at runtime, you may use the `mergeCasts` method. These cast definitions will be added to any of the casts already defined on the model:

```
$user->mergeCasts([
    'is_admin' => 'integer',
    'options' => 'object',
]);
```

{note} Attributes that are `null` will not be cast. In addition, you should never define a cast (or an attribute) that has the same name as a relationship.

Stringable Casting

You may use the `Illuminate\Database\Eloquent\Casts\AsStringable` cast class to cast a model attribute to a fluent `Illuminate\Support\Stringable` object:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Casts\AsStringable;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'directory' => AsStringable::class,
    ];
}
```

Array & JSON Casting

The `array` cast is particularly useful when working with columns that are stored as serialized JSON. For example, if your database has a `JSON` or `TEXT` field type that contains serialized JSON, adding the `array` cast to that attribute will automatically deserialize the attribute to a PHP array when you access it on your Eloquent model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'options' => 'array',
    ];
}
```

Once the cast is defined, you may access the `options` attribute and it will automatically be deserialized from JSON into a PHP array. When you set the value of the `options` attribute, the given array will automatically be serialized back into JSON for storage:

```php
use App\Models\User;

$user = User::find(1);

$options = $user->options;

$options['key'] = 'value';

$user->options = $options;

$user->save();
```

To update a single field of a JSON attribute with a more terse syntax, you may use the `->` operator when calling the `update` method:

```
$user = User::find(1);

$user->update(['options->key' => 'value']);
```

Array Object & Collection Casting

Although the standard `array` cast is sufficient for many applications, it does have some disadvantages. Since the `array` cast returns a primitive type, it is not possible to mutate an offset of the array directly. For example, the following code will trigger a PHP error:

```
$user = User::find(1);

$user->options['key'] = $value;
```

To solve this, Laravel offers an `AsArrayObject` cast that casts your JSON attribute to an ArrayObject class. This feature is implemented using Laravel's custom cast implementation, which allows Laravel to intelligently cache and transform the mutated object such that individual offsets may be modified without triggering a PHP error. To use the `AsArrayObject` cast, simply assign it to an attribute:

```
use Illuminate\Database\Eloquent\Casts\AsArrayObject;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'options' => AsArrayObject::class,
];
```

Similarly, Laravel offers an `AsCollection` cast that casts your JSON attribute to a Laravel Collection instance:

```
use Illuminate\Database\Eloquent\Casts\AsCollection;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'options' => AsCollection::class,
];
```

Date Casting

By default, Eloquent will cast the `created_at` and `updated_at` columns to instances of Carbon, which extends the PHP `DateTime` class and provides an assortment of helpful methods. You may cast additional date attributes by defining additional date casts within your model's `$casts` property array. Typically, dates should be cast using the `datetime` or `immutable_datetime` cast types.

When defining a `date` or `datetime` cast, you may also specify the date's format. This format will be used when the model is serialized to an array or JSON:

```
/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'created_at' => 'datetime:Y-m-d',
];
```

When a column is cast as a date, you may set the corresponding model attribute value to a UNIX timestamp, date string (`Y-m-d`), date-time string, or a `DateTime` / `Carbon` instance. The date's value will be correctly converted and stored in your database.

You may customize the default serialization format for all of your model's dates by defining a `serializeDate` method on your model. This method does not affect how your dates are formatted for storage in the database:

```
/**
 * Prepare a date for array / JSON serialization.
 *
 * @param  \DateTimeInterface  $date
 * @return string
 */
protected function serializeDate(DateTimeInterface $date)
{
    return $date->format('Y-m-d');
}
```

To specify the format that should be used when actually storing a model's dates within your database, you should define a `$dateFormat` property on your model:

```
/**
 * The storage format of the model's date columns.
 *
 * @var string
 */
protected $dateFormat = 'U';
```

Date Casting, Serialization, & Timezones

By default, the `date` and `datetime` casts will serialize dates to a UTC ISO-8601 date string (`1986-05-28T21:05:54.000000Z`), regardless of the timezone specified in your application's `timezone` configuration option. You are strongly encouraged to always use this serialization format, as well as to store your application's dates in the UTC timezone by not changing your application's `timezone` configuration option from its default `UTC` value. Consistently using the UTC timezone throughout your application will provide the maximum level of interoperability with other date manipulation libraries written in PHP and JavaScript.

If a custom format is applied to the `date` or `datetime` cast, such as `datetime:Y-m-d H:i:s`, the inner timezone of the Carbon instance will be used during date serialization. Typically, this will be the timezone specified in your application's `timezone` configuration option.

Enum Casting

{note} Enum casting is only available for PHP 8.1+.

Eloquent also allows you to cast your attribute values to PHP enums. To accomplish this, you may specify the attribute and enum you wish to cast in your model's `$casts` property array:

```
use App\Enums\ServerStatus;

/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'status' => ServerStatus::class,
];
```

Once you have defined the cast on your model, the specified attribute will be automatically cast to and from an enum when you interact with the attribute:

```
if ($server->status == ServerStatus::provisioned) {
    $server->status = ServerStatus::ready;

    $server->save();
}
```

Encrypted Casting

The `encrypted` cast will encrypt a model's attribute value using Laravel's built-in encryption features. In addition, the `encrypted:array`, `encrypted:collection`, `encrypted:object`, `AsEncryptedArrayObject`, and `AsEncryptedCollection` casts work like their unencrypted counterparts; however, as you might expect, the underlying value is encrypted when stored in your database.

As the final length of the encrypted text is not predictable and is longer than its plain text counterpart, make sure the associated database column is of `TEXT` type or larger. In addition, since the values are encrypted in the database, you will not be able to query or search encrypted attribute values.

Query Time Casting

Sometimes you may need to apply casts while executing a query, such as when selecting a raw value from a table. For example, consider the following query:

```
use App\Models\Post;
use App\Models\User;

$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
            ->whereColumn('user_id', 'users.id')
])->get();
```

The `last_posted_at` attribute on the results of this query will be a simple string. It would be wonderful if we could apply a `datetime` cast to this attribute when executing the query. Thankfully, we may accomplish this using the `withCasts` method:

```
$users = User::select([
    'users.*',
    'last_posted_at' => Post::selectRaw('MAX(created_at)')
            ->whereColumn('user_id', 'users.id')
])->withCasts([
    'last_posted_at' => 'datetime'
])->get();
```

Custom Casts

Laravel has a variety of built-in, helpful cast types; however, you may occasionally need to define your own cast types. You may accomplish this by defining a class that implements the `CastsAttributes` interface.

Classes that implement this interface must define a `get` and `set` method. The `get` method is responsible for transforming a raw value from the database into a cast value, while the `set` method should transform a cast value into a raw value that can be stored in the database. As an example, we will re-implement the built-in `json` cast type as a custom cast type:

```php
<?php

namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class Json implements CastsAttributes
{
    /**
     * Cast the given value.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  mixed  $value
     * @param  array  $attributes
     * @return array
     */
    public function get($model, $key, $value, $attributes)
    {
        return json_decode($value, true);
    }

    /**
     * Prepare the given value for storage.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  array  $value
     * @param  array  $attributes
     * @return string
     */
    public function set($model, $key, $value, $attributes)
    {
        return json_encode($value);
    }
}
```

Once you have defined a custom cast type, you may attach it to a model attribute using its class name:

```php
<?php

namespace App\Models;

use App\Casts\Json;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The attributes that should be cast.
     *
     * @var array
     */
    protected $casts = [
        'options' => Json::class,
    ];
}
```

Value Object Casting

You are not limited to casting values to primitive types. You may also cast values to objects. Defining custom casts that cast values to objects is very similar to casting to primitive types; however, the `set` method should return an array of key / value pairs that will be used to set raw, storable values on the model.

As an example, we will define a custom cast class that casts multiple model values into a single `Address` value object. We will assume the `Address` value has two public properties: `lineOne` and `lineTwo`:

```php
<?php

namespace App\Casts;

use App\Models\Address as AddressModel;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;
use InvalidArgumentException;

class Address implements CastsAttributes
{
    /**
```

```
     * Cast the given value.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  mixed  $value
     * @param  array  $attributes
     * @return \App\Models\Address
     */
    public function get($model, $key, $value, $attributes)
    {
        return new AddressModel(
            $attributes['address_line_one'],
            $attributes['address_line_two']
        );
    }

    /**
     * Prepare the given value for storage.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  \App\Models\Address  $value
     * @param  array  $attributes
     * @return array
     */
    public function set($model, $key, $value, $attributes)
    {
        if (! $value instanceof AddressModel) {
            throw new InvalidArgumentException('The given value is not
an Address instance.');
        }

        return [
            'address_line_one' => $value->lineOne,
            'address_line_two' => $value->lineTwo,
        ];
    }
}
```

When casting to value objects, any changes made to the value object will automatically be synced back to the model before the model is saved:

```
use App\Models\User;

$user = User::find(1);

$user->address->lineOne = 'Updated Address Value';

$user->save();
```

{tip} If you plan to serialize your Eloquent models containing value objects to JSON or arrays, you should implement the `Illuminate\Contracts\Support\Arrayable` and `JsonSerializable` interfaces on the value object.

Array / JSON Serialization

When an Eloquent model is converted to an array or JSON using the `toArray` and `toJson` methods, your custom cast value objects will typically be serialized as well as long as they implement the `Illuminate\Contracts\Support\Arrayable` and `JsonSerializable` interfaces. However, when using value objects provided by third-party libraries, you may not have the ability to add these interfaces to the object.

Therefore, you may specify that your custom cast class will be responsible for serializing the value object. To do so, your custom cast class should implement the `Illuminate\Contracts\Database\Eloquent\SerializesCastableAttributes` interface. This interface states that your class should contain a `serialize` method which should return the serialized form of your value object:

```
/**
 * Get the serialized representation of the value.
 *
 * @param  \Illuminate\Database\Eloquent\Model  $model
 * @param  string  $key
 * @param  mixed  $value
 * @param  array  $attributes
 * @return mixed
 */
public function serialize($model, string $key, $value, array
$attributes)
{
    return (string) $value;
}
```

Inbound Casting

Occasionally, you may need to write a custom cast that only transforms values that are being set on the model and does not perform any operations when attributes are being retrieved from the model. A classic example of an inbound only cast is a "hashing" cast. Inbound only custom casts should implement the `CastsInboundAttributes` interface, which only requires a `set` method to be defined.

```php
<?php

namespace App\Casts;

use Illuminate\Contracts\Database\Eloquent\CastsInboundAttributes;

class Hash implements CastsInboundAttributes
{
    /**
     * The hashing algorithm.
     *
     * @var string
     */
    protected $algorithm;

    /**
     * Create a new cast class instance.
     *
     * @param  string|null  $algorithm
     * @return void
     */
    public function __construct($algorithm = null)
    {
        $this->algorithm = $algorithm;
    }

    /**
     * Prepare the given value for storage.
     *
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @param  string  $key
     * @param  array  $value
     * @param  array  $attributes
     * @return string
     */
    public function set($model, $key, $value, $attributes)
    {
        return is_null($this->algorithm)
                    ? bcrypt($value)
                    : hash($this->algorithm, $value);
    }
}
```

Cast Parameters

When attaching a custom cast to a model, cast parameters may be specified by separating them from the class name using a `:` character and comma-delimiting multiple parameters. The parameters will be passed to the constructor of the cast class:

```
/**
 * The attributes that should be cast.
 *
 * @var array
 */
protected $casts = [
    'secret' => Hash::class.':sha256',
];
```

Castables

You may want to allow your application's value objects to define their own custom cast classes. Instead of attaching the custom cast class to your model, you may alternatively attach a value object class that implements the `Illuminate\Contracts\Database\Eloquent\Castable` interface:

```
use App\Models\Address;

protected $casts = [
    'address' => Address::class,
];
```

Objects that implement the `Castable` interface must define a `castUsing` method that returns the class name of the custom caster class that is responsible for casting to and from the `Castable` class:

```php
<?php

namespace App\Models;

use Illuminate\Contracts\Database\Eloquent\Castable;
use App\Casts\Address as AddressCast;

class Address implements Castable
{
    /**
     * Get the name of the caster class to use when casting from / to
this cast target.
     *
     * @param  array  $arguments
     * @return string
     */
    public static function castUsing(array $arguments)
    {
        return AddressCast::class;
    }
}
```

When using `Castable` classes, you may still provide arguments in the `$casts` definition. The arguments will be passed to the `castUsing` method:

```php
use App\Models\Address;

protected $casts = [
    'address' => Address::class.':argument',
];
```

Castables & Anonymous Cast Classes

By combining "castables" with PHP's anonymous classes, you may define a value object and its casting logic as a single castable object. To accomplish this, return an anonymous class from your value object's `castUsing` method. The anonymous class should implement the `CastsAttributes` interface:

```php
<?php

namespace App\Models;

use Illuminate\Contracts\Database\Eloquent\Castable;
use Illuminate\Contracts\Database\Eloquent\CastsAttributes;

class Address implements Castable
{
    // ...

    /**
     * Get the caster class to use when casting from / to this cast
target.
     *
     * @param  array  $arguments
     * @return object|string
     */
    public static function castUsing(array $arguments)
    {
        return new class implements CastsAttributes
        {
            public function get($model, $key, $value, $attributes)
            {
                return new Address(
                    $attributes['address_line_one'],
                    $attributes['address_line_two']
                );
            }

            public function set($model, $key, $value, $attributes)
            {
                return [
                    'address_line_one' => $value->lineOne,
                    'address_line_two' => $value->lineTwo,
                ];
            }
        };
    }
}
```

Eloquent: Relationships

- Introduction
- Defining Relationships
    - One To One
    - One To Many
    - One To Many (Inverse) / Belongs To
    - Has One Of Many
    - Has One Through
    - Has Many Through
- Many To Many Relationships
    - Retrieving Intermediate Table Columns
    - Filtering Queries Via Intermediate Table Columns
    - Defining Custom Intermediate Table Models
- Polymorphic Relationships
    - One To One
    - One To Many
    - One Of Many
    - Many To Many
    - Custom Polymorphic Types
- Dynamic Relationships
- Querying Relations
    - Relationship Methods Vs. Dynamic Properties
    - Querying Relationship Existence
    - Querying Relationship Absence
    - Querying Morph To Relationships
- Aggregating Related Models
    - Counting Related Models
    - Other Aggregate Functions
    - Counting Related Models On Morph To Relationships
- Eager Loading
    - Constraining Eager Loads
    - Lazy Eager Loading
    - Preventing Lazy Loading
- Inserting & Updating Related Models
    - The `save` Method
    - The `create` Method
    - Belongs To Relationships
    - Many To Many Relationships
- Touching Parent Timestamps

Introduction

Database tables are often related to one another. For example, a blog post may have many comments or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports a variety of common relationships:

- [One To One](#one-to-one) - [One To Many](#one-to-many) - [Many To Many](#many-to-many) - [Has One Through](#has-one-through) - [Has Many Through](#has-many-through) - [One To One (Polymorphic)](#one-to-one-polymorphic-relations) - [One To Many (Polymorphic)](#one-to-many-polymorphic-relations) - [Many To Many (Polymorphic)](#many-to-many-polymorphic-relations)

Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since relationships also serve as powerful query builders, defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional query constraints on this `posts` relationship:

```
$user->posts()->where('active', 1)->get();
```

But, before diving too deep into using relationships, let's learn how to define each type of relationship supported by Eloquent.

One To One

A one-to-one relationship is a very basic type of database relationship. For example, a `User` model might be associated with one `Phone` model. To define this relationship, we will place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result. The `hasOne` method is available to your model via the model's `Illuminate\Database\Eloquent\Model` base class:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone associated with the user.
     */
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}
```

The first argument passed to the `hasOne` method is the name of the related model class. Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship methods as if they were properties defined on the model:

```
$phone = User::find(1)->phone;
```

Eloquent determines the foreign key of the relationship based on the parent model name. In this case, the `Phone` model is automatically assumed to have a `user_id` foreign key. If you wish to override this convention, you may pass a second argument to the `hasOne` method:

```
return $this->hasOne(Phone::class, 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the primary key column of the parent. In other words, Eloquent will look for the value of the user's `id` column in the `user_id` column of the `Phone` record. If you would like the relationship to use a primary key value other than `id` or your model's `$primaryKey` property, you may pass a third argument to the `hasOne` method:

```
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relationship

So, we can access the `Phone` model from our `User` model. Next, let's define a relationship on the `Phone` model that will let us access the user that owns the phone. We can define the inverse of a `hasOne` relationship using the `belongsTo` method:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

When invoking the `user` method, Eloquent will attempt to find a `User` model that has an `id` which matches the `user_id` column on the `Phone` model.

Eloquent determines the foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. So, in this case, Eloquent assumes that the `Phone` model has a `user_id` column. However, if the foreign key on the `Phone` model is not `user_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```php
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo(User::class, 'foreign_key');
}
```

If the parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying the parent table's custom key:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
    return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

One To Many

A one-to-many relationship is used to define relationships where a single model is the parent to one or more child models. For example, a blog post may have an infinite number of comments. Like all other Eloquent relationships, one-to-many relationships are defined by defining a method on your Eloquent model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column for the `Comment` model. By convention, Eloquent will take the "snake case" name of the parent model and suffix it with `_id`. So, in this example, Eloquent will assume the foreign key column on the `Comment` model is `post_id`.

Once the relationship method has been defined, we can access the collection of related comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic relationship properties", we can access relationship methods as if they were

defined as properties on the model:

```
use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `comments` method and continuing to chain conditions onto the query:

```
$comment = Post::find(1)->comments()
                    ->where('title', 'foo')
                    ->first();
```

Like the `hasOne` method, you may also override the foreign and local keys by passing additional arguments to the `hasMany` method:

```
return $this->hasMany(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

One To Many (Inverse) / Belongs To

Now that we can access all of a post's comments, let's define a relationship to allow a comment to access its parent post. To define the inverse of a `hasMany` relationship, define a relationship method on the child model which calls the `belongsTo` method:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Once the relationship has been defined, we can retrieve a comment's parent post by accessing the `post` "dynamic relationship property":

```php
use App\Models\Comment;

$comment = Comment::find(1);

return $comment->post->title;
```

In the example above, Eloquent will attempt to find a `Post` model that has an `id` which matches the `post_id` column on the `Comment` model.

Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with a `_` followed by the name of the parent model's primary key column. So, in this example, Eloquent will assume the `Post` model's foreign key on the `comments` table is `post_id`.

However, if the foreign key for your relationship does not follow these conventions, you may pass a custom foreign key name as the second argument to the `belongsTo` method:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo(Post::class, 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to find the associated model using a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
    return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

Default Models

The `belongsTo`, `hasOne`, `hasOneThrough`, and `morphOne` relationships allow you to define a default model that will be returned if the given relationship is `null`. This pattern is often referred to as the Null Object pattern and can help remove conditional checks in your code. In the following example, the `user` relation will return an empty `App\Models\User` model if no user is attached to the `Post` model:

```
/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo(User::class)->withDefault();
}
```

To populate the default model with attributes, you may pass an array or closure to the `withDefault` method:

```
/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Guest Author',
    ]);
}

/**
 * Get the author of the post.
 */
public function user()
{
    return $this->belongsTo(User::class)->withDefault(function ($user,
$post) {
        $user->name = 'Guest Author';
    });
}
```

Querying Belongs To Relationships

When querying for the children of a "belongs to" relationship, you may manually build the `where` clause to retrieve the corresponding Eloquent models:

```
use App\Models\Post;

$posts = Post::where('user_id', $user->id)->get();
```

However, you may find it more convenient to use the `whereBelongsTo` method, which will automatically determine the proper relationship and foreign key for the given model:

```
$posts = Post::whereBelongsTo($user)->get();
```

By default, Laravel will determine the relationship associated with the given model based on the class name of the model; however, you may specify the relationship name manually by providing it as the second argument to the `whereBelongsTo` method:

```
$posts = Post::whereBelongsTo($user, 'author')->get();
```

Has One Of Many

Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a `User` model may be related to many `Order` models, but you want to define a convenient way to interact with the most recent order the user has placed. You may accomplish this using the `hasOne` relationship type combined with the `ofMany` methods:

```
/**
 * Get the user's most recent order.
 */
public function latestOrder()
{
    return $this->hasOne(Order::class)->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```
/**
 * Get the user's oldest order.
 */
public function oldestOrder()
{
    return $this->hasOne(Order::class)->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most expensive order. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (`min` or `max`) to apply when querying for the related model:

```php
/**
 * Get the user's largest order.
 */
public function largestOrder()
{
    return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

{note} Because PostgreSQL does not support executing the MAX function against UUID columns, it is not currently possible to use one-of-many relationships in combination with PostgreSQL UUID columns.

Advanced Has One Of Many Relationships

It is possible to construct more advanced "has one of many" relationships. For example, A `Product` model may have many associated `Price` models that are retained in the system even after new pricing is published. In addition, new pricing data for the product may be able to be published in advance to take effect at a future date via a `published_at` column.

So, in summary, we need to retrieve the latest published pricing where the published date is not in the future. In addition, if two prices have the same published date, we will prefer the price with the greatest ID. To accomplish this, we must pass an array to the `ofMany` method that contains the sortable columns which determine the latest price. In addition, a closure will be provided as the second argument to the `ofMany` method. This closure will be responsible for adding additional publish date constraints to the relationship query:

```php
/**
 * Get the current pricing for the product.
 */
public function currentPricing()
{
    return $this->hasOne(Price::class)->ofMany([
        'published_at' => 'max',
        'id' => 'max',
    ], function ($query) {
        $query->where('published_at', '<', now());
    });
}
```

Has One Through

The "has-one-through" relationship defines a one-to-one relationship with another model. However, this relationship indicates that the declaring model can be matched with one instance of another model by proceeding *through* a third model.

For example, in a vehicle repair shop application, each `Mechanic` model may be associated with one `Car` model, and each `Car` model may be associated with one `Owner` model. While the mechanic and the owner have no direct relationship within the database, the mechanic can access the owner *through* the `Car` model. Let's look at the tables necessary to define this relationship:

```
mechanics
    id - integer
    name - string

cars
    id - integer
    model - string
    mechanic_id - integer

owners
    id - integer
    name - string
    car_id - integer
```

Now that we have examined the table structure for the relationship, let's define the relationship on the `Mechanic` model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(Owner::class, Car::class);
    }
}
```

The first argument passed to the `hasOneThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasOneThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Foreign key on the cars table...
            'car_id', // Foreign key on the owners table...
            'id', // Local key on the mechanics table...
            'id' // Local key on the cars table...
        );
    }
}
```

Has Many Through

The "has-many-through" relationship provides a convenient way to access distant relations via an intermediate relation. For example, let's assume we are building a deployment platform like Laravel Vapor. A `Project` model might access many `Deployment` models through an intermediate `Environment` model. Using this example, you could easily gather all deployments for a given project. Let's look at the tables required to define this relationship:

```
projects
    id - integer
    name - string

environments
    id - integer
    project_id - integer
    name - string

deployments
    id - integer
    environment_id - integer
    commit_hash - string
```

Now that we have examined the table structure for the relationship, let's define the relationship on the `Project` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Project extends Model
{
    /**
     * Get all of the deployments for the project.
     */
    public function deployments()
    {
        return $this->hasManyThrough(Deployment::class,
Environment::class);
    }
}
```

The first argument passed to the `hasManyThrough` method is the name of the final model we wish to access, while the second argument is the name of the intermediate model.

Though the `Deployment` model's table does not contain a `project_id` column, the `hasManyThrough` relation provides access to a project's deployments via `$project->deployments`. To retrieve these models, Eloquent inspects the `project_id` column on the intermediate `Environment` model's table. After finding the relevant

environment IDs, they are used to query the `Deployment` model's table.

Key Conventions

Typical Eloquent foreign key conventions will be used when performing the relationship's queries. If you would like to customize the keys of the relationship, you may pass them as the third and fourth arguments to the `hasManyThrough` method. The third argument is the name of the foreign key on the intermediate model. The fourth argument is the name of the foreign key on the final model. The fifth argument is the local key, while the sixth argument is the local key of the intermediate model:

```
class Project extends Model
{
    public function deployments()
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // Foreign key on the environments table...
            'environment_id', // Foreign key on the deployments table...
            'id', // Local key on the projects table...
            'id' // Local key on the environments table...
        );
    }
}
```

Many To Many Relationships

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of a many-to-many relationship is a user that has many roles and those roles are also shared by other users in the application. For example, a user may be assigned the role of "Author" and "Editor"; however, those roles may also be assigned to other users as well. So, a user has many roles and a role has many users.

Table Structure

To define this relationship, three database tables are needed: `users`, `roles`, and `role_user`. The `role_user` table is derived from the alphabetical order of the related model names and contains `user_id` and `role_id` columns. This table is used as an intermediate table linking the users and roles.

Remember, since a role can belong to many users, we cannot simply place a `user_id` column on the `roles` table. This would mean that a role could only belong to a single user. In order to provide support for roles being assigned to multiple users, the `role_user` table is needed. We can summarize the relationship's table structure like so:

```
users
    id - integer
    name - string

roles
    id - integer
    name - string

role_user
    user_id - integer
    role_id - integer
```

Model Structure

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. The `belongsToMany` method is provided by the `Illuminate\Database\Eloquent\Model` base class that is used by all of your application's Eloquent models. For example, let's define a `roles` method on our `User` model. The first argument passed to this method is the name of the related model class:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

Once the relationship is defined, you may access the user's roles using the `roles` dynamic relationship property:

```php
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    //
}
```

Since all relationships also serve as query builders, you may add further constraints to the relationship query by calling the `roles` method and continuing to chain conditions onto the query:

```php
$roles = User::find(1)->roles()->orderBy('name')->get();
```

To determine the table name of the relationship's intermediate table, Eloquent will join the two related model names in alphabetical order. However, you are free to override this convention. You may do so by passing a second argument to the `belongsToMany` method:

```
return $this->belongsToMany(Role::class, 'role_user');
```

In addition to customizing the name of the intermediate table, you may also customize the column names of the keys on the table by passing additional arguments to the `belongsToMany` method. The third argument is the foreign key name of the model on which you are defining the relationship, while the fourth argument is the foreign key name of the model that you are joining to:

```
return $this->belongsToMany(Role::class, 'role_user', 'user_id',
'role_id');
```

Defining The Inverse Of The Relationship

To define the "inverse" of a many-to-many relationship, you should define a method on the related model which also returns the result of the `belongsToMany` method. To complete our user / role example, let's define the `users` method on the `Role` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

As you can see, the relationship is defined exactly the same as its `User` model counterpart with the exception of referencing the `App\Models\User` model. Since we're reusing the `belongsToMany` method, all of the usual table and key customization options are available when defining the "inverse" of many-to-many relationships.

Retrieving Intermediate Table Columns

As you have already learned, working with many-to-many relations requires the presence of an intermediate table. Eloquent provides some very helpful ways of interacting with this table. For example, let's assume our `User` model has many `Role` models that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Notice that each `Role` model we retrieve is automatically assigned a `pivot` attribute. This attribute contains a model representing the intermediate table.

By default, only the model keys will be present on the `pivot` model. If your intermediate table contains extra attributes, you must specify them when defining the relationship:

```
return $this->belongsToMany(Role::class)->withPivot('active',
'created_by');
```

If you would like your intermediate table to have `created_at` and `updated_at` timestamps that are automatically maintained by Eloquent, call the `withTimestamps` method when defining the relationship:

```
return $this->belongsToMany(Role::class)->withTimestamps();
```

{note} Intermediate tables that utilize Eloquent's automatically maintained timestamps are required to have both `created_at` and `updated_at` timestamp columns.

Customizing The `pivot` Attribute Name

As noted previously, attributes from the intermediate table may be accessed on models via the `pivot` attribute. However, you are free to customize the name of this attribute to better

reflect its purpose within your application.

For example, if your application contains users that may subscribe to podcasts, you likely have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table attribute to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```
return $this->belongsToMany(Podcast::class)
                ->as('subscription')
                ->withTimestamps();
```

Once the custom intermediate table attribute has been specified, you may access the intermediate table data using the customized name:

```
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

Filtering Queries Via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` relationship queries using the `wherePivot`, `wherePivotIn`, `wherePivotNotIn`, `wherePivotBetween`, `wherePivotNotBetween`, `wherePivotNull`, and `wherePivotNotNull` methods when defining the relationship:

```
return $this->belongsToMany(Role::class)
                ->wherePivot('approved', 1);

return $this->belongsToMany(Role::class)
                ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
                ->wherePivotNotIn('priority', [1, 2]);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotBetween('created_at', ['2020-01-01
00:00:00', '2020-12-31 00:00:00']);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNotBetween('created_at', ['2020-01-01
00:00:00', '2020-12-31 00:00:00']);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNull('expired_at');

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNotNull('expired_at');
```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your many-to-many relationship, you may call the `using` method when defining the relationship. Custom pivot models give you the opportunity to define additional methods on the pivot model.

Custom many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class while custom polymorphic many-to-many pivot models should extend the `Illuminate\Database\Eloquent\Relations\MorphPivot` class. For example, we may define a `Role` model which uses a custom `RoleUser` pivot model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return
$this->belongsToMany(User::class)->using(RoleUser::class);
    }
}
```

When defining the `RoleUser` model, you should extend the `Illuminate\Database\Eloquent\Relations\Pivot` class:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    //
}
```

{note} Pivot models may not use the `SoftDeletes` trait. If you need to soft delete pivot records consider converting your pivot model to an actual Eloquent model.

Custom Pivot Models And Incrementing IDs

If you have defined a many-to-many relationship that uses a custom pivot model, and that pivot model has an auto-incrementing primary key, you should ensure your custom pivot model class defines an `incrementing` property that is set to `true`.

```
/**
 * Indicates if the IDs are auto-incrementing.
 *
 * @var bool
 */
public $incrementing = true;
```

Polymorphic Relationships

A polymorphic relationship allows the child model to belong to more than one type of model using a single association. For example, imagine you are building an application that allows users to share blog posts and videos. In such an application, a `Comment` model might belong to both the `Post` and `Video` models.

One To One (Polymorphic)

Table Structure

A one-to-one polymorphic relation is similar to a typical one-to-one relation; however, the child model can belong to more than one type of model using a single association. For example, a blog `Post` and a `User` may share a polymorphic relation to an `Image` model. Using a one-to-one polymorphic relation allows you to have a single table of unique images that may be associated with posts and users. First, let's examine the table structure:

```
posts
    id - integer
    name - string

users
    id - integer
    name - string

images
    id - integer
    url - string
    imageable_id - integer
    imageable_type - string
```

Note the `imageable_id` and `imageable_type` columns on the `images` table. The `imageable_id` column will contain the ID value of the post or user, while the `imageable_type` column will contain the class name of the parent model. The `imageable_type` column is used by Eloquent to determine which "type" of parent model to return when accessing the `imageable` relation. In this case, the column would contain either `App\Models\Post` or `App\Models\User`.

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Image extends Model
{
    /**
     * Get the parent imageable model (user or post).
     */
    public function imageable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get the post's image.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}

class User extends Model
{
    /**
     * Get the user's image.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to retrieve the image for a post, we can access the `image` dynamic relationship property:

```
use App\Models\Post;

$post = Post::find(1);

$image = $post->image;
```

You may retrieve the parent of the polymorphic model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `imageable` method on the `Image` model. So, we will access that method as a dynamic relationship property:

```
use App\Models\Image;

$image = Image::find(1);

$imageable = $image->imageable;
```

The `imageable` relation on the `Image` model will return either a `Post` or `User` instance, depending on which type of model owns the image.

Key Conventions

If necessary, you may specify the name of the "id" and "type" columns utilized by your polymorphic child model. If you do so, ensure that you always pass the name of the relationship as the first argument to the `morphTo` method. Typically, this value should match the method name, so you may use PHP's `__FUNCTION__` constant:

```
/**
 * Get the model that the image belongs to.
 */
public function imageable()
{
    return $this->morphTo(__FUNCTION__, 'imageable_type',
'imageable_id');
}
```

One To Many (Polymorphic)

Table Structure

A one-to-many polymorphic relation is similar to a typical one-to-many relation; however, the child model can belong to more than one type of model using a single association. For example, imagine users of your application can "comment" on posts and videos. Using polymorphic relationships, you may use a single `comments` table to contain comments for both posts and videos. First, let's examine the table structure required to build this relationship:

```
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

Model Structure

Next, let's examine the model definitions needed to build this relationship:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the parent commentable model (post or video).
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your model's dynamic relationship properties. For example, to access all of the comments for a post, we can use the `comments` dynamic property:

```
use App\Models\Post;

$post = Post::find(1);

foreach ($post->comments as $comment) {
    //
}
```

You may also retrieve the parent of a polymorphic child model by accessing the name of the method that performs the call to `morphTo`. In this case, that is the `commentable` method on the `Comment` model. So, we will access that method as a dynamic relationship property in order to access the comment's parent model:

```
use App\Models\Comment;

$comment = Comment::find(1);

$commentable = $comment->commentable;
```

The `commentable` relation on the `Comment` model will return either a `Post` or `Video` instance, depending on which type of model is the comment's parent.

One Of Many (Polymorphic)

Sometimes a model may have many related models, yet you want to easily retrieve the "latest" or "oldest" related model of the relationship. For example, a `User` model may be related to many `Image` models, but you want to define a convenient way to interact with the most recent image the user has uploaded. You may accomplish this using the `morphOne` relationship type combined with the `ofMany` methods:

```php
/**
 * Get the user's most recent image.
 */
public function latestImage()
{
    return $this->morphOne(Image::class, 'imageable')->latestOfMany();
}
```

Likewise, you may define a method to retrieve the "oldest", or first, related model of a relationship:

```php
/**
 * Get the user's oldest image.
 */
public function oldestImage()
{
    return $this->morphOne(Image::class, 'imageable')->oldestOfMany();
}
```

By default, the `latestOfMany` and `oldestOfMany` methods will retrieve the latest or oldest related model based on the model's primary key, which must be sortable. However, sometimes you may wish to retrieve a single model from a larger relationship using a different sorting criteria.

For example, using the `ofMany` method, you may retrieve the user's most "liked" image. The `ofMany` method accepts the sortable column as its first argument and which aggregate function (`min` or `max`) to apply when querying for the related model:

```php
/**
 * Get the user's most popular image.
 */
public function bestImage()
{
    return $this->morphOne(Image::class, 'imageable')->ofMany('likes',
'max');
}
```

{tip} It is possible to construct more advanced "one of many" relationships. For more information, please consult the has one of many documentation.

Many To Many (Polymorphic)

Table Structure

Many-to-many polymorphic relations are slightly more complicated than "morph one" and "morph many" relationships. For example, a `Post` model and `Video` model could share a polymorphic relation to a `Tag` model. Using a many-to-many polymorphic relation in this situation would allow your application to have a single table of unique tags that may be associated with posts or videos. First, let's examine the table structure required to build this relationship:

```
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

{tip} Before diving into polymorphic many-to-many relationships, you may benefit from reading the documentation on typical many-to-many relationships.

Model Structure

Next, we're ready to define the relationships on the models. The `Post` and `Video` models will both contain a `tags` method that calls the `morphToMany` method provided by the base Eloquent model class.

The `morphToMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get all of the tags for the post.
     */
    public function tags()
    {
        return $this->morphToMany(Tag::class, 'taggable');
    }
}
```

Defining The Inverse Of The Relationship

Next, on the `Tag` model, you should define a method for each of its possible parent models. So, in this example, we will define a `posts` method and a `videos` method. Both of these methods should return the result of the `morphedByMany` method.

The `morphedByMany` method accepts the name of the related model as well as the "relationship name". Based on the name we assigned to our intermediate table name and the keys it contains, we will refer to the relationship as "taggable":

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
{
    /**
     * Get all of the posts that are assigned this tag.
     */
    public function posts()
    {
        return $this->morphedByMany(Post::class, 'taggable');
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

Retrieving The Relationship

Once your database table and models are defined, you may access the relationships via your models. For example, to access all of the tags for a post, you may use the `tags` dynamic relationship property:

```php
use App\Models\Post;

$post = Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

You may retrieve the parent of a polymorphic relation from the polymorphic child model by accessing the name of the method that performs the call to `morphedByMany`. In this case,

that is the `posts` or `videos` methods on the `Tag` model:

```
use App\Models\Tag;

$tag = Tag::find(1);

foreach ($tag->posts as $post) {
    //
}

foreach ($tag->videos as $video) {
    //
}
```

Custom Polymorphic Types

By default, Laravel will use the fully qualified class name to store the "type" of the related model. For instance, given the one-to-many relationship example above where a `Comment` model may belong to a `Post` or a `Video` model, the default `commentable_type` would be either `App\Models\Post` or `App\Models\Video`, respectively. However, you may wish to decouple these values from your application's internal structure.

For example, instead of using the model names as the "type", we may use simple strings such as `post` and `video`. By doing so, the polymorphic "type" column values in our database will remain valid even if the models are renamed:

```
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::enforceMorphMap([
    'post' => 'App\Models\Post',
    'video' => 'App\Models\Video',
]);
```

You may call the `enforceMorphMap` method in the `boot` method of your `App\Providers\AppServiceProvider` class or create a separate service provider if you wish.

You may determine the morph alias of a given model at runtime using the model's `getMorphClass` method. Conversely, you may determine the fully-qualified class name associated with a morph alias using the `Relation::getMorphedModel` method:

```
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```

{note} When adding a "morph map" to your existing application, every morphable `*_type` column value in your database that still contains a fully-qualified class will need to be converted to its "map" name.

Dynamic Relationships

You may use the `resolveRelationUsing` method to define relations between Eloquent models at runtime. While not typically recommended for normal application development, this may occasionally be useful when developing Laravel packages.

The `resolveRelationUsing` method accepts the desired relationship name as its first argument. The second argument passed to the method should be a closure that accepts the model instance and returns a valid Eloquent relationship definition. Typically, you should configure dynamic relationships within the boot method of a service provider:

```
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function ($orderModel) {
    return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```

{note} When defining dynamic relationships, always provide explicit key name arguments to the Eloquent relationship methods.

Querying Relations

Since all Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing a query to load the related models. In addition, all types of Eloquent relationships also serve as query builders, allowing you to continue to chain constraints onto the relationship query before finally executing the SQL query against your database.

For example, imagine a blog application in which a `User` model has many associated `Post` models:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

You may query the `posts` relationship and add additional constraints to the relationship like so:

```
use App\Models\User;

$user = User::find(1);

$user->posts()->where('active', 1)->get();
```

You are able to use any of the Laravel query builder's methods on the relationship, so be sure to explore the query builder documentation to learn about all of the methods that are

available to you.

Chaining `orWhere` Clauses After Relationships

As demonstrated in the example above, you are free to add additional constraints to relationships when querying them. However, use caution when chaining `orWhere` clauses onto a relationship, as the `orWhere` clauses will be logically grouped at the same level as the relationship constraint:

```
$user->posts()
        ->where('active', 1)
        ->orWhere('votes', '>=', 100)
        ->get();
```

The example above will generate the following SQL. As you can see, the `or` clause instructs the query to return *any* user with greater than 100 votes. The query is no longer constrained to a specific user:

```
select *
from posts
where user_id = ? and active = 1 or votes >= 100
```

In most situations, you should use logical groups to group the conditional checks between parentheses:

```
use Illuminate\Database\Eloquent\Builder;

$user->posts()
        ->where(function (Builder $query) {
            return $query->where('active', 1)
                         ->orWhere('votes', '>=', 100);
        })
        ->get();
```

The example above will produce the following SQL. Note that the logical grouping has properly grouped the constraints and the query remains constrained to a specific user:

```
select *
from posts
where user_id = ? and (active = 1 or votes >= 100)
```

Relationship Methods Vs. Dynamic Properties

If you do not need to add additional constraints to an Eloquent relationship query, you may access the relationship as if it were a property. For example, continuing to use our `User` and `Post` example models, we may access all of a user's posts like so:

```
use App\Models\User;

$user = User::find(1);

foreach ($user->posts as $post) {
    //
}
```

Dynamic relationship properties perform "lazy loading", meaning they will only load their relationship data when you actually access them. Because of this, developers often use eager loading to pre-load relationships they know will be accessed after loading the model. Eager loading provides a significant reduction in SQL queries that must be executed to load a model's relations.

Querying Relationship Existence

When retrieving model records, you may wish to limit your results based on the existence of a relationship. For example, imagine you want to retrieve all blog posts that have at least one comment. To do so, you may pass the name of the relationship to the `has` and `orHas` methods:

```
use App\Models\Post;

// Retrieve all posts that have at least one comment...
$posts = Post::has('comments')->get();
```

You may also specify an operator and count value to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment that has at least one image:

```
// Retrieve posts that have at least one comment with images...
$posts = Post::has('comments.images')->get();
```

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to define additional query constraints on your `has` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like
code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();

// Retrieve posts with at least ten comments containing words like
code%...
$posts = Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

{note} Eloquent does not currently support querying for relationship existence across databases. The relationships must exist within the same database.

Inline Relationship Existence Queries

If you would like to query for a relationship's existence with a single, simple where condition attached to the relationship query, you may find it more convenient to use the `whereRelation` and `whereMorphRelation` methods. For example, we may query for all posts that have unapproved comments:

```
use App\Models\Post;

$posts = Post::whereRelation('comments', 'is_approved', false)->get();
```

Of course, like calls to the query builder's `where` method, you may also specify an operator:

```
$posts = Post::whereRelation(
    'comments', 'created_at', '>=', now()->subHour()
)->get();
```

Querying Relationship Absence

When retrieving model records, you may wish to limit your results based on the absence of a relationship. For example, imagine you want to retrieve all blog posts that **don't** have any comments. To do so, you may pass the name of the relationship to the `doesntHave` and `orDoesntHave` methods:

```
use App\Models\Post;

$posts = Post::doesntHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` and `orWhereDoesntHave` methods to add additional query constraints to your `doesntHave` queries, such as inspecting the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
    $query->where('content', 'like', 'code%');
})->get();
```

You may use "dot" notation to execute a query against a nested relationship. For example, the following query will retrieve all posts that do not have comments; however, posts that have comments from authors that are not banned will be included in the results:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments.author', function (Builder
$query) {
    $query->where('banned', 0);
})->get();
```

Querying Morph To Relationships

To query the existence of "morph to" relationships, you may use the `whereHasMorph` and `whereDoesntHaveMorph` methods. These methods accept the name of the relationship as their first argument. Next, the methods accept the names of the related models that you wish to include in the query. Finally, you may provide a closure which customizes the relationship query:

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;

// Retrieve comments associated to posts or videos with a title like
code%...
$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();

// Retrieve comments associated to posts with a title not like code%...
$comments = Comment::whereDoesntHaveMorph(
    'commentable',
    Post::class,
    function (Builder $query) {
        $query->where('title', 'like', 'code%');
    }
)->get();
```

You may occasionally need to add query constraints based on the "type" of the related

polymorphic model. The closure passed to the `whereHasMorph` method may receive a `$type` value as its second argument. This argument allows you to inspect the "type" of the query that is being built:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph(
    'commentable',
    [Post::class, Video::class],
    function (Builder $query, $type) {
        $column = $type === Post::class ? 'content' : 'title';

        $query->where($column, 'like', 'code%');
    }
)->get();
```

Querying All Related Models

Instead of passing an array of possible polymorphic models, you may provide `*` as a wildcard value. This will instruct Laravel to retrieve all of the possible polymorphic types from the database. Laravel will execute an additional query in order to perform this operation:

```
use Illuminate\Database\Eloquent\Builder;

$comments = Comment::whereHasMorph('commentable', '*', function (Builder
$query) {
    $query->where('title', 'like', 'foo%');
})->get();
```

Aggregating Related Models

Counting Related Models

Sometimes you may want to count the number of related models for a given relationship without actually loading the models. To accomplish this, you may use the `withCount` method. The `withCount` method will place a `{relation}_count` attribute on the resulting models:

```
use App\Models\Post;

$posts = Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

By passing an array to the `withCount` method, you may add the "counts" for multiple relations as well as add additional constraints to the queries:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount(['votes', 'comments' => function (Builder
$query) {
    $query->where('content', 'like', 'code%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

You may also alias the relationship count result, allowing multiple counts on the same relationship:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::withCount([
    'comments',
    'comments as pending_comments_count' => function (Builder $query) {
        $query->where('approved', false);
    },
])->get();

echo $posts[0]->comments_count;
echo $posts[0]->pending_comments_count;
```

Deferred Count Loading

Using the `loadCount` method, you may load a relationship count after the parent model has already been retrieved:

```
$book = Book::first();

$book->loadCount('genres');
```

If you need to set additional query constraints on the count query, you may pass an array keyed by the relationships you wish to count. The array values should be closures which receive the query builder instance:

```
$book->loadCount(['reviews' => function ($query) {
    $query->where('rating', 5);
}])
```

Relationship Counting & Custom Select Statements

If you're combining `withCount` with a `select` statement, ensure that you call `withCount` after the `select` method:

```
$posts = Post::select(['title', 'body'])
                ->withCount('comments')
                ->get();
```

Other Aggregate Functions

In addition to the `withCount` method, Eloquent provides `withMin`, `withMax`, `withAvg`, `withSum`, and `withExists` methods. These methods will place a `{relation}_{function}_{column}` attribute on your resulting models:

```
use App\Models\Post;

$posts = Post::withSum('comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->comments_sum_votes;
}
```

If you wish to access the result of the aggregate function using another name, you may specify your own alias:

```
$posts = Post::withSum('comments as total_comments', 'votes')->get();

foreach ($posts as $post) {
    echo $post->total_comments;
}
```

Like the `loadCount` method, deferred versions of these methods are also available. These additional aggregate operations may be performed on Eloquent models that have already been retrieved:

```
$post = Post::first();

$post->loadSum('comments', 'votes');
```

If you're combining these aggregate methods with a `select` statement, ensure that you call

the aggregate methods after the `select` method:

```
$posts = Post::select(['title', 'body'])
                ->withExists('comments')
                ->get();
```

Counting Related Models On Morph To Relationships

If you would like to eager load a "morph to" relationship, as well as related model counts for the various entities that may be returned by that relationship, you may utilize the `with` method in combination with the `morphTo` relationship's `morphWithCount` method.

In this example, let's assume that `Photo` and `Post` models may create `ActivityFeed` models. We will assume the `ActivityFeed` model defines a "morph to" relationship named `parentable` that allows us to retrieve the parent `Photo` or `Post` model for a given `ActivityFeed` instance. Additionally, let's assume that `Photo` models "have many" `Tag` models and `Post` models "have many" `Comment` models.

Now, let's imagine we want to retrieve `ActivityFeed` instances and eager load the `parentable` parent models for each `ActivityFeed` instance. In addition, we want to retrieve the number of tags that are associated with each parent photo and the number of comments that are associated with each parent post:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
    'parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWithCount([
            Photo::class => ['tags'],
            Post::class => ['comments'],
        ]);
    }])->get();
```

Deferred Count Loading

Let's assume we have already retrieved a set of `ActivityFeed` models and now we would like to load the nested relationship counts for the various `parentable` models associated with the activity feeds. You may use the `loadMorphCount` method to accomplish this:

```
$activities = ActivityFeed::with('parentable')->get();

$activities->loadMorphCount('parentable', [
    Photo::class => ['tags'],
    Post::class => ['comments'],
]);
```

Eager Loading

When accessing Eloquent relationships as properties, the related models are "lazy loaded". This means the relationship data is not actually loaded until you first access the property. However, Eloquent can "eager load" relationships at the time you query the parent model. Eager loading alleviates the "N + 1" query problem. To illustrate the N + 1 query problem, consider a `Book` model that "belongs to" to an `Author` model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

Now, let's retrieve all books and their authors:

```
use App\Models\Book;

$books = Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute one query to retrieve all of the books within the database table, then another query for each book in order to retrieve the book's author. So, if we have 25 books, the code above would run 26 queries: one for the original book, and 25 additional queries to retrieve the author of each book.

Thankfully, we can use eager loading to reduce this operation to just two queries. When building a query, you may specify which relationships should be eager loaded using the `with` method:

```
$books = Book::with('author')->get();

foreach ($books as $book) {
    echo $book->author->name;
}
```

For this operation, only two queries will be executed - one query to retrieve all of the books and one query to retrieve all of the authors for all of the books:

```
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Eager Loading Multiple Relationships

Sometimes you may need to eager load several different relationships. To do so, just pass an array of relationships to the `with` method:

```
$books = Book::with(['author', 'publisher'])->get();
```

Nested Eager Loading

To eager load a relationship's relationships, you may use "dot" syntax. For example, let's eager load all of the book's authors and all of the author's personal contacts:

```
$books = Book::with('author.contacts')->get();
```

Nested Eager Loading `morphTo` Relationships

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `with` method in combination with the `morphTo` relationship's `morphWith` method. To help

illustrate this method, let's consider the following model:

```
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model instances and eager load all `parentable` models and their respective nested relationships:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::query()
    ->with(['parentable' => function (MorphTo $morphTo) {
        $morphTo->morphWith([
            Event::class => ['calendar'],
            Photo::class => ['tags'],
            Post::class => ['author'],
        ]);
    }])->get();
```

Eager Loading Specific Columns

You may not always need every column from the relationships you are retrieving. For this reason, Eloquent allows you to specify which columns of the relationship you would like to retrieve:

```
$books = Book::with('author:id,name,book_id')->get();
```

{note} When using this feature, you should always include the `id` column and any relevant foreign key columns in the list of columns you wish to retrieve.

Eager Loading By Default

Sometimes you might want to always load some relationships when retrieving a model. To accomplish this, you may define a `$with` property on the model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * The relationships that should always be loaded.
     *
     * @var array
     */
    protected $with = ['author'];

    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo(Author::class);
    }

    /**
     * Get the genre of the book.
     */
    public function genre()
    {
        return $this->belongsTo(Genre::class);
    }
}
```

If you would like to remove an item from the `$with` property for a single query, you may use the `without` method:

```php
$books = Book::without('author')->get();
```

If you would like to override all items within the `$with` property for a single query, you may use the `withOnly` method:

```
$books = Book::withOnly('genre')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship but also specify additional query conditions for the eager loading query. You can accomplish this by passing an array of relationships to the `with` method where the array key is a relationship name and the array value is a closure that adds additional constraints to the eager loading query:

```
use App\Models\User;

$users = User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `code`. You may call other query builder methods to further customize the eager loading operation:

```
$users = User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

{note} The `limit` and `take` query builder methods may not be used when constraining eager loads.

Constraining Eager Loading Of `morphTo` Relationships

If you are eager loading a `morphTo` relationship, Eloquent will run multiple queries to fetch each type of related model. You may add additional constraints to each of these queries using the `MorphTo` relation's `constrain` method:

```
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo)
{
    $morphTo->constrain([
        Post::class => function (Builder $query) {
            $query->whereNull('hidden_at');
        },
        Video::class => function (Builder $query) {
            $query->where('type', 'educational');
        },
    ]);
}])->get();
```

In this example, Eloquent will only eager load posts that have not been hidden and videos have a `type` value of "educational".

Lazy Eager Loading

Sometimes you may need to eager load a relationship after the parent model has already been retrieved. For example, this may be useful if you need to dynamically decide whether to load related models:

```
use App\Models\Book;

$books = Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
}
```

If you need to set additional query constraints on the eager loading query, you may pass an array keyed by the relationships you wish to load. The array values should be closure instances which receive the query instance:

```
$author->load(['books' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

To load a relationship only when it has not already been loaded, use the `loadMissing` method:

```
$book->loadMissing('author');
```

Nested Lazy Eager Loading & `morphTo`

If you would like to eager load a `morphTo` relationship, as well as nested relationships on the various entities that may be returned by that relationship, you may use the `loadMorph` method.

This method accepts the name of the `morphTo` relationship as its first argument, and an array of model / relationship pairs as its second argument. To help illustrate this method, let's consider the following model:

```
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

In this example, let's assume `Event`, `Photo`, and `Post` models may create `ActivityFeed` models. Additionally, let's assume that `Event` models belong to a `Calendar` model, `Photo` models are associated with `Tag` models, and `Post` models belong to an `Author` model.

Using these model definitions and relationships, we may retrieve `ActivityFeed` model

instances and eager load all `parentable` models and their respective nested relationships:

```
$activities = ActivityFeed::with('parentable')
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
```

Preventing Lazy Loading

As previously discussed, eager loading relationships can often provide significant performance benefits to your application. Therefore, if you would like, you may instruct Laravel to always prevent the lazy loading of relationships. To accomplish this, you may invoke the `preventLazyLoading` method offered by the base Eloquent model class. Typically, you should call this method within the `boot` method of your application's `AppServiceProvider` class.

The `preventLazyLoading` method accepts an optional boolean argument that indicates if lazy loading should be prevented. For example, you may wish to only disable lazy loading in non-production environments so that your production environment will continue to function normally even if a lazy loaded relationship is accidentally present in production code:

```
use Illuminate\Database\Eloquent\Model;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Model::preventLazyLoading(! $this->app->isProduction());
}
```

After preventing lazy loading, Eloquent will throw a `Illuminate\Database\LazyLoadingViolationException` exception when your application attempts to lazy load any Eloquent relationship.

You may customize the behavior of lazy loading violations using the `handleLazyLoadingViolationsUsing` method. For example, using this method, you may instruct lazy loading violations to only be logged instead of interrupting the application's execution with exceptions:

```
Model::handleLazyLoadingViolationUsing(function ($model, $relation) {
    $class = get_class($model);

    info("Attempted to lazy load [{$relation}] on model [{$class}].");
});
```

Inserting & Updating Related Models

The `save` Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to add a new comment to a post. Instead of manually setting the `post_id` attribute on the `Comment` model you may insert the comment using the relationship's `save` method:

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

$post->comments()->save($comment);
```

Note that we did not access the `comments` relationship as a dynamic property. Instead, we called the `comments` method to obtain an instance of the relationship. The `save` method will automatically add the appropriate `post_id` value to the new `Comment` model.

If you need to save multiple related models, you may use the `saveMany` method:

```
$post = Post::find(1);

$post->comments()->saveMany([
    new Comment(['message' => 'A new comment.']),
    new Comment(['message' => 'Another new comment.']),
]);
```

The `save` and `saveMany` methods will persist the given model instances, but will not add the newly persisted models to any in-memory relationships that are already loaded onto the parent model. If you plan on accessing the relationship after using the `save` or `saveMany` methods, you may wish to use the `refresh` method to reload the model and its

relationships:

```
$post->comments()->save($comment);

$post->refresh();

// All comments, including the newly saved comment...
$post->comments;
```

Recursively Saving Models & Relationships

If you would like to `save` your model and all of its associated relationships, you may use the `push` method. In this example, the `Post` model will be saved as well as its comments and the comment's authors:

```
$post = Post::find(1);

$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

The `create` Method

In addition to the `save` and `saveMany` methods, you may also use the `create` method, which accepts an array of attributes, creates a model, and inserts it into the database. The difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`. The newly created model will be returned by the `create` method:

```
use App\Models\Post;

$post = Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

You may use the `createMany` method to create multiple related models:

```
$post = Post::find(1);

$post->comments()->createMany([
    ['message' => 'A new comment.'],
    ['message' => 'Another new comment.'],
]);
```

You may also use the `findOrNew`, `firstOrNew`, `firstOrCreate`, and `updateOrCreate` methods to create and update models on relationships.

{tip} Before using the `create` method, be sure to review the mass assignment documentation.

Belongs To Relationships

If you would like to assign a child model to a new parent model, you may use the `associate` method. In this example, the `User` model defines a `belongsTo` relationship to the `Account` model. This `associate` method will set the foreign key on the child model:

```
use App\Models\Account;

$account = Account::find(10);

$user->account()->associate($account);

$user->save();
```

To remove a parent model from a child model, you may use the `dissociate` method. This method will set the relationship's foreign key to `null`:

```
$user->account()->dissociate();

$user->save();
```

Many To Many Relationships

Attaching / Detaching

Eloquent also provides methods to make working with many-to-many relationships more convenient. For example, let's imagine a user can have many roles and a role can have many users. You may use the `attach` method to attach a role to a user by inserting a record in the relationship's intermediate table:

```
use App\Models\User;

$user = User::find(1);

$user->roles()->attach($roleId);
```

When attaching a relationship to a model, you may also pass an array of additional data to be inserted into the intermediate table:

```
$user->roles()->attach($roleId, ['expires' => $expires]);
```

Sometimes it may be necessary to remove a role from a user. To remove a many-to-many relationship record, use the `detach` method. The `detach` method will delete the appropriate record out of the intermediate table; however, both models will remain in the database:

```
// Detach a single role from the user...
$user->roles()->detach($roleId);

// Detach all roles from the user...
$user->roles()->detach();
```

For convenience, `attach` and `detach` also accept arrays of IDs as input:

```
$user = User::find(1);

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires],
]);
```

Syncing Associations

You may also use the `sync` method to construct many-to-many associations. The `sync` method accepts an array of IDs to place on the intermediate table. Any IDs that are not in the given array will be removed from the intermediate table. So, after this operation is complete, only the IDs in the given array will exist in the intermediate table:

```
$user->roles()->sync([1, 2, 3]);
```

You may also pass additional intermediate table values with the IDs:

```
$user->roles()->sync([1 => ['expires' => true], 2, 3]);
```

If you would like to insert the same intermediate table values with each of the synced model IDs, you may use the `syncWithPivotValues` method:

```
$user->roles()->syncWithPivotValues([1, 2, 3], ['active' => true]);
```

If you do not want to detach existing IDs that are missing from the given array, you may use the `syncWithoutDetaching` method:

```
$user->roles()->syncWithoutDetaching([1, 2, 3]);
```

Toggling Associations

The many-to-many relationship also provides a `toggle` method which "toggles" the attachment status of the given related model IDs. If the given ID is currently attached, it will be detached. Likewise, if it is currently detached, it will be attached:

```
$user->roles()->toggle([1, 2, 3]);
```

Updating A Record On The Intermediate Table

If you need to update an existing row in your relationship's intermediate table, you may use the `updateExistingPivot` method. This method accepts the intermediate record foreign key and an array of attributes to update:

```
$user = User::find(1);

$user->roles()->updateExistingPivot($roleId, [
    'active' => false,
]);
```

Touching Parent Timestamps

When a model defines a `belongsTo` or `belongsToMany` relationship to another model, such as a `Comment` which belongs to a `Post`, it is sometimes helpful to update the parent's timestamp when the child model is updated.

For example, when a `Comment` model is updated, you may want to automatically "touch" the `updated_at` timestamp of the owning `Post` so that it is set to the current date and time. To accomplish this, you may add a `touches` property to your child model containing the names of the relationships that should have their `updated_at` timestamps updated when the child model is updated:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * All of the relationships to be touched.
     *
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

{note} Parent model timestamps will only be updated if the child model is updated using Eloquent's `save` method.

Eloquent: Getting Started

- Introduction
- Generating Model Classes
- Eloquent Model Conventions
    - Table Names
    - Primary Keys
    - Timestamps
    - Database Connections
    - Default Attribute Values
- Retrieving Models
    - Collections
    - Chunking Results
    - Streaming Results Lazily
    - Cursors
    - Advanced Subqueries
- Retrieving Single Models / Aggregates
    - Retrieving Or Creating Models
    - Retrieving Aggregates
- Inserting & Updating Models
    - Inserts
    - Updates
    - Mass Assignment
    - Upserts
- Deleting Models
    - Soft Deleting
    - Querying Soft Deleted Models
- Pruning Models
- Replicating Models
- Query Scopes
    - Global Scopes
    - Local Scopes
- Comparing Models
- Events
    - Using Closures
    - Observers
    - Muting Events

Introduction

Laravel includes Eloquent, an object-relational mapper (ORM) that makes it enjoyable to interact with your database. When using Eloquent, each database table has a corresponding "Model" that is used to interact with that table. In addition to retrieving records from the database table, Eloquent models allow you to insert, update, and delete records from the table as well.

{tip} Before getting started, be sure to configure a database connection in your application's `config/database.php` configuration file. For more information on configuring your database, check out the database configuration documentation.

Generating Model Classes

To get started, let's create an Eloquent model. Models typically live in the `app\Models` directory and extend the `Illuminate\Database\Eloquent\Model` class. You may use the `make:model` Artisan command to generate a new model:

```
php artisan make:model Flight
```

If you would like to generate a database migration when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model Flight --migration
```

You may generate various other types of classes when generating a model, such as factories, seeders, policies, controllers, and form requests. In addition, these options may be combined to create multiple classes at once:

```
# Generate a model and a FlightFactory class...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Generate a model and a FlightSeeder class...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Generate a model and a FlightController class...
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generate a model, FlightController resource class, and form request
classes...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Generate a model and a FlightPolicy class...
php artisan make:model Flight --policy

# Generate a model and a migration, factory, seeder, and controller...
php artisan make:model Flight -mfsc

# Shortcut to generate a model, migration, factory, seeder, policy,
controller, and form requests...
php artisan make:model Flight --all

# Generate a pivot model...
php artisan make:model Member --pivot
```

Eloquent Model Conventions

Models generated by the `make:model` command will be placed in the `app/Models` directory. Let's examine a basic model class and discuss some of Eloquent's key conventions:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Table Names

After glancing at the example above, you may have noticed that we did not tell Eloquent which database table corresponds to our `Flight` model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the `Flight` model stores records in the `flights` table, while an `AirTrafficController` model would store records in an `air_traffic_controllers` table.

If your model's corresponding database table does not fit this convention, you may manually specify the model's table name by defining a `table` property on the model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Primary Keys

Eloquent will also assume that each model's corresponding database table has a primary key column named `id`. If necessary, you may define a protected `$primaryKey` property on your model to specify a different column that serves as your model's primary key:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that Eloquent will automatically cast the primary key to an integer. If you wish to use

a non-incrementing or a non-numeric primary key you must define a public `$incrementing` property on your model that is set to `false`:

```
<?php

class Flight extends Model
{
    /**
     * Indicates if the model's ID is auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

If your model's primary key is not an integer, you should define a protected `$keyType` property on your model. This property should have a value of `string`:

```
<?php

class Flight extends Model
{
    /**
     * The data type of the auto-incrementing ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

"Composite" Primary Keys

Eloquent requires each model to have at least one uniquely identifying "ID" that can serve as its primary key. "Composite" primary keys are not supported by Eloquent models. However, you are free to add additional multi-column, unique indexes to your database tables in addition to the table's uniquely identifying primary key.

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your model's corresponding database table. Eloquent will automatically set these column's values when models are created or updated. If you do not want these columns to be automatically managed by Eloquent, you should define a `$timestamps` property on your model with a value of `false`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

If you need to customize the format of your model's timestamps, set the `$dateFormat` property on your model. This property determines how date attributes are stored in the database as well as their format when the model is serialized to an array or JSON:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```

If you need to customize the names of the columns used to store the timestamps, you may define `CREATED_AT` and `UPDATED_AT` constants on your model:

```php
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

Database Connections

By default, all Eloquent models will use the default database connection that is configured for your application. If you would like to specify a different connection that should be used when interacting with a particular model, you should define a `$connection` property on the model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The database connection that should be used by the model.
     *
     * @var string
     */
    protected $connection = 'sqlite';
}
```

Default Attribute Values

By default, a newly instantiated model instance will not contain any attribute values. If you would like to define the default values for some of your model's attributes, you may define an `$attributes` property on your model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The model's default values for attributes.
     *
     * @var array
     */
    protected $attributes = [
        'delayed' => false,
    ];
}
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. You can think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. The model's `all` method will retrieve all of the records from the model's associated database table:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Building Queries

The Eloquent `all` method will return all of the results in the model's table. However, since each Eloquent model serves as a query builder, you may add additional constraints to queries and then invoke the `get` method to retrieve the results:

```
$flights = Flight::where('active', 1)
               ->orderBy('name')
               ->take(10)
               ->get();
```

{tip} Since Eloquent models are query builders, you should review all of the methods provided by Laravel's query builder. You may use any of these methods when writing your Eloquent queries.

Refreshing Models

If you already have an instance of an Eloquent model that was retrieved from the database, you can "refresh" the model using the `fresh` and `refresh` methods. The `fresh` method will re-retrieve the model from the database. The existing model instance will not be affected:

```
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

The `refresh` method will re-hydrate the existing model using fresh data from the database. In addition, all of its loaded relationships will be refreshed as well:

```
$flight = Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Collections

As we have seen, Eloquent methods like `all` and `get` retrieve multiple records from the database. However, these methods don't return a plain PHP array. Instead, an instance of `Illuminate\Database\Eloquent\Collection` is returned.

The Eloquent `Collection` class extends Laravel's base `Illuminate\Support\Collection` class, which provides a variety of helpful methods for interacting with data collections. For example, the `reject` method may be used to remove models from a collection based on the results of an invoked closure:

```
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

In addition to the methods provided by Laravel's base collection class, the Eloquent collection class provides a few extra methods that are specifically intended for interacting with collections of Eloquent models.

Since all of Laravel's collections implement PHP's iterable interfaces, you may loop over collections as if they were an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Chunking Results

Your application may run out of memory if you attempt to load tens of thousands of Eloquent records via the `all` or `get` methods. Instead of using these methods, the `chunk` method may be used to process large numbers of models more efficiently.

The `chunk` method will retrieve a subset of Eloquent models, passing them to a closure for processing. Since only the current chunk of Eloquent models is retrieved at a time, the `chunk` method will provide significantly reduced memory usage when working with a large number of models:

```
use App\Models\Flight;

Flight::chunk(200, function ($flights) {
    foreach ($flights as $flight) {
        //
    }
});
```

The first argument passed to the `chunk` method is the number of records you wish to receive per "chunk". The closure passed as the second argument will be invoked for each chunk that is retrieved from the database. A database query will be executed to retrieve each chunk of records passed to the closure.

If you are filtering the results of the `chunk` method based on a column that you will also be updating while iterating over the results, you should use the `chunkById` method. Using the `chunk` method in these scenarios could lead to unexpected and inconsistent results. Internally, the `chunkById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
    ->chunkById(200, function ($flights) {
        $flights->each->update(['departed' => false]);
    }, $column = 'id');
```

Streaming Results Lazily

The `lazy` method works similarly to the chunk method in the sense that, behind the scenes, it executes the query in chunks. However, instead of passing each chunk directly into a callback as is, the `lazy` method returns a flattened `LazyCollection` of Eloquent models, which lets you interact with the results as a single stream:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    //
}
```

If you are filtering the results of the `lazy` method based on a column that you will also be updating while iterating over the results, you should use the `lazyById` method. Internally, the `lazyById` method will always retrieve models with an `id` column greater than the last model in the previous chunk:

```
Flight::where('departed', true)
    ->lazyById(200, $column = 'id')
    ->each->update(['departed' => false]);
```

You may filter the results based on the descending order of the `id` using the `lazyByIdDesc` method.

Cursors

Similar to the `lazy` method, the `cursor` method may be used to significantly reduce your application's memory consumption when iterating through tens of thousands of Eloquent model records.

The `cursor` method will only execute a single database query; however, the individual Eloquent models will not be hydrated until they are actually iterated over. Therefore, only one Eloquent model is kept in memory at any given time while iterating over the cursor.

{note} Since the `cursor` method only ever holds a single Eloquent model in memory at a time, it cannot eager load relationships. If you need to eager load relationships, consider using the `lazy` method instead.

Internally, the `cursor` method uses PHP generators to implement this functionality:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
    //
}
```

The `cursor` returns an `Illuminate\Support\LazyCollection` instance. Lazy collections allow you to use many of the collection methods available on typical Laravel collections while only loading a single model into memory at a time:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
    return $user->id > 500;
});

foreach ($users as $user) {
    echo $user->id;
}
```

Although the `cursor` method uses far less memory than a regular query (by only holding a single Eloquent model in memory at a time), it will still eventually run out of memory. This is due to PHP's PDO driver internally caching all raw query results in its buffer. If you're dealing with a very large number of Eloquent records, consider using the `lazy` method instead.

Advanced Subqueries

Subquery Selects

Eloquent also offers advanced subquery support, which allows you to pull information from related tables in a single query. For example, let's imagine that we have a table of flight `destinations` and a table of `flights` to destinations. The `flights` table contains an `arrived_at` column which indicates when the flight arrived at the destination.

Using the subquery functionality available to the query builder's `select` and `addSelect` methods, we can select all of the `destinations` and the name of the flight that most recently arrived at that destination using a single query:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
])->get();
```

Subquery Ordering

In addition, the query builder's `orderBy` function supports subqueries. Continuing to use our flight example, we may use this functionality to sort all destinations based on when the last flight arrived at that destination. Again, this may be done while executing a single database query:

```
return Destination::orderByDesc(
    Flight::select('arrived_at')
        ->whereColumn('destination_id', 'destinations.id')
        ->orderByDesc('arrived_at')
        ->limit(1)
)->get();
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records matching a given query, you may also retrieve single records using the `find`, `first`, or `firstWhere` methods. Instead of returning a collection of models, these methods return a single model instance:

```
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query
constraints...
$flight = Flight::firstWhere('active', 1);
```

Sometimes you may wish to retrieve the first result of a query or perform some other action if no results are found. The `firstOr` method will return the first result matching the query or, if no results are found, execute the given closure. The value returned by the closure will be considered the result of the `firstOr` method:

```
$model = Flight::where('legs', '>', 3)->firstOr(function () {
    // ...
});
```

Not Found Exceptions

Sometimes you may wish to throw an exception if a model is not found. This is particularly useful in routes or controllers. The `findOrFail` and `firstOrFail` methods will retrieve the first result of the query; however, if no result is found, an `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

If the `ModelNotFoundException` is not caught, a 404 HTTP response is automatically sent back to the client:

```
use App\Models\Flight;

Route::get('/api/flights/{id}', function ($id) {
    return Flight::findOrFail($id);
});
```

Retrieving Or Creating Models

The `firstOrCreate` method will attempt to locate a database record using the given column / value pairs. If the model can not be found in the database, a record will be inserted with the attributes resulting from merging the first array argument with the optional second array argument:

The `firstOrNew` method, like `firstOrCreate`, will attempt to locate a record in the database matching the given attributes. However, if a model is not found, a new model instance will be returned. Note that the model returned by `firstOrNew` has not yet been persisted to the database. You will need to manually call the `save` method to persist it:

```
use App\Models\Flight;

// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or create it with the name, delayed, and
arrival_time attributes...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and
arrival_time attributes...
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Retrieving Aggregates

When interacting with Eloquent models, you may also use the `count`, `sum`, `max`, and other aggregate methods provided by the Laravel query builder. As you might expect, these methods return a scalar value instead of an Eloquent model instance:

```
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

Inserting & Updating Models

Inserts

Of course, when using Eloquent, we don't only need to retrieve models from the database. We also need to insert new records. Thankfully, Eloquent makes it simple. To insert a new record into the database, you should instantiate a new model instance and set attributes on the model. Then, call the `save` method on the model instance:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\Request;

class FlightController extends Controller
{
    /**
     * Store a new flight in the database.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```

In this example, we assign the `name` field from the incoming HTTP request to the `name` attribute of the `App\Models\Flight` model instance. When we call the `save` method, a

record will be inserted into the database. The model's `created_at` and `updated_at` timestamps will automatically be set when the `save` method is called, so there is no need to set them manually.

Alternatively, you may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the `create` method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default. To learn more about mass assignment, please consult the mass assignment documentation.

Updates

The `save` method may also be used to update models that already exist in the database. To update a model, you should retrieve it and set any attributes you wish to update. Then, you should call the model's `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();
```

Mass Updates

Updates can also be performed against models that match a given query. In this example, all flights that are `active` and have a `destination` of `San Diego` will be marked as delayed:

```
Flight::where('active', 1)
      ->where('destination', 'San Diego')
      ->update(['delayed' => 1]);
```

The `update` method expects an array of column and value pairs representing the columns that should be updated.

{note} When issuing a mass update via Eloquent, the `saving`, `saved`, `updating`, and `updated` model events will not be fired for the updated models. This is because the models are never actually retrieved when issuing a mass update.

Examining Attribute Changes

Eloquent provides the `isDirty`, `isClean`, and `wasChanged` methods to examine the internal state of your model and determine how its attributes have changed from when the model was originally retrieved.

The `isDirty` method determines if any of the model's attributes have been changed since the model was retrieved. You may pass a specific attribute name to the `isDirty` method to determine if a particular attribute is dirty. The `isClean` will determine if an attribute has remained unchanged since the model was retrieved. This method also accepts an optional attribute argument:

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

The `wasChanged` method determines if any attributes were changed when the model was last saved within the current request cycle. If needed, you may pass an attribute name to see if a particular attribute was changed:

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged('first_name'); // false
```

The `getOriginal` method returns an array containing the original attributes of the model

regardless of any changes to the model since it was retrieved. If needed, you may pass a specific attribute name to get the original value of a particular attribute:

```
$user = User::find(1);

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Array of original attributes...
```

Mass Assignment

You may use the `create` method to "save" a new model using a single PHP statement. The inserted model instance will be returned to you by the method:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

However, before using the `create` method, you will need to specify either a `fillable` or `guarded` property on your model class. These properties are required because all Eloquent models are protected against mass assignment vulnerabilities by default.

A mass assignment vulnerability occurs when a user passes an unexpected HTTP request field and that field changes a column in your database that you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed to your model's `create` method, allowing the user to escalate themselves to an administrator.

So, to get started, you should define which model attributes you want to make mass assignable. You may do this using the `$fillable` property on the model. For example, let's make the `name` attribute of our `Flight` model mass assignable:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once you have specified which attributes are mass assignable, you may use the `create` method to insert a new record in the database. The `create` method returns the newly created model instance:

```php
$flight = Flight::create(['name' => 'London to Paris']);
```

If you already have a model instance, you may use the `fill` method to populate it with an array of attributes:

```php
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Mass Assignment & JSON Columns

When assigning JSON columns, each column's mass assignable key must be specified in your model's `$fillable` array. For security, Laravel does not support updating nested JSON attributes when using the `guarded` property:

```
/**
 * The attributes that are mass assignable.
 *
 * @var array
 */
protected $fillable = [
    'options->enabled',
];
```

Allowing Mass Assignment

If you would like to make all of your attributes mass assignable, you may define your model's `$guarded` property as an empty array. If you choose to unguard your model, you should take special care to always hand-craft the arrays passed to Eloquent's `fill`, `create`, and `update` methods:

```
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

Upserts

Occasionally, you may need to update an existing model or create a new model if no matching model exists. Like the `firstOrCreate` method, the `updateOrCreate` method persists the model, so there's no need to manually call the `save` method.

In the example below, if a flight exists with a `departure` location of `Oakland` and a `destination` location of `San Diego`, its `price` and `discounted` columns will be updated. If no such flight exists, a new flight will be created which has the attributes resulting from merging the first argument array with the second argument array:

```
$flight = Flight::updateOrCreate(
    ['departure' => 'Oakland', 'destination' => 'San Diego'],
    ['price' => 99, 'discounted' => 1]
);
```

If you would like to perform multiple "upserts" in a single query, then you should use the `upsert` method instead. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of the columns that should be updated if a matching record already exists in the database. The `upsert` method will automatically set the `created_at` and `updated_at` timestamps if timestamps are enabled on the model:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' =>
99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' =>
150]
], ['departure', 'destination'], ['price']);
```

Deleting Models

To delete a model, you may call the `delete` method on the model instance:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

You may call the `truncate` method to delete all of the model's associated database records. The `truncate` operation will also reset any auto-incrementing IDs on the model's associated table:

```
Flight::truncate();
```

Deleting An Existing Model By Its Primary Key

In the example above, we are retrieving the model from the database before calling the `delete` method. However, if you know the primary key of the model, you may delete the model without explicitly retrieving it by calling the `destroy` method. In addition to accepting the single primary key, the `destroy` method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```

{note} The `destroy` method loads each model individually and calls the `delete` method so that the `deleting` and `deleted` events are properly dispatched for each model.

Deleting Models Using Queries

Of course, you may build an Eloquent query to delete all models matching your query's criteria. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not dispatch model events for the models that are deleted:

```
$deletedRows = Flight::where('active', 0)->delete();
```

{note} When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be dispatched for the deleted models. This is because the models are never actually retrieved when executing the delete statement.

Soft Deleting

In addition to actually removing records from your database, Eloquent can also "soft delete" models. When models are soft deleted, they are not actually removed from your database. Instead, a `deleted_at` attribute is set on the model indicating the date and time at which the model was "deleted". To enable soft deletes for a model, add the `Illuminate\Database\Eloquent\SoftDeletes` trait to the model:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

{tip} The `SoftDeletes` trait will automatically cast the `deleted_at` attribute to a `DateTime` / `Carbon` instance for you.

You should also add the `deleted_at` column to your database table. The Laravel schema builder contains a helper method to create this column:

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. However, the model's database record will be left in the table. When querying a model that uses soft deletes, the soft deleted models will automatically be excluded from all query results.

To determine if a given model instance has been soft deleted, you may use the `trashed` method:

```
if ($flight->trashed()) {
    //
}
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model, you may call the `restore` method on a model instance. The `restore` method will set the model's `deleted_at` column to `null`:

```
$flight->restore();
```

You may also use the `restore` method in a query to restore multiple models. Again, like other "mass" operations, this will not dispatch any model events for the models that are restored:

```
Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

The `restore` method may also be used when building relationship queries:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. You may use the `forceDelete` method to permanently remove a soft deleted model from the database table:

```
$flight->forceDelete();
```

You may also use the `forceDelete` method when building Eloquent relationship queries:

```
$flight->history()->forceDelete();
```

Querying Soft Deleted Models

Including Soft Deleted Models

As noted above, soft deleted models will automatically be excluded from query results. However, you may force soft deleted models to be included in a query's results by calling the `withTrashed` method on the query:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

The `withTrashed` method may also be called when building a relationship query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = Flight::onlyTrashed()
                ->where('airline_id', 1)
                ->get();
```

Pruning Models

Sometimes you may want to periodically delete models that are no longer needed. To accomplish this, you may add the `Illuminate\Database\Eloquent\Prunable` or `Illuminate\Database\Eloquent\MassPrunable` trait to the models you would like to periodically prune. After adding one of the traits to the model, implement a `prunable` method which returns an Eloquent query builder that resolves the models that are no longer needed:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Flight extends Model
{
    use Prunable;

    /**
     * Get the prunable model query.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function prunable()
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

When marking models as `Prunable`, you may also define a `pruning` method on the model. This method will be called before the model is deleted. This method can be useful for deleting any additional resources associated with the model, such as stored files, before the model is permanently removed from the database:

```
/**
 * Prepare the model for pruning.
 *
 * @return void
 */
protected function pruning()
{
    //
}
```

After configuring your prunable model, you should schedule the `model:prune` Artisan command in your application's `App\Console\Kernel` class. You are free to choose the appropriate interval at which this command should be run:

```
/**
 * Define the application's command schedule.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
    $schedule->command('model:prune')->daily();
}
```

Behind the scenes, the `model:prune` command will automatically detect "Prunable" models within your application's `app/Models` directory. If your models are in a different location, you may use the `--model` option to specify the model class names:

```
$schedule->command('model:prune', [
    '--model' => [Address::class, Flight::class],
])->daily();
```

If you wish to exclude certain models from being pruned while pruning all other detected models, you may use the `--except` option:

```
$schedule->command('model:prune', [
    '--except' => [Address::class, Flight::class],
])->daily();
```

You may test your `prunable` query by executing the `model:prune` command with the `--pretend` option. When pretending, the `model:prune` command will simply report how many records would be pruned if the command were to actually run:

```
php artisan model:prune --pretend
```

{note} Soft deleting models will be permanently deleted (`forceDelete`) if they match the prunable query.

Mass Pruning

When models are marked with the `Illuminate\Database\Eloquent\MassPrunable` trait, models are deleted from the database using mass-deletion queries. Therefore, the `pruning` method will not be invoked, nor will the `deleting` and `deleted` model events be dispatched. This is because the models are never actually retrieved before deletion, thus making the pruning process much more efficient:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\MassPrunable;

class Flight extends Model
{
    use MassPrunable;

    /**
     * Get the prunable model query.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function prunable()
    {
        return static::where('created_at', '<=', now()->subMonth());
    }
}
```

Replicating Models

You may create an unsaved copy of an existing model instance using the `replicate` method. This method is particularly useful when you have model instances that share many of the same attributes:

```
use App\Models\Address;

$shipping = Address::create([
    'type' => 'shipping',
    'line_1' => '123 Example Street',
    'city' => 'Victorville',
    'state' => 'CA',
    'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
    'type' => 'billing'
]);

$billing->save();
```

To exclude one or more attributes from being replicated to the new model, you may pass an array to the `replicate` method:

```
$flight = Flight::create([
    'destination' => 'LAX',
    'origin' => 'LHR',
    'last_flown' => '2020-03-04 11:00:00',
    'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
    'last_flown',
    'last_pilot_id'
]);
```

Query Scopes

Global Scopes

Global scopes allow you to add constraints to all queries for a given model. Laravel's own soft delete functionality utilizes global scopes to only retrieve "non-deleted" models from the database. Writing your own global scopes can provide a convenient, easy way to make sure every query for a given model receives certain constraints.

Writing Global Scopes

Writing a global scope is simple. First, define a class that implements the `Illuminate\Database\Eloquent\Scope` interface. Laravel does not have a conventional location that you should place scope classes, so you are free to place this class in any directory that you wish.

The `Scope` interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints or other types of clauses to the query as needed:

```php
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
    /**
     * Apply the scope to a given Eloquent query builder.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $builder
     * @param  \Illuminate\Database\Eloquent\Model  $model
     * @return void
     */
    public function apply(Builder $builder, Model $model)
    {
        $builder->where('created_at', '<', now()->subYears(2000));
    }
}
```

{tip} If your global scope is adding columns to the select clause of the query, you should use the `addSelect` method instead of `select`. This will prevent the unintentional replacement of the query's existing select clause.

Applying Global Scopes

To assign a global scope to a model, you should override the model's `booted` method and invoke the model's `addGlobalScope` method. The `addGlobalScope` method accepts an instance of your scope as its only argument:

```php
<?php

namespace App\Models;

use App\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     *
     * @return void
     */
    protected static function booted()
    {
        static::addGlobalScope(new AncientScope);
    }
}
```

After adding the scope in the example above to the `App\Models\User` model, a call to the `User::all()` method will execute the following SQL query:

```sql
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

Anonymous Global Scopes

Eloquent also allows you to define global scopes using closures, which is particularly useful for simple scopes that do not warrant a separate class of their own. When defining a global scope using a closure, you should provide a scope name of your own choosing as the first argument to the `addGlobalScope` method:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     *
     * @return void
     */
    protected static function booted()
    {
        static::addGlobalScope('ancient', function (Builder $builder) {
            $builder->where('created_at', '<', now()->subYears(2000));
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. This method accepts the class name of the global scope as its only argument:

```php
User::withoutGlobalScope(AncientScope::class)->get();
```

Or, if you defined the global scope using a closure, you should pass the string name that you assigned to the global scope:

```php
User::withoutGlobalScope('ancient')->get();
```

If you would like to remove several or even all of the query's global scopes, you may use the `withoutGlobalScopes` method:

```
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();

// Remove some of the global scopes...
User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

Local Scopes

Local scopes allow you to define common sets of query constraints that you may easily re-use throughout your application. For example, you may need to frequently retrieve all users that are considered "popular". To define a scope, prefix an Eloquent model method with `scope`.

Scopes should always return the same query builder instance or `void`:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return void
     */
    public function scopeActive($query)
    {
        $query->where('active', 1);
    }
}
```

Utilizing A Local Scope

Once the scope has been defined, you may call the scope methods when querying the model. However, you should not include the `scope` prefix when calling the method. You can even chain calls to various scopes:

```php
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an `or` query operator may require the use of closures to achieve the correct logical grouping:

```
$users = User::popular()->orWhere(function (Builder $query) {
    $query->active();
})->get();
```

However, since this can be cumbersome, Laravel provides a "higher order" `orWhere` method that allows you to fluently chain scopes together without the use of closures:

```
$users = App\Models\User::popular()->orWhere->active()->get();
```

Dynamic Scopes

Sometimes you may wish to define a scope that accepts parameters. To get started, just add your additional parameters to your scope method's signature. Scope parameters should be defined after the `$query` parameter:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Scope a query to only include users of a given type.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @param  mixed  $type
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeOfType($query, $type)
    {
        return $query->where('type', $type);
    }
}
```

Once the expected arguments have been added to your scope method's signature, you may pass the arguments when calling the scope:

```
$users = User::ofType('admin')->get();
```

Comparing Models

Sometimes you may need to determine if two models are the "same" or not. The `is` and `isNot` methods may be used to quickly verify two models have the same primary key, table, and database connection or not:

```
if ($post->is($anotherPost)) {
    //
}

if ($post->isNot($anotherPost)) {
    //
}
```

The `is` and `isNot` methods are also available when using the `belongsTo`, `hasOne`, `morphTo`, and `morphOne` relationships. This method is particularly helpful when you would like to compare a related model without issuing a query to retrieve that model:

```
if ($post->author()->is($user)) {
    //
}
```

Events

{tip} Want to broadcast your Eloquent events directly to your client-side application? Check out Laravel's model event broadcasting.

Eloquent models dispatch several events, allowing you to hook into the following moments in a model's lifecycle: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored`, and `replicating`.

The `retrieved` event will dispatch when an existing model is retrieved from the database. When a new model is saved for the first time, the `creating` and `created` events will dispatch. The `updating` / `updated` events will dispatch when an existing model is modified and the `save` method is called. The `saving` / `saved` events will dispatch when a model is created or updated - even if the model's attributes have not been changed. Event names ending with `-ing` are dispatched before any changes to the model are persisted, while events ending with `-ed` are dispatched after the changes to the model are persisted.

To start listening to model events, define a `$dispatchesEvents` property on your Eloquent model. This property maps various points of the Eloquent model's lifecycle to your own event classes. Each model event class should expect to receive an instance of the affected model via its constructor:

```php
<?php

namespace App\Models;

use App\Events\UserDeleted;
use App\Events\UserSaved;
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
    use Notifiable;

    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

After defining and mapping your Eloquent events, you may use event listeners to handle the events.

{note} When issuing a mass update or delete query via Eloquent, the `saved`, `updated`, `deleting`, and `deleted` model events will not be dispatched for the affected models. This is because the models are never actually retrieved when performing mass updates or deletes.

Using Closures

Instead of using custom event classes, you may register closures that execute when various model events are dispatched. Typically, you should register these closures in the `booted` method of your model:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booted" method of the model.
     *
     * @return void
     */
    protected static function booted()
    {
        static::created(function ($user) {
            //
        });
    }
}
```

If needed, you may utilize queueable anonymous event listeners when registering model events. This will instruct Laravel to execute the model event listener in the background using your application's queue:

```php
use function Illuminate\Events\queueable;

static::created(queueable(function ($user) {
    //
}));
```

Observers

Defining Observers

If you are listening for many events on a given model, you may use observers to group all of your listeners into a single class. Observer classes have method names which reflect the Eloquent events you wish to listen for. Each of these methods receives the affected model as their only argument. The `make:observer` Artisan command is the easiest way to create

a new observer class:

```
php artisan make:observer UserObserver --model=User
```

This command will place the new observer in your `App/Observers` directory. If this directory does not exist, Artisan will create it for you. Your fresh observer will look like the following:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Handle the User "created" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }

    /**
     * Handle the User "updated" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function updated(User $user)
    {
        //
    }

    /**
     * Handle the User "deleted" event.
     *
     * @param  \App\Models\User  $user
     * @return void
```

```
     */
    public function deleted(User $user)
    {
        //
    }

    /**
     * Handle the User "forceDeleted" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function forceDeleted(User $user)
    {
        //
    }
}
```

To register an observer, you need to call the `observe` method on the model you wish to observe. You may register observers in the `boot` method of your application's `App\Providers\EventServiceProvider` service provider:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Register any events for your application.
 *
 * @return void
 */
public function boot()
{
    User::observe(UserObserver::class);
}
```

{tip} There are additional events an observer can listen to, such as `saving` and `retrieved`. These events are described within the events documentation.

Observers & Database Transactions

When models are being created within a database transaction, you may want to instruct an observer to only execute its event handlers after the database transaction is committed. You

may accomplish this by defining an `$afterCommit` property on the observer. If a database transaction is not in progress, the event handlers will execute immediately:

```
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
    /**
     * Handle events after all transactions are committed.
     *
     * @var bool
     */
    public $afterCommit = true;

    /**
     * Handle the User "created" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function created(User $user)
    {
        //
    }
}
```

Muting Events

You may occasionally need to temporarily "mute" all events fired by a model. You may achieve this using the `withoutEvents` method. The `withoutEvents` method accepts a closure as its only argument. Any code executed within this closure will not dispatch model events, and any value returned by the closure will be returned by the `withoutEvents` method:

```
use App\Models\User;

$user = User::withoutEvents(function () use () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Saving A Single Model Without Events

Sometimes you may wish to "save" a given model without dispatching any events. You may accomplish this using the `saveQuietly` method:

```
$user = User::findOrFail(1);

$user->name = 'Victoria Faith';

$user->saveQuietly();
```

Encryption

- Introduction
- Configuration
- Using The Encrypter

Introduction

Laravel's encryption services provide a simple, convenient interface for encrypting and decrypting text via OpenSSL using AES-256 and AES-128 encryption. All of Laravel's encrypted values are signed using a message authentication code (MAC) so that their underlying value can not be modified or tampered with once encrypted.

Configuration

Before using Laravel's encrypter, you must set the `key` configuration option in your `config/app.php` configuration file. This configuration value is driven by the `APP_KEY` environment variable. You should use the `php artisan key:generate` command to generate this variable's value since the `key:generate` command will use PHP's secure random bytes generator to build a cryptographically secure key for your application. Typically, the value of the `APP_KEY` environment variable will be generated for you during Laravel's installation.

Using The Encrypter

Encrypting A Value

You may encrypt a value using the `encryptString` method provided by the `Crypt` facade. All encrypted values are encrypted using OpenSSL and the AES-256-CBC cipher. Furthermore, all encrypted values are signed with a message authentication code (MAC). The integrated message authentication code will prevent the decryption of any values that have been tampered with by malicious users:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Crypt;

class DigitalOceanTokenController extends Controller
{
    /**
     * Store a DigitalOcean API token for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function storeSecret(Request $request)
    {
        $request->user()->fill([
            'token' => Crypt::encryptString($request->token),
        ])->save();
    }
}
```

Decrypting A Value

You may decrypt values using the `decryptString` method provided by the `Crypt` facade. If the value can not be properly decrypted, such as when the message authentication code is invalid, an `Illuminate\Contracts\Encryption\DecryptException` will be thrown:

```
use Illuminate\Contracts\Encryption\DecryptException;
use Illuminate\Support\Facades\Crypt;

try {
    $decrypted = Crypt::decryptString($encryptedValue);
} catch (DecryptException $e) {
    //
}
```

Error Handling

- Introduction
- Configuration
- The Exception Handler
    - Reporting Exceptions
    - Ignoring Exceptions By Type
    - Rendering Exceptions
    - Reportable & Renderable Exceptions
- HTTP Exceptions
    - Custom HTTP Error Pages

Introduction

When you start a new Laravel project, error and exception handling is already configured for you. The `App\Exceptions\Handler` class is where all exceptions thrown by your application are logged and then rendered to the user. We'll dive deeper into this class throughout this documentation.

Configuration

The `debug` option in your `config/app.php` configuration file determines how much information about an error is actually displayed to the user. By default, this option is set to respect the value of the `APP_DEBUG` environment variable, which is stored in your `.env` file.

During local development, you should set the `APP_DEBUG` environment variable to `true`. **In your production environment, this value should always be `false`. If the value is set to `true` in production, you risk exposing sensitive configuration values to your application's end users.**

The Exception Handler

Reporting Exceptions

All exceptions are handled by the `App\Exceptions\Handler` class. This class contains a `register` method where you may register custom exception reporting and rendering callbacks. We'll examine each of these concepts in detail. Exception reporting is used to log exceptions or send them to an external service like Flare, Bugsnag or Sentry. By default, exceptions will be logged based on your logging configuration. However, you are free to log exceptions however you wish.

For example, if you need to report different types of exceptions in different ways, you may use the `reportable` method to register a closure that should be executed when an exception of a given type needs to be reported. Laravel will deduce what type of exception the closure reports by examining the type-hint of the closure:

```
use App\Exceptions\InvalidOrderException;

/**
 * Register the exception handling callbacks for the application.
 *
 * @return void
 */
public function register()
{
    $this->reportable(function (InvalidOrderException $e) {
        //
    });
}
```

When you register a custom exception reporting callback using the `reportable` method, Laravel will still log the exception using the default logging configuration for the application. If you wish to stop the propagation of the exception to the default logging stack, you may use the `stop` method when defining your reporting callback or return `false` from the callback:

```
$this->reportable(function (InvalidOrderException $e) {
    //
})->stop();

$this->reportable(function (InvalidOrderException $e) {
    return false;
});
```

{tip} To customize the exception reporting for a given exception, you may also utilize reportable exceptions.

Global Log Context

If available, Laravel automatically adds the current user's ID to every exception's log message as contextual data. You may define your own global contextual data by overriding the `context` method of your application's `App\Exceptions\Handler` class. This information will be included in every exception's log message written by your application:

```
/**
 * Get the default context variables for logging.
 *
 * @return array
 */
protected function context()
{
    return array_merge(parent::context(), [
        'foo' => 'bar',
    ]);
}
```

Exception Log Context

While adding context to every log message can be useful, sometimes a particular exception may have unique context that you would like to include in your logs. By defining a `context` method on one of your application's custom exceptions, you may specify any data relevant to that exception that should be added to the exception's log entry:

```php
<?php

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception
{
    // ...

    /**
     * Get the exception's context information.
     *
     * @return array
     */
    public function context()
    {
        return ['order_id' => $this->orderId];
    }
}
```

The `report` Helper

Sometimes you may need to report an exception but continue handling the current request. The `report` helper function allows you to quickly report an exception via the exception handler without rendering an error page to the user:

```php
public function isValid($value)
{
    try {
        // Validate the value...
    } catch (Throwable $e) {
        report($e);

        return false;
    }
}
```

Ignoring Exceptions By Type

When building your application, there will be some types of exceptions you simply want to ignore and never report. Your application's exception handler contains a `$dontReport` property which is initialized to an empty array. Any classes that you add to this property will never be reported; however, they may still have custom rendering logic:

```
use App\Exceptions\InvalidOrderException;

/**
 * A list of the exception types that should not be reported.
 *
 * @var array
 */
protected $dontReport = [
    InvalidOrderException::class,
];
```

{tip} Behind the scenes, Laravel already ignores some types of errors for you, such as exceptions resulting from 404 HTTP "not found" errors or 419 HTTP responses generated by invalid CSRF tokens.

Rendering Exceptions

By default, the Laravel exception handler will convert exceptions into an HTTP response for you. However, you are free to register a custom rendering closure for exceptions of a given type. You may accomplish this via the `renderable` method of your exception handler.

The closure passed to the `renderable` method should return an instance of `Illuminate\Http\Response`, which may be generated via the `response` helper. Laravel will deduce what type of exception the closure renders by examining the type-hint of the closure:

```
use App\Exceptions\InvalidOrderException;

/**
 * Register the exception handling callbacks for the application.
 *
 * @return void
 */
public function register()
{
    $this->renderable(function (InvalidOrderException $e, $request) {
        return response()->view('errors.invalid-order', [], 500);
    });
}
```

You may also use the `renderable` method to override the rendering behavior for built-in Laravel or Symfony exceptions such as `NotFoundHttpException`. If the closure given to the `renderable` method does not return a value, Laravel's default exception rendering will be utilized:

```
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

/**
 * Register the exception handling callbacks for the application.
 *
 * @return void
 */
public function register()
{
    $this->renderable(function (NotFoundHttpException $e, $request) {
        if ($request->is('api/*')) {
            return response()->json([
                'message' => 'Record not found.'
            ], 404);
        }
    });
}
```

Reportable & Renderable Exceptions

Instead of type-checking exceptions in the exception handler's `register` method, you may define `report` and `render` methods directly on your custom exceptions. When these methods exist, they will be automatically called by the framework:

```
<?php

namespace App\Exceptions;

use Exception;

class InvalidOrderException extends Exception
{
    /**
     * Report the exception.
     *
     * @return bool|null
     */
    public function report()
    {
        //
    }

    /**
     * Render the exception into an HTTP response.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function render($request)
    {
        return response(...);
    }
}
```

If your exception extends an exception that is already renderable, such as a built-in Laravel or Symfony exception, you may return `false` from the exception's `render` method to render the exception's default HTTP response:

```
/**
 * Render the exception into an HTTP response.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function render($request)
{
    // Determine if the exception needs custom rendering...

    return false;
}
```

If your exception contains custom reporting logic that is only necessary when certain conditions are met, you may need to instruct Laravel to sometimes report the exception using the default exception handling configuration. To accomplish this, you may return `false` from the exception's `report` method:

```
/**
 * Report the exception.
 *
 * @return bool|null
 */
public function report()
{
    // Determine if the exception needs custom reporting...

    return false;
}
```

{tip} You may type-hint any required dependencies of the `report` method and they will automatically be injected into the method by Laravel's service container.

HTTP Exceptions

Some exceptions describe HTTP error codes from the server. For example, this may be a "page not found" error (404), an "unauthorized error" (401) or even a developer generated 500 error. In order to generate such a response from anywhere in your application, you may use the `abort` helper:

```
abort(404);
```

Custom HTTP Error Pages

Laravel makes it easy to display custom error pages for various HTTP status codes. For example, if you wish to customize the error page for 404 HTTP status codes, create a `resources/views/errors/404.blade.php` view template. This view will be rendered on all 404 errors generated by your application. The views within this directory should be named to match the HTTP status code they correspond to. The `Symfony\Component\HttpKernel\Exception\HttpException` instance raised by the `abort` function will be passed to the view as an `$exception` variable:

```
<h2>{{ $exception->getMessage() }}</h2>
```

You may publish Laravel's default error page templates using the `vendor:publish` Artisan command. Once the templates have been published, you may customize them to your liking:

```
php artisan vendor:publish --tag=laravel-errors
```

Fallback HTTP Error Pages

You may also define a "fallback" error page for a given series of HTTP status codes. This page will be rendered if there is not a corresponding page for the specific HTTP status code that occurred. To accomplish this, define a `4xx.blade.php` template and a `5xx.blade.php` template in your application's `resources/views/errors` directory.

File Storage

- Introduction
- Configuration
    - The Local Driver
    - The Public Disk
    - Driver Prerequisites
    - Amazon S3 Compatible Filesystems
    - Caching
- Obtaining Disk Instances
    - On-Demand Disks
- Retrieving Files
    - Downloading Files
    - File URLs
    - File Metadata
- Storing Files
    - File Uploads
    - File Visibility
- Deleting Files
- Directories
- Custom Filesystems

Introduction

Laravel provides a powerful filesystem abstraction thanks to the wonderful Flysystem PHP package by Frank de Jonge. The Laravel Flysystem integration provides simple drivers for working with local filesystems, SFTP, and Amazon S3. Even better, it's amazingly simple to switch between these storage options between your local development machine and production server as the API remains the same for each system.

Configuration

Laravel's filesystem configuration file is located at `config/filesystems.php`. Within this file, you may configure all of your filesystem "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file so you can modify the configuration to reflect your storage preferences and credentials.

The `local` driver interacts with files stored locally on the server running the Laravel application while the `s3` driver is used to write to Amazon's S3 cloud storage service.

{tip} You may configure as many disks as you like and may even have multiple disks that use the same driver.

The Local Driver

When using the `local` driver, all file operations are relative to the `root` directory defined in your `filesystems` configuration file. By default, this value is set to the `storage/app` directory. Therefore, the following method would write to `storage/app/example.txt`:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('local')->put('example.txt', 'Contents');
```

The Public Disk

The `public` disk included in your application's `filesystems` configuration file is intended for files that are going to be publicly accessible. By default, the `public` disk uses the `local` driver and stores its files in `storage/app/public`.

To make these files accessible from the web, you should create a symbolic link from `public/storage` to `storage/app/public`. Utilizing this folder convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like Envoyer.

To create the symbolic link, you may use the `storage:link` Artisan command:

```
php artisan storage:link
```

Once a file has been stored and the symbolic link has been created, you can create a URL to the files using the `asset` helper:

```
echo asset('storage/file.txt');
```

You may configure additional symbolic links in your `filesystems` configuration file. Each of the configured links will be created when you run the `storage:link` command:

```
'links' => [
    public_path('storage') => storage_path('app/public'),
    public_path('images') => storage_path('app/images'),
],
```

Driver Prerequisites

Composer Packages

Before using the S3 or SFTP drivers, you will need to install the appropriate package via the Composer package manager:

- Amazon S3: `composer require --with-all-dependencies league/flysystem-aws-s3-v3 "^1.0"`
- SFTP: `composer require league/flysystem-sftp "~1.0"`

In addition, you may choose to install a cached adapter for increased performance:

- CachedAdapter: `composer require league/flysystem-cached-adapter "~1.0"`

S3 Driver Configuration

The S3 driver configuration information is located in your `config/filesystems.php` configuration file. This file contains an example configuration array for an S3 driver. You are free to modify this array with your own S3 configuration and credentials. For convenience, these environment variables match the naming convention used by the AWS CLI.

FTP Driver Configuration

Laravel's Flysystem integrations work great with FTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure an FTP filesystem, you may use the configuration example below:

```
'ftp' => [
    'driver' => 'ftp',
    'host' => 'ftp.example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Optional FTP Settings...
    // 'port' => 21,
    // 'root' => '',
    // 'passive' => true,
    // 'ssl' => true,
    // 'timeout' => 30,
],
```

SFTP Driver Configuration

Laravel's Flysystem integrations work great with SFTP; however, a sample configuration is not included with the framework's default `filesystems.php` configuration file. If you need to configure an SFTP filesystem, you may use the configuration example below:

```
'sftp' => [
    'driver' => 'sftp',
    'host' => 'example.com',
    'username' => 'your-username',
    'password' => 'your-password',

    // Settings for SSH key based authentication...
    'privateKey' => '/path/to/privateKey',
    'password' => 'encryption-password',

    // Optional SFTP Settings...
    // 'port' => 22,
    // 'root' => '',
    // 'timeout' => 30,
],
```

Amazon S3 Compatible Filesystems

By default, your application's `filesystems` configuration file contains a disk configuration for the `s3` disk. In addition to using this disk to interact with Amazon S3, you may use it to interact with any S3 compatible file storage service such as MinIO or DigitalOcean Spaces.

Typically, after updating the disk's credentials to match the credentials of the service you are planning to use, you only need to update the value of the `url` configuration option. This option's value is typically defined via the `AWS_ENDPOINT` environment variable:

```
'endpoint' => env('AWS_ENDPOINT', 'https://minio:9000'),
```

Caching

To enable caching for a given disk, you may add a `cache` directive to the disk's configuration options. The `cache` option should be an array of caching options containing the `disk` name, the `expire` time in seconds, and the cache `prefix`:

```
's3' => [
    'driver' => 's3',

    // Other Disk Options...

    'cache' => [
        'store' => 'memcached',
        'expire' => 600,
        'prefix' => 'cache-prefix',
    ],
],
```

Obtaining Disk Instances

The `Storage` facade may be used to interact with any of your configured disks. For example, you may use the `put` method on the facade to store an avatar on the default disk. If you call methods on the `Storage` facade without first calling the `disk` method, the method will automatically be passed to the default disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('avatars/1', $content);
```

If your application interacts with multiple disks, you may use the `disk` method on the `Storage` facade to work with files on a particular disk:

```
Storage::disk('s3')->put('avatars/1', $content);
```

On-Demand Disks

Sometimes you may wish to create a disk at runtime using a given configuration without that configuration actually being present in your application's `filesystems` configuration file. To accomplish this, you may pass a configuration array to the `Storage` facade's `build` method:

```
use Illuminate\Support\Facades\Storage;

$disk = Storage::build([
    'driver' => 'local',
    'root' => '/path/to/root',
]);

$disk->put('image.jpg', $content);
```

Retrieving Files

The `get` method may be used to retrieve the contents of a file. The raw string contents of the file will be returned by the method. Remember, all file paths should be specified relative to the disk's "root" location:

```
$contents = Storage::get('file.jpg');
```

The `exists` method may be used to determine if a file exists on the disk:

```
if (Storage::disk('s3')->exists('file.jpg')) {
    // ...
}
```

The `missing` method may be used to determine if a file is missing from the disk:

```
if (Storage::disk('s3')->missing('file.jpg')) {
    // ...
}
```

Downloading Files

The `download` method may be used to generate a response that forces the user's browser to download the file at the given path. The `download` method accepts a filename as the second argument to the method, which will determine the filename that is seen by the user downloading the file. Finally, you may pass an array of HTTP headers as the third argument to the method:

```
return Storage::download('file.jpg');

return Storage::download('file.jpg', $name, $headers);
```

File URLs

You may use the `url` method to get the URL for a given file. If you are using the `local` driver, this will typically just prepend `/storage` to the given path and return a relative URL to the file. If you are using the `s3` driver, the fully qualified remote URL will be returned:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::url('file.jpg');
```

When using the `local` driver, all files that should be publicly accessible should be placed in the `storage/app/public` directory. Furthermore, you should create a symbolic link at `public/storage` which points to the `storage/app/public` directory.

{note} When using the `local` driver, the return value of `url` is not URL encoded. For this reason, we recommend always storing your files using names that will create valid URLs.

Temporary URLs

Using the `temporaryUrl` method, you may create temporary URLs to files stored using the `s3` driver. This method accepts a path and a `DateTime` instance specifying when the URL should expire:

```
use Illuminate\Support\Facades\Storage;

$url = Storage::temporaryUrl(
    'file.jpg', now()->addMinutes(5)
);
```

If you need to specify additional S3 request parameters, you may pass the array of request parameters as the third argument to the `temporaryUrl` method:

```
$url = Storage::temporaryUrl(
    'file.jpg',
    now()->addMinutes(5),
    [
        'ResponseContentType' => 'application/octet-stream',
        'ResponseContentDisposition' => 'attachment;
filename=file2.jpg',
    ]
);
```

URL Host Customization

If you would like to pre-define the host for URLs generated using the `Storage` facade, you may add a `url` option to the disk's configuration array:

```
'public' => [
    'driver' => 'local',
    'root' => storage_path('app/public'),
    'url' => env('APP_URL').'/storage',
    'visibility' => 'public',
],
```

File Metadata

In addition to reading and writing files, Laravel can also provide information about the files themselves. For example, the `size` method may be used to get the size of a file in bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file.jpg');
```

The `lastModified` method returns the UNIX timestamp of the last time the file was modified:

```
$time = Storage::lastModified('file.jpg');
```

File Paths

You may use the `path` method to get the path for a given file. If you are using the `local` driver, this will return the absolute path to the file. If you are using the `s3` driver, this method will return the relative path to the file in the S3 bucket:

```
use Illuminate\Support\Facades\Storage;

$path = Storage::path('file.jpg');
```

Storing Files

The `put` method may be used to store file contents on a disk. You may also pass a PHP `resource` to the `put` method, which will use Flysystem's underlying stream support. Remember, all file paths should be specified relative to the "root" location configured for the disk:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);

Storage::put('file.jpg', $resource);
```

Automatic Streaming

Streaming files to storage offers significantly reduced memory usage. If you would like Laravel to automatically manage streaming a given file to your storage location, you may use the `putFile` or `putFileAs` method. This method accepts either an `Illuminate\Http\File` or `Illuminate\Http\UploadedFile` instance and will automatically stream the file to your desired location:

```
use Illuminate\Http\File;
use Illuminate\Support\Facades\Storage;

// Automatically generate a unique ID for filename...
$path = Storage::putFile('photos', new File('/path/to/photo'));

// Manually specify a filename...
$path = Storage::putFileAs('photos', new File('/path/to/photo'),
'photo.jpg');
```

There are a few important things to note about the `putFile` method. Note that we only specified a directory name and not a filename. By default, the `putFile` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `putFile` method so you can store the path, including the generated filename, in your database.

The `putFile` and `putFileAs` methods also accept an argument to specify the "visibility" of the stored file. This is particularly useful if you are storing the file on a cloud disk such as

Amazon S3 and would like the file to be publicly accessible via generated URLs:

```
Storage::putFile('photos', new File('/path/to/photo'), 'public');
```

Prepending & Appending To Files

The `prepend` and `append` methods allow you to write to the beginning or end of a file:

```
Storage::prepend('file.log', 'Prepended Text');

Storage::append('file.log', 'Appended Text');
```

Copying & Moving Files

The `copy` method may be used to copy an existing file to a new location on the disk, while the `move` method may be used to rename or move an existing file to a new location:

```
Storage::copy('old/file.jpg', 'new/file.jpg');

Storage::move('old/file.jpg', 'new/file.jpg');
```

File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as photos and documents. Laravel makes it very easy to store uploaded files using the `store` method on an uploaded file instance. Call the `store` method with the path at which you wish to store the uploaded file:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserAvatarController extends Controller
{
    /**
     * Update the avatar for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        $path = $request->file('avatar')->store('avatars');

        return $path;
    }
}
```

There are a few important things to note about this example. Note that we only specified a directory name, not a filename. By default, the `store` method will generate a unique ID to serve as the filename. The file's extension will be determined by examining the file's MIME type. The path to the file will be returned by the `store` method so you can store the path, including the generated filename, in your database.

You may also call the `putFile` method on the `Storage` facade to perform the same file storage operation as the example above:

```php
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Specifying A File Name

If you do not want a filename to be automatically assigned to your stored file, you may use the `storeAs` method, which receives the path, the filename, and the (optional) disk as its arguments:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

You may also use the `putFileAs` method on the `Storage` facade, which will perform the same file storage operation as the example above:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

{note} Unprintable and invalid unicode characters will automatically be removed from file paths. Therefore, you may wish to sanitize your file paths before passing them to Laravel's file storage methods. File paths are normalized using the `League\Flysystem\Util::normalizePath` method.

Specifying A Disk

By default, this uploaded file's `store` method will use your default disk. If you would like to specify another disk, pass the disk name as the second argument to the `store` method:

```
$path = $request->file('avatar')->store(
    'avatars/'.$request->user()->id, 's3'
);
```

If you are using the `storeAs` method, you may pass the disk name as the third argument to the method:

```
$path = $request->file('avatar')->storeAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

Other Uploaded File Information

If you would like to get the original name and extension of the uploaded file, you may do so using the `getClientOriginalName` and `getClientOriginalExtension` methods:

```
$file = $request->file('avatar');

$name = $file->getClientOriginalName();
$extension = $file->getClientOriginalExtension();
```

However, keep in mind that the `getClientOriginalName` and `getClientOriginalExtension` methods are considered unsafe, as the file name and extension may be tampered with by a malicious user. For this reason, you should typically prefer the `hashName` and `extension` methods to get a name and an extension for the given file upload:

```
$file = $request->file('avatar');

$name = $file->hashName(); // Generate a unique, random name...
$extension = $file->extension(); // Determine the file's extension based
on the file's MIME type...
```

File Visibility

In Laravel's Flysystem integration, "visibility" is an abstraction of file permissions across multiple platforms. Files may either be declared `public` or `private`. When a file is declared `public`, you are indicating that the file should generally be accessible to others. For example, when using the S3 driver, you may retrieve URLs for `public` files.

You can set the visibility when writing the file via the `put` method:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents, 'public');
```

If the file has already been stored, its visibility can be retrieved and set via the `getVisibility` and `setVisibility` methods:

```
$visibility = Storage::getVisibility('file.jpg');

Storage::setVisibility('file.jpg', 'public');
```

When interacting with uploaded files, you may use the `storePublicly` and `storePubliclyAs` methods to store the uploaded file with `public` visibility:

```
$path = $request->file('avatar')->storePublicly('avatars', 's3');

$path = $request->file('avatar')->storePubliclyAs(
    'avatars',
    $request->user()->id,
    's3'
);
```

Local Files & Visibility

When using the `local` driver, `public` visibility translates to `0755` permissions for directories and `0644` permissions for files. You can modify the permissions mappings in your application's `filesystems` configuration file:

```
'local' => [
    'driver' => 'local',
    'root' => storage_path('app'),
    'permissions' => [
        'file' => [
            'public' => 0664,
            'private' => 0600,
        ],
        'dir' => [
            'public' => 0775,
            'private' => 0700,
        ],
    ],
],
```

Deleting Files

The `delete` method accepts a single filename or an array of files to delete:

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');

Storage::delete(['file.jpg', 'file2.jpg']);
```

If necessary, you may specify the disk that the file should be deleted from:

```
use Illuminate\Support\Facades\Storage;

Storage::disk('s3')->delete('path/file.jpg');
```

Directories

Get All Files Within A Directory

The `files` method returns an array of all of the files in a given directory. If you would like to retrieve a list of all files within a given directory including all subdirectories, you may use the `allFiles` method:

```
use Illuminate\Support\Facades\Storage;

$files = Storage::files($directory);

$files = Storage::allFiles($directory);
```

Get All Directories Within A Directory

The `directories` method returns an array of all the directories within a given directory. Additionally, you may use the `allDirectories` method to get a list of all directories within a given directory and all of its subdirectories:

```
$directories = Storage::directories($directory);

$directories = Storage::allDirectories($directory);
```

Create A Directory

The `makeDirectory` method will create the given directory, including any needed subdirectories:

```
Storage::makeDirectory($directory);
```

Delete A Directory

Finally, the `deleteDirectory` method may be used to remove a directory and all of its

files:

```
Storage::deleteDirectory($directory);
```

Custom Filesystems

Laravel's Flysystem integration provides support for several "drivers" out of the box; however, Flysystem is not limited to these and has adapters for many other storage systems. You can create a custom driver if you want to use one of these additional adapters in your Laravel application.

In order to define a custom filesystem you will need a Flysystem adapter. Let's add a community maintained Dropbox adapter to our project:

```
composer require spatie/flysystem-dropbox
```

Next, you can register the driver within the `boot` method of one of your application's service providers. To accomplish this, you should use the `extend` method of the `Storage` facade:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Storage;
use Illuminate\Support\ServiceProvider;
use League\Flysystem\Filesystem;
use Spatie\Dropbox\Client as DropboxClient;
use Spatie\FlysystemDropbox\DropboxAdapter;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Storage::extend('dropbox', function ($app, $config) {
            $client = new DropboxClient(
                $config['authorization_token']
            );

            return new Filesystem(new DropboxAdapter($client));
        });
    }
}
```

The first argument of the `extend` method is the name of the driver and the second is a closure that receives the `$app` and `$config` variables. The closure must return an instance of `League\Flysystem\Filesystem`. The `$config` variable contains the values defined in `config/filesystems.php` for the specified disk.

Once you have created and registered the extension's service provider, you may use the `dropbox` driver in your `config/filesystems.php` configuration file.

Hashing

- Introduction
- Configuration
- Basic Usage
    - Hashing Passwords
    - Verifying That A Password Matches A Hash
    - Determining If A Password Needs To Be Rehashed

Introduction

The Laravel `Hash` facade provides secure Bcrypt and Argon2 hashing for storing user passwords. If you are using one of the Laravel application starter kits, Bcrypt will be used for registration and authentication by default.

Bcrypt is a great choice for hashing passwords because its "work factor" is adjustable, which means that the time it takes to generate a hash can be increased as hardware power increases. When hashing passwords, slow is good. The longer an algorithm takes to hash a password, the longer it takes malicious users to generate "rainbow tables" of all possible string hash values that may be used in brute force attacks against applications.

Configuration

The default hashing driver for your application is configured in your application's `config/hashing.php` configuration file. There are currently several supported drivers: Bcrypt and Argon2 (Argon2i and Argon2id variants).

{note} The Argon2i driver requires PHP 7.2.0 or greater and the Argon2id driver requires PHP 7.3.0 or greater.

Basic Usage

Hashing Passwords

You may hash a password by calling the `make` method on the `Hash` facade:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;

class PasswordController extends Controller
{
    /**
     * Update the password for the user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request)
    {
        // Validate the new password length...

        $request->user()->fill([
            'password' => Hash::make($request->newPassword)
        ])->save();
    }
}
```

Adjusting The Bcrypt Work Factor

If you are using the Bcrypt algorithm, the `make` method allows you to manage the work factor of the algorithm using the `rounds` option; however, the default work factor managed by Laravel is acceptable for most applications:

```
$hashed = Hash::make('password', [
    'rounds' => 12,
]);
```

Adjusting The Argon2 Work Factor

If you are using the Argon2 algorithm, the `make` method allows you to manage the work factor of the algorithm using the `memory`, `time`, and `threads` options; however, the default values managed by Laravel are acceptable for most applications:

```
$hashed = Hash::make('password', [
    'memory' => 1024,
    'time' => 2,
    'threads' => 2,
]);
```

{tip} For more information on these options, please refer to the official PHP documentation regarding Argon hashing.

Verifying That A Password Matches A Hash

The `check` method provided by the `Hash` facade allows you to verify that a given plain-text string corresponds to a given hash:

```
if (Hash::check('plain-text', $hashedPassword)) {
    // The passwords match...
}
```

Determining If A Password Needs To Be Rehashed

The `needsRehash` method provided by the `Hash` facade allows you to determine if the work factor used by the hasher has changed since the password was hashed. Some applications choose to perform this check during the application's authentication process:

```
if (Hash::needsRehash($hashed)) {
    $hashed = Hash::make('plain-text');
}
```

Helpers

- Introduction
- Available Methods

Introduction

Laravel includes a variety of global "helper" PHP functions. Many of these functions are used by the framework itself; however, you are free to use them in your own applications if you find them convenient.

Available Methods

Arrays & Objects

Arr::accessible Arr::add Arr::collapse Arr::crossJoin Arr::divide Arr::dot Arr::except Arr::exists Arr::first Arr::flatten Arr::forget Arr::get Arr::has Arr::hasAny Arr::isAssoc Arr::last Arr::only Arr::pluck Arr::prepend Arr::pull Arr::query Arr::random Arr::set Arr::shuffle Arr::sort Arr::sortRecursive Arr::toCssClasses Arr::undot Arr::where Arr::whereNotNull Arr::wrap data_fill data_get data_set head last

Paths

app_path base_path config_path database_path mix public_path resource_path storage_path

Strings

__ class_basename e preg_replace_array Str::after Str::afterLast Str::ascii Str::before Str::beforeLast Str::between Str::camel Str::contains Str::containsAll Str::endsWith Str::finish Str::headline Str::is Str::isAscii Str::isUuid Str::kebab Str::length Str::limit Str::lower Str::markdown Str::mask Str::orderedUuid Str::padBoth Str::padLeft Str::padRight Str::plural Str::pluralStudly Str::random Str::remove Str::replace Str::replaceArray Str::replaceFirst Str::replaceLast Str::reverse Str::singular Str::slug Str::snake Str::start Str::startsWith Str::studly Str::substr Str::substrCount Str::title Str::ucfirst Str::upper Str::uuid Str::wordCount Str::words trans trans_choice

Fluent Strings

after afterLast append ascii basename before beforeLast camel contains containsAll dirname endsWith exactly explode finish is isAscii isEmpty isNotEmpty isUuid kebab length limit lower

ltrim markdown mask match matchAll padBoth padLeft padRight pipe plural prepend remove replace replaceArray replaceFirst replaceLast replaceMatches rtrim singular slug snake split start startsWith studly substr tap test title trim ucfirst upper when whenEmpty wordCount words

URLs

action asset route secure_asset secure_url url

Miscellaneous

abort abort_if abort_unless app auth back bcrypt blank broadcast cache class_uses_recursive collect config cookie csrf_field csrf_token dd dispatch dump env event filled info logger method_field now old optional policy redirect report request rescue resolve response retry session tap throw_if throw_unless today trait_uses_recursive transform validator value view with

Method Listing

Arrays & Objects

`Arr::accessible()` {.collection-method .first-collection-method}

The `Arr::accessible` method determines if the given value is array accessible:

```
use Illuminate\Support\Arr;
use Illuminate\Support\Collection;

$isAccessible = Arr::accessible(['a' => 1, 'b' => 2]);

// true

$isAccessible = Arr::accessible(new Collection);

// true

$isAccessible = Arr::accessible('abc');

// false

$isAccessible = Arr::accessible(new stdClass);

// false
```

`Arr::add()` {.collection-method}

The `Arr::add` method adds a given key / value pair to an array if the given key doesn't already exist in the array or is set to `null`:

```
use Illuminate\Support\Arr;

$array = Arr::add(['name' => 'Desk'], 'price', 100);

// ['name' => 'Desk', 'price' => 100]

$array = Arr::add(['name' => 'Desk', 'price' => null], 'price', 100);

// ['name' => 'Desk', 'price' => 100]
```

`Arr::collapse()` {.collection-method}

The `Arr::collapse` method collapses an array of arrays into a single array:

```
use Illuminate\Support\Arr;

$array = Arr::collapse([[1, 2, 3], [4, 5, 6], [7, 8, 9]]);

// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`Arr::crossJoin()` {.collection-method}

The `Arr::crossJoin` method cross joins the given arrays, returning a Cartesian product with all possible permutations:

```
use Illuminate\Support\Arr;

$matrix = Arr::crossJoin([1, 2], ['a', 'b']);

/*
    [
        [1, 'a'],
        [1, 'b'],
        [2, 'a'],
        [2, 'b'],
    ]
*/

$matrix = Arr::crossJoin([1, 2], ['a', 'b'], ['I', 'II']);

/*
    [
        [1, 'a', 'I'],
        [1, 'a', 'II'],
        [1, 'b', 'I'],
        [1, 'b', 'II'],
        [2, 'a', 'I'],
        [2, 'a', 'II'],
        [2, 'b', 'I'],
        [2, 'b', 'II'],
    ]
*/
```

`Arr::divide()` {.collection-method}

The `Arr::divide` method returns two arrays: one containing the keys and the other containing the values of the given array:

```
use Illuminate\Support\Arr;

[$keys, $values] = Arr::divide(['name' => 'Desk']);

// $keys: ['name']

// $values: ['Desk']
```

`Arr::dot()` {.collection-method}

The `Arr::dot` method flattens a multi-dimensional array into a single level array that uses "dot" notation to indicate depth:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$flattened = Arr::dot($array);

// ['products.desk.price' => 100]
```

`Arr::except()` {.collection-method}

The `Arr::except` method removes the given key / value pairs from an array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$filtered = Arr::except($array, ['price']);

// ['name' => 'Desk']
```

`Arr::exists()` {.collection-method}

The `Arr::exists` method checks that the given key exists in the provided array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'John Doe', 'age' => 17];

$exists = Arr::exists($array, 'name');

// true

$exists = Arr::exists($array, 'salary');

// false
```

`Arr::first()` {.collection-method}

The `Arr::first` method returns the first element of an array passing a given truth test:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300];

$first = Arr::first($array, function ($value, $key) {
    return $value >= 150;
});

// 200
```

A default value may also be passed as the third parameter to the method. This value will be returned if no value passes the truth test:

```
use Illuminate\Support\Arr;

$first = Arr::first($array, $callback, $default);
```

`Arr::flatten()` {.collection-method}

The `Arr::flatten` method flattens a multi-dimensional array into a single level array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Joe', 'languages' => ['PHP', 'Ruby']];

$flattened = Arr::flatten($array);

// ['Joe', 'PHP', 'Ruby']
```

`Arr::forget()` {.collection-method}

The `Arr::forget` method removes a given key / value pair from a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::forget($array, 'products.desk');

// ['products' => []]
```

`Arr::get()` {.collection-method}

The `Arr::get` method retrieves a value from a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

$price = Arr::get($array, 'products.desk.price');

// 100
```

The `Arr::get` method also accepts a default value, which will be returned if the specified key is not present in the array:

```
use Illuminate\Support\Arr;

$discount = Arr::get($array, 'products.desk.discount', 0);

// 0
```

`Arr::has()` {.collection-method}

The `Arr::has` method checks whether a given item or items exists in an array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::has($array, 'product.name');

// true

$contains = Arr::has($array, ['product.price', 'product.discount']);

// false
```

`Arr::hasAny()` {.collection-method}

The `Arr::hasAny` method checks whether any item in a given set exists in an array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['product' => ['name' => 'Desk', 'price' => 100]];

$contains = Arr::hasAny($array, 'product.name');

// true

$contains = Arr::hasAny($array, ['product.name', 'product.discount']);

// true

$contains = Arr::hasAny($array, ['category', 'product.discount']);

// false
```

`Arr::isAssoc()` {.collection-method}

The `Arr::isAssoc` returns `true` if the given array is an associative array. An array is considered "associative" if it doesn't have sequential numerical keys beginning with zero:

```
use Illuminate\Support\Arr;

$isAssoc = Arr::isAssoc(['product' => ['name' => 'Desk', 'price' =>
100]]);

// true

$isAssoc = Arr::isAssoc([1, 2, 3]);

// false
```

`Arr::last()` {.collection-method}

The `Arr::last` method returns the last element of an array passing a given truth test:

```
use Illuminate\Support\Arr;

$array = [100, 200, 300, 110];

$last = Arr::last($array, function ($value, $key) {
    return $value >= 150;
});

// 300
```

A default value may be passed as the third argument to the method. This value will be returned if no value passes the truth test:

```
use Illuminate\Support\Arr;

$last = Arr::last($array, $callback, $default);
```

`Arr::only()` {.collection-method}

The `Arr::only` method returns only the specified key / value pairs from the given array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100, 'orders' => 10];

$slice = Arr::only($array, ['name', 'price']);

// ['name' => 'Desk', 'price' => 100]
```

`Arr::pluck()` {.collection-method}

The `Arr::pluck` method retrieves all of the values for a given key from an array:

```
use Illuminate\Support\Arr;

$array = [
    ['developer' => ['id' => 1, 'name' => 'Taylor']],
    ['developer' => ['id' => 2, 'name' => 'Abigail']],
];

$names = Arr::pluck($array, 'developer.name');

// ['Taylor', 'Abigail']
```

You may also specify how you wish the resulting list to be keyed:

```
use Illuminate\Support\Arr;

$names = Arr::pluck($array, 'developer.name', 'developer.id');

// [1 => 'Taylor', 2 => 'Abigail']
```

`Arr::prepend()` {.collection-method}

The `Arr::prepend` method will push an item onto the beginning of an array:

```
use Illuminate\Support\Arr;

$array = ['one', 'two', 'three', 'four'];

$array = Arr::prepend($array, 'zero');

// ['zero', 'one', 'two', 'three', 'four']
```

If needed, you may specify the key that should be used for the value:

```
use Illuminate\Support\Arr;

$array = ['price' => 100];

$array = Arr::prepend($array, 'Desk', 'name');

// ['name' => 'Desk', 'price' => 100]
```

`Arr::pull()` {.collection-method}

The `Arr::pull` method returns and removes a key / value pair from an array:

```
use Illuminate\Support\Arr;

$array = ['name' => 'Desk', 'price' => 100];

$name = Arr::pull($array, 'name');

// $name: Desk

// $array: ['price' => 100]
```

A default value may be passed as the third argument to the method. This value will be returned if the key doesn't exist:

```
use Illuminate\Support\Arr;

$value = Arr::pull($array, $key, $default);
```

`Arr::query()` {.collection-method}

The `Arr::query` method converts the array into a query string:

```
use Illuminate\Support\Arr;

$array = [
    'name' => 'Taylor',
    'order' => [
        'column' => 'created_at',
        'direction' => 'desc'
    ]
];

Arr::query($array);

// name=Taylor&order[column]=created_at&order[direction]=desc
```

`Arr::random()` {.collection-method}

The `Arr::random` method returns a random value from an array:

```
use Illuminate\Support\Arr;

$array = [1, 2, 3, 4, 5];

$random = Arr::random($array);

// 4 - (retrieved randomly)
```

You may also specify the number of items to return as an optional second argument. Note that providing this argument will return an array even if only one item is desired:

```
use Illuminate\Support\Arr;

$items = Arr::random($array, 2);

// [2, 5] - (retrieved randomly)
```

`Arr::set()` {.collection-method}

The `Arr::set` method sets a value within a deeply nested array using "dot" notation:

```
use Illuminate\Support\Arr;

$array = ['products' => ['desk' => ['price' => 100]]];

Arr::set($array, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

`Arr::shuffle()` {.collection-method}

The `Arr::shuffle` method randomly shuffles the items in the array:

```
use Illuminate\Support\Arr;

$array = Arr::shuffle([1, 2, 3, 4, 5]);

// [3, 2, 5, 1, 4] - (generated randomly)
```

`Arr::sort()` {.collection-method}

The `Arr::sort` method sorts an array by its values:

```
use Illuminate\Support\Arr;

$array = ['Desk', 'Table', 'Chair'];

$sorted = Arr::sort($array);

// ['Chair', 'Desk', 'Table']
```

You may also sort the array by the results of a given closure:

```
use Illuminate\Support\Arr;

$array = [
    ['name' => 'Desk'],
    ['name' => 'Table'],
    ['name' => 'Chair'],
];

$sorted = array_values(Arr::sort($array, function ($value) {
    return $value['name'];
}));

/*
    [
        ['name' => 'Chair'],
        ['name' => 'Desk'],
        ['name' => 'Table'],
    ]
*/
```

`Arr::sortRecursive()` {.collection-method}

The `Arr::sortRecursive` method recursively sorts an array using the `sort` function for numerically indexed sub-arrays and the `ksort` function for associative sub-arrays:

```
use Illuminate\Support\Arr;

$array = [
    ['Roman', 'Taylor', 'Li'],
    ['PHP', 'Ruby', 'JavaScript'],
    ['one' => 1, 'two' => 2, 'three' => 3],
];

$sorted = Arr::sortRecursive($array);

/*
    [
        ['JavaScript', 'PHP', 'Ruby'],
        ['one' => 1, 'three' => 3, 'two' => 2],
        ['Li', 'Roman', 'Taylor'],
    ]
*/
```

`Arr::toCssClasses()` {.collection-method}

The `Arr::toCssClasses` conditionally compiles a CSS class string. The method accepts an array of classes where the array key contains the class or classes you wish to add, while the value is a boolean expression. If the array element has a numeric key, it will always be included in the rendered class list:

```
use Illuminate\Support\Arr;

$isActive = false;
$hasError = true;

$array = ['p-4', 'font-bold' => $isActive, 'bg-red' => $hasError];

$classes = Arr::toCssClasses($array);

/*
    'p-4 bg-red'
*/
```

This method powers Laravel's functionality allowing merging classes with a Blade component's attribute bag as well as the `@class` Blade directive.

`Arr::undot()` {.collection-method}

The `Arr::undot` method expands a single-dimensional array that uses "dot" notation into a multi-dimensional array:

```
use Illuminate\Support\Arr;

$array = [
    'user.name' => 'Kevin Malone',
    'user.occupation' => 'Accountant',
];

$array = Arr::undot($array);

// ['user' => ['name' => 'Kevin Malone', 'occupation' => 'Accountant']]
```

`Arr::where()` {.collection-method}

The `Arr::where` method filters an array using the given closure:

```
use Illuminate\Support\Arr;

$array = [100, '200', 300, '400', 500];

$filtered = Arr::where($array, function ($value, $key) {
    return is_string($value);
});

// [1 => '200', 3 => '400']
```

`Arr::whereNotNull()` {.collection-method}

The `Arr::whereNotNull` method removes all `null` values from the given array:

```
use Illuminate\Support\Arr;

$array = [0, null];

$filtered = Arr::whereNotNull($array);

// [0 => 0]
```

`Arr::wrap()` {.collection-method}

The `Arr::wrap` method wraps the given value in an array. If the given value is already an array it will be returned without modification:

```
use Illuminate\Support\Arr;

$string = 'Laravel';

$array = Arr::wrap($string);

// ['Laravel']
```

If the given value is `null`, an empty array will be returned:

```
use Illuminate\Support\Arr;

$array = Arr::wrap(null);

// []
```

`data_fill()` {.collection-method}

The `data_fill` function sets a missing value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_fill($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 100]]]

data_fill($data, 'products.desk.discount', 10);

// ['products' => ['desk' => ['price' => 100, 'discount' => 10]]]
```

This function also accepts asterisks as wildcards and will fill the target accordingly:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2'],
    ],
];

data_fill($data, 'products.*.price', 200);

/*
    [
        'products' => [
            ['name' => 'Desk 1', 'price' => 100],
            ['name' => 'Desk 2', 'price' => 200],
        ],
    ]
*/
```

`data_get()` {.collection-method}

The `data_get` function retrieves a value from a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

$price = data_get($data, 'products.desk.price');

// 100
```

The `data_get` function also accepts a default value, which will be returned if the specified key is not found:

```
$discount = data_get($data, 'products.desk.discount', 0);

// 0
```

The function also accepts wildcards using asterisks, which may target any key of the array or object:

```
$data = [
    'product-one' => ['name' => 'Desk 1', 'price' => 100],
    'product-two' => ['name' => 'Desk 2', 'price' => 150],
];

data_get($data, '*.name');

// ['Desk 1', 'Desk 2'];
```

`data_set()` {.collection-method}

The `data_set` function sets a value within a nested array or object using "dot" notation:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200);

// ['products' => ['desk' => ['price' => 200]]]
```

This function also accepts wildcards using asterisks and will set values on the target accordingly:

```
$data = [
    'products' => [
        ['name' => 'Desk 1', 'price' => 100],
        ['name' => 'Desk 2', 'price' => 150],
    ],
];

data_set($data, 'products.*.price', 200);

/*
    [
        'products' => [
            ['name' => 'Desk 1', 'price' => 200],
            ['name' => 'Desk 2', 'price' => 200],
        ],
    ]
*/
```

By default, any existing values are overwritten. If you wish to only set a value if it doesn't exist, you may pass `false` as the fourth argument to the function:

```
$data = ['products' => ['desk' => ['price' => 100]]];

data_set($data, 'products.desk.price', 200, $overwrite = false);

// ['products' => ['desk' => ['price' => 100]]]
```

`head()` {.collection-method}

The head function returns the first element in the given array:

```
$array = [100, 200, 300];

$first = head($array);

// 100
```

`last()` {.collection-method}

The `last` function returns the last element in the given array:

```
$array = [100, 200, 300];

$last = last($array);

// 300
```

Paths

`app_path()` {.collection-method}

The `app_path` function returns the fully qualified path to your application's `app` directory. You may also use the `app_path` function to generate a fully qualified path to a file relative to the application directory:

```
$path = app_path();

$path = app_path('Http/Controllers/Controller.php');
```

`base_path()` {.collection-method}

The `base_path` function returns the fully qualified path to your application's root directory. You may also use the `base_path` function to generate a fully qualified path to a given file relative to the project root directory:

```
$path = base_path();

$path = base_path('vendor/bin');
```

`config_path()` {.collection-method}

The `config_path` function returns the fully qualified path to your application's `config` directory. You may also use the `config_path` function to generate a fully qualified path to a given file within the application's configuration directory:

```
$path = config_path();

$path = config_path('app.php');
```

`database_path()` {.collection-method}

The `database_path` function returns the fully qualified path to your application's `database` directory. You may also use the `database_path` function to generate a fully qualified path to a given file within the database directory:

```
$path = database_path();

$path = database_path('factories/UserFactory.php');
```

`mix()` {.collection-method}

The `mix` function returns the path to a versioned Mix file:

```
$path = mix('css/app.css');
```

`public_path()` {.collection-method}

The `public_path` function returns the fully qualified path to your application's `public` directory. You may also use the `public_path` function to generate a fully qualified path to a given file within the public directory:

```
$path = public_path();

$path = public_path('css/app.css');
```

`resource_path()` {.collection-method}

The `resource_path` function returns the fully qualified path to your application's `resources` directory. You may also use the `resource_path` function to generate a fully qualified path to a given file within the resources directory:

```
$path = resource_path();

$path = resource_path('sass/app.scss');
```

`storage_path()` {.collection-method}

The `storage_path` function returns the fully qualified path to your application's `storage` directory. You may also use the `storage_path` function to generate a fully qualified path to a given file within the storage directory:

```
$path = storage_path();

$path = storage_path('app/file.txt');
```

Strings

`__()` {.collection-method}

The `__` function translates the given translation string or translation key using your localization files:

```
echo __('Welcome to our application');

echo __('messages.welcome');
```

If the specified translation string or key does not exist, the `__` function will return the given value. So, using the example above, the `__` function would return `messages.welcome` if that translation key does not exist.

`class_basename()` {.collection-method}

The `class_basename` function returns the class name of the given class with the class's namespace removed:

```
$class = class_basename('Foo\Bar\Baz');

// Baz
```

`e()` {.collection-method}

The `e` function runs PHP's `htmlspecialchars` function with the `double_encode` option set to `true` by default:

```
echo e('<html>foo</html>');

// &lt;html&gt;foo&lt;/html&gt;
```

`preg_replace_array()` {.collection-method}

The `preg_replace_array` function replaces a given pattern in the string sequentially using an array:

```
$string = 'The event will take place between :start and :end';

$replaced = preg_replace_array('/:[a-z_]+/', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

`Str::after()` {.collection-method}

The `Str::after` method returns everything after the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::after('This is my name', 'This is');

// ' my name'
```

`Str::afterLast()` {.collection-method}

The `Str::afterLast` method returns everything after the last occurrence of the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::afterLast('App\Http\Controllers\Controller', '\\');

// 'Controller'
```

`Str::ascii()` {.collection-method}

The `Str::ascii` method will attempt to transliterate the string into an ASCII value:

```
use Illuminate\Support\Str;

$slice = Str::ascii('û');

// 'u'
```

`Str::before()` {.collection-method}

The `Str::before` method returns everything before the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::before('This is my name', 'my name');

// 'This is '
```

`Str::beforeLast()` {.collection-method}

The `Str::beforeLast` method returns everything before the last occurrence of the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::beforeLast('This is my name', 'is');

// 'This '
```

`Str::between()` {.collection-method}

The `Str::between` method returns the portion of a string between two values:

```
use Illuminate\Support\Str;

$slice = Str::between('This is my name', 'This', 'name');

// ' is my '
```

`Str::camel()` {.collection-method}

The `Str::camel` method converts the given string to `camelCase`:

```
use Illuminate\Support\Str;

$converted = Str::camel('foo_bar');

// fooBar
```

`Str::contains()` {.collection-method}

The `Str::contains` method determines if the given string contains the given value. This method is case sensitive:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', 'my');

// true
```

You may also pass an array of values to determine if the given string contains any of the values in the array:

```
use Illuminate\Support\Str;

$contains = Str::contains('This is my name', ['my', 'foo']);

// true
```

`Str::containsAll()` {.collection-method}

The `Str::containsAll` method determines if the given string contains all of the values in a given array:

```
use Illuminate\Support\Str;

$containsAll = Str::containsAll('This is my name', ['my', 'name']);

// true
```

`Str::endsWith()` {.collection-method}

The `Str::endsWith` method determines if the given string ends with the given value:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', 'name');

// true
```

You may also pass an array of values to determine if the given string ends with any of the values in the array:

```
use Illuminate\Support\Str;

$result = Str::endsWith('This is my name', ['name', 'foo']);

// true

$result = Str::endsWith('This is my name', ['this', 'foo']);

// false
```

`Str::finish()` {.collection-method}

The `Str::finish` method adds a single instance of the given value to a string if it does not already end with that value:

```
use Illuminate\Support\Str;

$adjusted = Str::finish('this/string', '/');

// this/string/

$adjusted = Str::finish('this/string/', '/');

// this/string/
```

`Str::headline()` {.collection-method}

The `Str::headline` method will convert strings delimited by casing, hyphens, or underscores into a space delimited string with each word's first letter capitalized:

```
use Illuminate\Support\Str;

$headline = Str::headline('steve_jobs');

// Steve Jobs

$headline = Str::headline('EmailNotificationSent');

// Email Notification Sent
```

`Str::is()` {.collection-method}

The `Str::is` method determines if a given string matches a given pattern. Asterisks may be used as wildcard values:

```
use Illuminate\Support\Str;

$matches = Str::is('foo*', 'foobar');

// true

$matches = Str::is('baz*', 'foobar');

// false
```

`Str::isAscii()` {.collection-method}

The `Str::isAscii` method determines if a given string is 7 bit ASCII:

```
use Illuminate\Support\Str;

$isAscii = Str::isAscii('Taylor');

// true

$isAscii = Str::isAscii('ü');

// false
```

`Str::isUuid()` {.collection-method}

The `Str::isUuid` method determines if the given string is a valid UUID:

```
use Illuminate\Support\Str;

$isUuid = Str::isUuid('a0a2a2d2-0b87-4a18-83f2-2529882be2de');

// true

$isUuid = Str::isUuid('laravel');

// false
```

`Str::kebab()` {.collection-method}

The `Str::kebab` method converts the given string to `kebab-case`:

```
use Illuminate\Support\Str;

$converted = Str::kebab('fooBar');

// foo-bar
```

`Str::length()` {.collection-method}

The `Str::length` method returns the length of the given string:

```
use Illuminate\Support\Str;

$length = Str::length('Laravel');

// 7
```

`Str::limit()` {.collection-method}

The `Str::limit` method truncates the given string to the specified length:

```
use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog',
20);

// The quick brown fox...
```

You may pass a third argument to the method to change the string that will be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

$truncated = Str::limit('The quick brown fox jumps over the lazy dog',
20, ' (...)');

// The quick brown fox (...)
```

`Str::lower()` {.collection-method}

The `Str::lower` method converts the given string to lowercase:

```
use Illuminate\Support\Str;

$converted = Str::lower('LARAVEL');

// laravel
```

`Str::markdown()` {.collection-method}

The `Str::markdown` method converts GitHub flavored Markdown into HTML:

```
use Illuminate\Support\Str;

$html = Str::markdown('# Laravel');

// <h1>Laravel</h1>

$html = Str::markdown('# Taylor <b>Otwell</b>', [
    'html_input' => 'strip',
]);

// <h1>Taylor Otwell</h1>
```

`Str::mask()` {.collection-method}

The `Str::mask` method masks a portion of a string with a repeated character, and may be used to obfuscate segments of strings such as email addresses and phone numbers:

```
use Illuminate\Support\Str;

$string = Str::mask('taylor@example.com', '*', 3);

// tay**************
```

If needed, you provide a negative number as the third argument to the `mask` method, which will instruct the method to begin masking at the given distance from the end of the string:

```
$string = Str::mask('taylor@example.com', '*', -15, 3);

// tay***@example.com
```

`Str::orderedUuid()` {.collection-method}

The `Str::orderedUuid` method generates a "timestamp first" UUID that may be efficiently stored in an indexed database column. Each UUID that is generated using this method will be sorted after UUIDs previously generated using the method:

```
use Illuminate\Support\Str;

return (string) Str::orderedUuid();
```

`Str::padBoth()` {.collection-method}

The `Str::padBoth` method wraps PHP's `str_pad` function, padding both sides of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padBoth('James', 10, '_');

// '__James___'

$padded = Str::padBoth('James', 10);

// '  James   '
```

`Str::padLeft()` {.collection-method}

The `Str::padLeft` method wraps PHP's `str_pad` function, padding the left side of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padLeft('James', 10, '-=');

// '-=-=-James'

$padded = Str::padLeft('James', 10);

// '     James'
```

`Str::padRight()` {.collection-method}

The `Str::padRight` method wraps PHP's `str_pad` function, padding the right side of a string with another string until the final string reaches a desired length:

```
use Illuminate\Support\Str;

$padded = Str::padRight('James', 10, '-');

// 'James-----'

$padded = Str::padRight('James', 10);

// 'James     '
```

`Str::plural()` {.collection-method}

The `Str::plural` method converts a singular word string to its plural form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$plural = Str::plural('car');

// cars

$plural = Str::plural('child');

// children
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
use Illuminate\Support\Str;

$plural = Str::plural('child', 2);

// children

$singular = Str::plural('child', 1);

// child
```

`Str::pluralStudly()` {.collection-method}

The `Str::pluralStudly` method converts a singular word string formatted in studly caps case to its plural form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$plural = Str::pluralStudly('VerifiedHuman');

// VerifiedHumans

$plural = Str::pluralStudly('UserFeedback');

// UserFeedback
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
use Illuminate\Support\Str;

$plural = Str::pluralStudly('VerifiedHuman', 2);

// VerifiedHumans

$singular = Str::pluralStudly('VerifiedHuman', 1);

// VerifiedHuman
```

`Str::random()` {.collection-method}

The `Str::random` method generates a random string of the specified length. This function uses PHP's `random_bytes` function:

```
use Illuminate\Support\Str;

$random = Str::random(40);
```

`Str::remove()` {.collection-method}

The `Str::remove` method removes the given value or array of values from the string:

```
use Illuminate\Support\Str;

$string = 'Peter Piper picked a peck of pickled peppers.';

$removed = Str::remove('e', $string);

// Ptr Pipr pickd a pck of pickld ppprs.
```

You may also pass `false` as a third argument to the `remove` method to ignore case when removing strings.

`Str::replace()` {.collection-method}

The `Str::replace` method replaces a given string within the string:

```
use Illuminate\Support\Str;

$string = 'Laravel 8.x';

$replaced = Str::replace('8.x', '9.x', $string);

// Laravel 9.x
```

`Str::replaceArray()` {.collection-method}

The `Str::replaceArray` method replaces a given value in the string sequentially using an array:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::replaceArray('?', ['8:30', '9:00'], $string);

// The event will take place between 8:30 and 9:00
```

`Str::replaceFirst()` {.collection-method}

The `Str::replaceFirst` method replaces the first occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceFirst('the', 'a', 'the quick brown fox jumps
over the lazy dog');

// a quick brown fox jumps over the lazy dog
```

`Str::replaceLast()` {.collection-method}

The `Str::replaceLast` method replaces the last occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::replaceLast('the', 'a', 'the quick brown fox jumps over
the lazy dog');

// the quick brown fox jumps over a lazy dog
```

`Str::reverse()` {.collection-method}

The `Str::reverse` method reverses the given string:

```
use Illuminate\Support\Str;

$reversed = Str::reverse('Hello World');

// dlroW olleH
```

`Str::singular()` {.collection-method}

The `Str::singular` method converts a string to its singular form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$singular = Str::singular('cars');

// car

$singular = Str::singular('children');

// child
```

`Str::slug()` {.collection-method}

The `Str::slug` method generates a URL friendly "slug" from the given string:

```
use Illuminate\Support\Str;

$slug = Str::slug('Laravel 5 Framework', '-');

// laravel-5-framework
```

`Str::snake()` {.collection-method}

The `Str::snake` method converts the given string to `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::snake('fooBar');

// foo_bar

$converted = Str::snake('fooBar', '-');

// foo-bar
```

`Str::start()` {.collection-method}

The `Str::start` method adds a single instance of the given value to a string if it does not already start with that value:

```
use Illuminate\Support\Str;

$adjusted = Str::start('this/string', '/');

// /this/string

$adjusted = Str::start('/this/string', '/');

// /this/string
```

`Str::startsWith()` {.collection-method}

The `Str::startsWith` method determines if the given string begins with the given value:

```
use Illuminate\Support\Str;

$result = Str::startsWith('This is my name', 'This');

// true
```

If an array of possible values is passed, the `startsWith` method will return `true` if the string begins with any of the given values:

```
$result = Str::startsWith('This is my name', ['This', 'That', 'There']);

// true
```

`Str::studly()` {.collection-method}

The `Str::studly` method converts the given string to `StudlyCase`:

```
use Illuminate\Support\Str;

$converted = Str::studly('foo_bar');

// FooBar
```

`Str::substr()` {.collection-method}

The `Str::substr` method returns the portion of string specified by the start and length parameters:

```
use Illuminate\Support\Str;

$converted = Str::substr('The Laravel Framework', 4, 7);

// Laravel
```

`Str::substrCount()` {.collection-method}

The `Str::substrCount` method returns the number of occurrences of a given value in the given string:

```
use Illuminate\Support\Str;

$count = Str::substrCount('If you like ice cream, you will like snow
cones.', 'like');

// 2
```

`Str::title()` {.collection-method}

The `Str::title` method converts the given string to `Title Case`:

```
use Illuminate\Support\Str;

$converted = Str::title('a nice title uses the correct case');

// A Nice Title Uses The Correct Case
```

`Str::ucfirst()` {.collection-method}

The `Str::ucfirst` method returns the given string with the first character capitalized:

```
use Illuminate\Support\Str;

$string = Str::ucfirst('foo bar');

// Foo bar
```

`Str::upper()` {.collection-method}

The `Str::upper` method converts the given string to uppercase:

```
use Illuminate\Support\Str;

$string = Str::upper('laravel');

// LARAVEL
```

`Str::uuid()` {.collection-method}

The `Str::uuid` method generates a UUID (version 4):

```
use Illuminate\Support\Str;

return (string) Str::uuid();
```

`Str::wordCount()` {.collection-method}

The `Str::wordCount` method returns the number of words that a string contains:

```
use Illuminate\Support\Str;

Str::wordCount('Hello, world!'); // 2
```

`Str::words()` {.collection-method}

The `Str::words` method limits the number of words in a string. An additional string may be passed to this method via its third argument to specify which string should be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

return Str::words('Perfectly balanced, as all things should be.', 3, '
>>>');

// Perfectly balanced, as >>>
```

`trans()` {.collection-method}

The `trans` function translates the given translation key using your localization files:

```
echo trans('messages.welcome');
```

If the specified translation key does not exist, the `trans` function will return the given key. So, using the example above, the `trans` function would return `messages.welcome` if the translation key does not exist.

`trans_choice()` {.collection-method}

The `trans_choice` function translates the given translation key with inflection:

```
echo trans_choice('messages.notifications', $unreadCount);
```

If the specified translation key does not exist, the `trans_choice` function will return the given key. So, using the example above, the `trans_choice` function would return `messages.notifications` if the translation key does not exist.

Fluent Strings

Fluent strings provide a more fluent, object-oriented interface for working with string values, allowing you to chain multiple string operations together using a more readable syntax compared to traditional string operations.

`after` {.collection-method}

The `after` method returns everything after the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->after('This is');

// ' my name'
```

`afterLast` {.collection-method}

The `afterLast` method returns everything after the last occurrence of the given value in a string. The entire string will be returned if the value does not exist within the string:

```
use Illuminate\Support\Str;

$slice = Str::of('App\Http\Controllers\Controller')->afterLast('\\');

// 'Controller'
```

`append` {.collection-method}

The `append` method appends the given values to the string:

```
use Illuminate\Support\Str;

$string = Str::of('Taylor')->append(' Otwell');

// 'Taylor Otwell'
```

`ascii` {.collection-method}

The `ascii` method will attempt to transliterate the string into an ASCII value:

```
use Illuminate\Support\Str;

$string = Str::of('ü')->ascii();

// 'u'
```

`basename` {.collection-method}

The `basename` method will return the trailing name component of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->basename();

// 'baz'
```

If needed, you may provide an "extension" that will be removed from the trailing component:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz.jpg')->basename('.jpg');

// 'baz'
```

`before` {.collection-method}

The `before` method returns everything before the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->before('my name');

// 'This is '
```

`beforeLast` {.collection-method}

The `beforeLast` method returns everything before the last occurrence of the given value in a string:

```
use Illuminate\Support\Str;

$slice = Str::of('This is my name')->beforeLast('is');

// 'This '
```

`camel` {.collection-method}

The `camel` method converts the given string to `camelCase`:

```
use Illuminate\Support\Str;

$converted = Str::of('foo_bar')->camel();

// fooBar
```

`contains` {.collection-method}

The `contains` method determines if the given string contains the given value. This method is case sensitive:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains('my');

// true
```

You may also pass an array of values to determine if the given string contains any of the values in the array:

```
use Illuminate\Support\Str;

$contains = Str::of('This is my name')->contains(['my', 'foo']);

// true
```

`containsAll` {.collection-method}

The `containsAll` method determines if the given string contains all of the values in the given array:

```
use Illuminate\Support\Str;

$containsAll = Str::of('This is my name')->containsAll(['my', 'name']);

// true
```

`dirname` {.collection-method}

The `dirname` method returns the parent directory portion of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->dirname();

// '/foo/bar'
```

If necessary, you may specify how many directory levels you wish to trim from the string:

```
use Illuminate\Support\Str;

$string = Str::of('/foo/bar/baz')->dirname(2);

// '/foo'
```

endsWith {.collection-method}

The endsWith method determines if the given string ends with the given value:

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->endsWith('name');

// true
```

You may also pass an array of values to determine if the given string ends with any of the values in the array:

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->endsWith(['name', 'foo']);

// true

$result = Str::of('This is my name')->endsWith(['this', 'foo']);

// false
```

exactly {.collection-method}

The exactly method determines if the given string is an exact match with another string:

```
use Illuminate\Support\Str;

$result = Str::of('Laravel')->exactly('Laravel');

// true
```

`explode` {.collection-method}

The `explode` method splits the string by the given delimiter and returns a collection containing each section of the split string:

```
use Illuminate\Support\Str;

$collection = Str::of('foo bar baz')->explode(' ');

// collect(['foo', 'bar', 'baz'])
```

`finish` {.collection-method}

The `finish` method adds a single instance of the given value to a string if it does not already end with that value:

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->finish('/');

// this/string/

$adjusted = Str::of('this/string/')->finish('/');

// this/string/
```

`is` {.collection-method}

The `is` method determines if a given string matches a given pattern. Asterisks may be used as wildcard values

```
use Illuminate\Support\Str;

$matches = Str::of('foobar')->is('foo*');

// true

$matches = Str::of('foobar')->is('baz*');

// false
```

`isAscii` {.collection-method}

The `isAscii` method determines if a given string is an ASCII string:

```
use Illuminate\Support\Str;

$result = Str::of('Taylor')->isAscii();

// true

$result = Str::of('ü')->isAscii();

// false
```

`isEmpty` {.collection-method}

The `isEmpty` method determines if the given string is empty:

```
use Illuminate\Support\Str;

$result = Str::of('  ')->trim()->isEmpty();

// true

$result = Str::of('Laravel')->trim()->isEmpty();

// false
```

`isNotEmpty` {.collection-method}

The `isNotEmpty` method determines if the given string is not empty:

```
use Illuminate\Support\Str;

$result = Str::of('   ')->trim()->isNotEmpty();

// false

$result = Str::of('Laravel')->trim()->isNotEmpty();

// true
```

`isUuid` {.collection-method}

The `isUuid` method determines if a given string is a UUID:

```
use Illuminate\Support\Str;

$result = Str::of('5ace9ab9-e9cf-4ec6-a19d-5881212a452c')->isUuid();

// true

$result = Str::of('Taylor')->isUuid();

// false
```

`kebab` {.collection-method}

The `kebab` method converts the given string to `kebab-case`:

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->kebab();

// foo-bar
```

`length` {.collection-method}

The `length` method returns the length of the given string:

```
use Illuminate\Support\Str;

$length = Str::of('Laravel')->length();

// 7
```

`limit` {.collection-method}

The `limit` method truncates the given string to the specified length:

```
use Illuminate\Support\Str;

$truncated = Str::of('The quick brown fox jumps over the lazy
dog')->limit(20);

// The quick brown fox...
```

You may also pass a second argument to change the string that will be appended to the end of the truncated string:

```
use Illuminate\Support\Str;

$truncated = Str::of('The quick brown fox jumps over the lazy
dog')->limit(20, ' (...)');

// The quick brown fox (...)
```

`lower` {.collection-method}

The `lower` method converts the given string to lowercase:

```
use Illuminate\Support\Str;

$result = Str::of('LARAVEL')->lower();

// 'laravel'
```

`ltrim` {.collection-method}

The `ltrim` method trims the left side of the string:

```
use Illuminate\Support\Str;

$string = Str::of('  Laravel  ')->ltrim();

// 'Laravel  '

$string = Str::of('/Laravel/')->ltrim('/');

// 'Laravel/'
```

`markdown` {.collection-method}

The `markdown` method converts GitHub flavored Markdown into HTML:

```
use Illuminate\Support\Str;

$html = Str::of('# Laravel')->markdown();

// <h1>Laravel</h1>

$html = Str::of('# Taylor <b>Otwell</b>')->markdown([
    'html_input' => 'strip',
]);

// <h1>Taylor Otwell</h1>
```

`mask` {.collection-method}

The `mask` method masks a portion of a string with a repeated character, and may be used to obfuscate segments of strings such as email addresses and phone numbers:

```
use Illuminate\Support\Str;

$string = Str::of('taylor@example.com')->mask('*', 3);

// tay***************
```

If needed, you provide a negative number as the third argument to the `mask` method, which will instruct the method to begin masking at the given distance from the end of the string:

```
$string = Str::of('taylor@example.com')->mask('*', -15, 3);

// tay***@example.com
```

`match` {.collection-method}

The `match` method will return the portion of a string that matches a given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('foo bar')->match('/bar/');

// 'bar'

$result = Str::of('foo bar')->match('/foo (.*)/');

// 'bar'
```

`matchAll` {.collection-method}

The `matchAll` method will return a collection containing the portions of a string that match a given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('bar foo bar')->matchAll('/bar/');

// collect(['bar', 'bar'])
```

If you specify a matching group within the expression, Laravel will return a collection of that group's matches:

```
use Illuminate\Support\Str;

$result = Str::of('bar fun bar fly')->matchAll('/f(\w*)/');

// collect(['un', 'ly']);
```

If no matches are found, an empty collection will be returned.

`padBoth` {.collection-method}

The `padBoth` method wraps PHP's `str_pad` function, padding both sides of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padBoth(10, '_');

// '__James___'

$padded = Str::of('James')->padBoth(10);

// '  James   '
```

`padLeft` {.collection-method}

The `padLeft` method wraps PHP's `str_pad` function, padding the left side of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padLeft(10, '-=');

// '-=-=-James'

$padded = Str::of('James')->padLeft(10);

// '     James'
```

`padRight` {.collection-method}

The `padRight` method wraps PHP's `str_pad` function, padding the right side of a string with another string until the final string reaches the desired length:

```
use Illuminate\Support\Str;

$padded = Str::of('James')->padRight(10, '-');

// 'James-----'

$padded = Str::of('James')->padRight(10);

// 'James     '
```

`pipe` {.collection-method}

The `pipe` method allows you to transform the string by passing its current value to the given callable:

```
use Illuminate\Support\Str;

$hash = Str::of('Laravel')->pipe('md5')->prepend('Checksum: ');

// 'Checksum: a5c95b86291ea299fcbe64458ed12702'

$closure = Str::of('foo')->pipe(function ($str) {
    return 'bar';
});

// 'bar'
```

`plural` {.collection-method}

The `plural` method converts a singular word string to its plural form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$plural = Str::of('car')->plural();

// cars

$plural = Str::of('child')->plural();

// children
```

You may provide an integer as a second argument to the function to retrieve the singular or plural form of the string:

```
use Illuminate\Support\Str;

$plural = Str::of('child')->plural(2);

// children

$plural = Str::of('child')->plural(1);

// child
```

`prepend` {.collection-method}

The `prepend` method prepends the given values onto the string:

```
use Illuminate\Support\Str;

$string = Str::of('Framework')->prepend('Laravel ');

// Laravel Framework
```

`remove` {.collection-method}

The `remove` method removes the given value or array of values from the string:

```
use Illuminate\Support\Str;

$string = Str::of('Arkansas is quite beautiful!')->remove('quite');

// Arkansas is beautiful!
```

You may also pass `false` as a second parameter to ignore case when removing.

`replace` {.collection-method}

The `replace` method replaces a given string within the string:

```
use Illuminate\Support\Str;

$replaced = Str::of('Laravel 6.x')->replace('6.x', '7.x');

// Laravel 7.x
```

`replaceArray` {.collection-method}

The `replaceArray` method replaces a given value in the string sequentially using an array:

```
use Illuminate\Support\Str;

$string = 'The event will take place between ? and ?';

$replaced = Str::of($string)->replaceArray('?', ['8:30', '9:00']);

// The event will take place between 8:30 and 9:00
```

`replaceFirst` {.collection-method}

The `replaceFirst` method replaces the first occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy
dog')->replaceFirst('the', 'a');

// a quick brown fox jumps over the lazy dog
```

`replaceLast` {.collection-method}

The `replaceLast` method replaces the last occurrence of a given value in a string:

```
use Illuminate\Support\Str;

$replaced = Str::of('the quick brown fox jumps over the lazy
dog')->replaceLast('the', 'a');

// the quick brown fox jumps over a lazy dog
```

`replaceMatches` {.collection-method}

The `replaceMatches` method replaces all portions of a string matching a pattern with the given replacement string:

```
use Illuminate\Support\Str;

$replaced = Str::of('(+1) 501-555-1000')->replaceMatches('/[^A-Za-
z0-9]++/', '')

// '15015551000'
```

The `replaceMatches` method also accepts a closure that will be invoked with each portion of the string matching the given pattern, allowing you to perform the replacement logic within the closure and return the replaced value:

```
use Illuminate\Support\Str;

$replaced = Str::of('123')->replaceMatches('/\d/', function ($match) {
    return '['.$match[0].']';
});

// '[1][2][3]'
```

`rtrim` {.collection-method}

The `rtrim` method trims the right side of the given string:

```
use Illuminate\Support\Str;

$string = Str::of('  Laravel  ')->rtrim();

// '  Laravel'

$string = Str::of('/Laravel/')->rtrim('/');

// '/Laravel'
```

`singular` {.collection-method}

The `singular` method converts a string to its singular form. This function currently only supports the English language:

```
use Illuminate\Support\Str;

$singular = Str::of('cars')->singular();

// car

$singular = Str::of('children')->singular();

// child
```

`slug` {.collection-method}

The `slug` method generates a URL friendly "slug" from the given string:

```
use Illuminate\Support\Str;

$slug = Str::of('Laravel Framework')->slug('-');

// laravel-framework
```

`snake` {.collection-method}

The `snake` method converts the given string to `snake_case`:

```
use Illuminate\Support\Str;

$converted = Str::of('fooBar')->snake();

// foo_bar
```

`split` {.collection-method}

The `split` method splits a string into a collection using a regular expression:

```
use Illuminate\Support\Str;

$segments = Str::of('one, two, three')->split('/[\s,]+/');

// collect(["one", "two", "three"])
```

`start` {.collection-method}

The `start` method adds a single instance of the given value to a string if it does not already start with that value:

```
use Illuminate\Support\Str;

$adjusted = Str::of('this/string')->start('/');

// /this/string

$adjusted = Str::of('/this/string')->start('/');

// /this/string
```

`startsWith` {.collection-method}

The `startsWith` method determines if the given string begins with the given value:

```
use Illuminate\Support\Str;

$result = Str::of('This is my name')->startsWith('This');

// true
```

`studly` {.collection-method}

The `studly` method converts the given string to `StudlyCase`:

```
use Illuminate\Support\Str;

$converted = Str::of('foo_bar')->studly();

// FooBar
```

`substr` {.collection-method}

The `substr` method returns the portion of the string specified by the given start and length parameters:

```
use Illuminate\Support\Str;

$string = Str::of('Laravel Framework')->substr(8);

// Framework

$string = Str::of('Laravel Framework')->substr(8, 5);

// Frame
```

`tap` {.collection-method}

The `tap` method passes the string to the given closure, allowing you to examine and interact with the string while not affecting the string itself. The original string is returned by the `tap` method regardless of what is returned by the closure:

```
use Illuminate\Support\Str;

$string = Str::of('Laravel')
    ->append(' Framework')
    ->tap(function ($string) {
        dump('String after append: ' . $string);
    })
    ->upper();

// LARAVEL FRAMEWORK
```

`test` {.collection-method}

The `test` method determines if a string matches the given regular expression pattern:

```
use Illuminate\Support\Str;

$result = Str::of('Laravel Framework')->test('/Laravel/');

// true
```

`title` {.collection-method}

The `title` method converts the given string to `Title Case`:

```
use Illuminate\Support\Str;

$converted = Str::of('a nice title uses the correct case')->title();

// A Nice Title Uses The Correct Case
```

`trim` {.collection-method}

The `trim` method trims the given string:

```
use Illuminate\Support\Str;

$string = Str::of('  Laravel  ')->trim();

// 'Laravel'

$string = Str::of('/Laravel/')->trim('/');

// 'Laravel'
```

`ucfirst` {.collection-method}

The `ucfirst` method returns the given string with the first character capitalized:

```
use Illuminate\Support\Str;

$string = Str::of('foo bar')->ucfirst();

// Foo bar
```

`upper` {.collection-method}

The `upper` method converts the given string to uppercase:

```
use Illuminate\Support\Str;

$adjusted = Str::of('laravel')->upper();

// LARAVEL
```

`when` {.collection-method}

The `when` method invokes the given closure if a given condition is `true`. The closure will receive the fluent string instance:

```
use Illuminate\Support\Str;

$string = Str::of('Taylor')
                ->when(true, function ($string) {
                    return $string->append(' Otwell');
                });

// 'Taylor Otwell'
```

If necessary, you may pass another closure as the third parameter to the `when` method. This closure will execute if the condition parameter evaluates to `false`.

`whenEmpty` {.collection-method}

The `whenEmpty` method invokes the given closure if the string is empty. If the closure returns a value, that value will also be returned by the `whenEmpty` method. If the closure does not return a value, the fluent string instance will be returned:

```
use Illuminate\Support\Str;

$string = Str::of('  ')->whenEmpty(function ($string) {
    return $string->trim()->prepend('Laravel');
});

// 'Laravel'
```

`wordCount` {.collection-method}

The `wordCount` method returns the number of words that a string contains:

```
use Illuminate\Support\Str;

Str::of('Hello, world!')->wordCount(); // 2
```

`words` {.collection-method}

The `words` method limits the number of words in a string. If necessary, you may specify an additional string that will be appended to the truncated string:

```
use Illuminate\Support\Str;

$string = Str::of('Perfectly balanced, as all things should
be.')->words(3, ' >>>');

// Perfectly balanced, as >>>
```

URLs

`action()` {.collection-method}

The `action` function generates a URL for the given controller action:

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

If the method accepts route parameters, you may pass them as the second argument to the method:

```
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

`asset()` {.collection-method}

The `asset` function generates a URL for an asset using the current scheme of the request (HTTP or HTTPS):

```
$url = asset('img/photo.jpg');
```

You can configure the asset URL host by setting the `ASSET_URL` variable in your `.env` file. This can be useful if you host your assets on an external service like Amazon S3 or another CDN:

```
// ASSET_URL=http://example.com/assets

$url = asset('img/photo.jpg'); //
http://example.com/assets/img/photo.jpg
```

`route()` {.collection-method}

The `route` function generates a URL for a given named route:

```
$url = route('route.name');
```

If the route accepts parameters, you may pass them as the second argument to the function:

```
$url = route('route.name', ['id' => 1]);
```

By default, the `route` function generates an absolute URL. If you wish to generate a relative URL, you may pass `false` as the third argument to the function:

```
$url = route('route.name', ['id' => 1], false);
```

`secure_asset()` {.collection-method}

The `secure_asset` function generates a URL for an asset using HTTPS:

```
$url = secure_asset('img/photo.jpg');
```

`secure_url()` {.collection-method}

The `secure_url` function generates a fully qualified HTTPS URL to the given path. Additional URL segments may be passed in the function's second argument:

```
$url = secure_url('user/profile');

$url = secure_url('user/profile', [1]);
```

`url()` {.collection-method}

The `url` function generates a fully qualified URL to the given path:

```
$url = url('user/profile');

$url = url('user/profile', [1]);
```

If no path is provided, an `Illuminate\Routing\UrlGenerator` instance is returned:

```
$current = url()->current();

$full = url()->full();

$previous = url()->previous();
```

Miscellaneous

`abort()` {.collection-method}

The `abort` function throws an HTTP exception which will be rendered by the exception handler:

```
abort(403);
```

You may also provide the exception's message and custom HTTP response headers that should be sent to the browser:

```
abort(403, 'Unauthorized.', $headers);
```

`abort_if()` {.collection-method}

The `abort_if` function throws an HTTP exception if a given boolean expression evaluates to `true`:

```
abort_if(! Auth::user()->isAdmin(), 403);
```

Like the `abort` method, you may also provide the exception's response text as the third argument and an array of custom response headers as the fourth argument to the function.

`abort_unless()` {.collection-method}

The `abort_unless` function throws an HTTP exception if a given boolean expression evaluates to `false`:

```
abort_unless(Auth::user()->isAdmin(), 403);
```

Like the `abort` method, you may also provide the exception's response text as the third argument and an array of custom response headers as the fourth argument to the function.

`app()` {.collection-method}

The `app` function returns the service container instance:

```
$container = app();
```

You may pass a class or interface name to resolve it from the container:

```
$api = app('HelpSpot\API');
```

`auth()` {.collection-method}

The `auth` function returns an authenticator instance. You may use it as an alternative to the `Auth` facade:

```
$user = auth()->user();
```

If needed, you may specify which guard instance you would like to access:

```
$user = auth('admin')->user();
```

`back()` {.collection-method}

The `back` function generates a redirect HTTP response to the user's previous location:

```
return back($status = 302, $headers = [], $fallback = '/');

return back();
```

`bcrypt()` {.collection-method}

The `bcrypt` function hashes the given value using Bcrypt. You may use this function as an alternative to the `Hash` facade:

```
$password = bcrypt('my-secret-password');
```

`blank()` {.collection-method}

The `blank` function determines whether the given value is "blank":

```
blank('');
blank('   ');
blank(null);
blank(collect());

// true

blank(0);
blank(true);
blank(false);

// false
```

For the inverse of `blank`, see the `filled` method.

`broadcast()` {.collection-method}

The `broadcast` function broadcasts the given event to its listeners:

```
broadcast(new UserRegistered($user));

broadcast(new UserRegistered($user))->toOthers();
```

`cache()` {.collection-method}

The `cache` function may be used to get values from the cache. If the given key does not exist in the cache, an optional default value will be returned:

```
$value = cache('key');

$value = cache('key', 'default');
```

You may add items to the cache by passing an array of key / value pairs to the function. You should also pass the number of seconds or duration the cached value should be considered valid:

```
cache(['key' => 'value'], 300);

cache(['key' => 'value'], now()->addSeconds(10));
```

`class_uses_recursive()` {.collection-method}

The `class_uses_recursive` function returns all traits used by a class, including traits used by all of its parent classes:

```
$traits = class_uses_recursive(App\Models\User::class);
```

`collect()` {.collection-method}

The `collect` function creates a collection instance from the given value:

```
$collection = collect(['taylor', 'abigail']);
```

`config()` {.collection-method}

The `config` function gets the value of a configuration variable. The configuration values may be accessed using "dot" syntax, which includes the name of the file and the option you wish to access. A default value may be specified and is returned if the configuration option does not exist:

```
$value = config('app.timezone');

$value = config('app.timezone', $default);
```

You may set configuration variables at runtime by passing an array of key / value pairs. However, note that this function only affects the configuration value for the current request and does not update your actual configuration values:

```
config(['app.debug' => true]);
```

`cookie()` {.collection-method}

The `cookie` function creates a new cookie instance:

```
$cookie = cookie('name', 'value', $minutes);
```

`csrf_field()` {.collection-method}

The `csrf_field` function generates an HTML `hidden` input field containing the value of the CSRF token. For example, using Blade syntax:

```
{{ csrf_field() }}
```

`csrf_token()` {.collection-method}

The `csrf_token` function retrieves the value of the current CSRF token:

```
$token = csrf_token();
```

`dd()` {.collection-method}

The `dd` function dumps the given variables and ends execution of the script:

```
dd($value);

dd($value1, $value2, $value3, ...);
```

If you do not want to halt the execution of your script, use the dump function instead.

dispatch() {.collection-method}

The dispatch function pushes the given job onto the Laravel job queue:

```
dispatch(new App\Jobs\SendEmails);
```

dump() {.collection-method}

The dump function dumps the given variables:

```
dump($value);

dump($value1, $value2, $value3, ...);
```

If you want to stop executing the script after dumping the variables, use the dd function instead.

env() {.collection-method}

The env function retrieves the value of an environment variable or returns a default value:

```
$env = env('APP_ENV');

$env = env('APP_ENV', 'production');
```

{note} If you execute the config:cache command during your deployment process, you should be sure that you are only calling the env function from within your configuration files. Once the configuration has been cached, the .env file will not be loaded and all calls to the env function will return null.

`event()` {.collection-method}

The `event` function dispatches the given event to its listeners:

```
event(new UserRegistered($user));
```

`filled()` {.collection-method}

The `filled` function determines whether the given value is not "blank":

```
filled(0);
filled(true);
filled(false);

// true

filled('');
filled('   ');
filled(null);
filled(collect());

// false
```

For the inverse of `filled`, see the `blank` method.

`info()` {.collection-method}

The `info` function will write information to your application's log:

```
info('Some helpful information!');
```

An array of contextual data may also be passed to the function:

```
info('User login attempt failed.', ['id' => $user->id]);
```

`logger()` {.collection-method}

The `logger` function can be used to write a `debug` level message to the log:

```
logger('Debug message');
```

An array of contextual data may also be passed to the function:

```
logger('User has logged in.', ['id' => $user->id]);
```

A logger instance will be returned if no value is passed to the function:

```
logger()->error('You are not allowed here.');
```

`method_field()` {.collection-method}

The `method_field` function generates an HTML `hidden` input field containing the spoofed value of the form's HTTP verb. For example, using Blade syntax:

```
<form method="POST">
    {{ method_field('DELETE') }}
</form>
```

`now()` {.collection-method}

The `now` function creates a new `Illuminate\Support\Carbon` instance for the current time:

```
$now = now();
```

`old()` {.collection-method}

The `old` function retrieves an old input value flashed into the session:

```
$value = old('value');

$value = old('value', 'default');
```

`optional()` {.collection-method}

The `optional` function accepts any argument and allows you to access properties or call methods on that object. If the given object is `null`, properties and methods will return `null` instead of causing an error:

```
return optional($user->address)->street;

{!! old('name', optional($user)->name) !!}
```

The `optional` function also accepts a closure as its second argument. The closure will be invoked if the value provided as the first argument is not null:

```
return optional(User::find($id), function ($user) {
    return $user->name;
});
```

`policy()` {.collection-method}

The `policy` method retrieves a policy instance for a given class:

```
$policy = policy(App\Models\User::class);
```

`redirect()` {.collection-method}

The `redirect` function returns a redirect HTTP response, or returns the redirector instance if called with no arguments:

```
return redirect($to = null, $status = 302, $headers = [], $https =
null);

return redirect('/home');

return redirect()->route('route.name');
```

`report()` {.collection-method}

The `report` function will report an exception using your exception handler:

```
report($e);
```

The `report` function also accepts a string as an argument. When a string is given to the function, the function will create an exception with the given string as its message:

```
report('Something went wrong.');
```

`request()` {.collection-method}

The `request` function returns the current request instance or obtains an input field's value from the current request:

```
$request = request();

$value = request('key', $default);
```

`rescue()` {.collection-method}

The `rescue` function executes the given closure and catches any exceptions that occur during its execution. All exceptions that are caught will be sent to your exception handler; however, the request will continue processing:

```
return rescue(function () {
    return $this->method();
});
```

You may also pass a second argument to the `rescue` function. This argument will be the "default" value that should be returned if an exception occurs while executing the closure:

```
return rescue(function () {
    return $this->method();
}, false);

return rescue(function () {
    return $this->method();
}, function () {
    return $this->failure();
});
```

`resolve()` {.collection-method}

The `resolve` function resolves a given class or interface name to an instance using the service container:

```
$api = resolve('HelpSpot\API');
```

`response()` {.collection-method}

The `response` function creates a response instance or obtains an instance of the response factory:

```
return response('Hello World', 200, $headers);

return response()->json(['foo' => 'bar'], 200, $headers);
```

`retry()` {.collection-method}

The `retry` function attempts to execute the given callback until the given maximum

attempt threshold is met. If the callback does not throw an exception, its return value will be returned. If the callback throws an exception, it will automatically be retried. If the maximum attempt count is exceeded, the exception will be thrown:

```
return retry(5, function () {
    // Attempt 5 times while resting 100ms in between attempts...
}, 100);
```

If you would like to manually calculate the number of milliseconds to sleep in between attempts, you may pass a closure as the third argument to the `retry` function:

```
return retry(5, function () {
    // ...
}, function ($attempt) {
    return $attempt * 100;
});
```

To only retry under specific conditions, you may pass a closure as the fourth argument to the `retry` function:

```
return retry(5, function () {
    // ...
}, 100, function ($exception) {
    return $exception instanceof RetryException;
});
```

`session()` {.collection-method}

The `session` function may be used to get or set session values:

```
$value = session('key');
```

You may set values by passing an array of key / value pairs to the function:

```
session(['chairs' => 7, 'instruments' => 3]);
```

The session store will be returned if no value is passed to the function:

```
$value = session()->get('key');

session()->put('key', $value);
```

`tap()` {.collection-method}

The `tap` function accepts two arguments: an arbitrary `$value` and a closure. The `$value` will be passed to the closure and then be returned by the `tap` function. The return value of the closure is irrelevant:

```
$user = tap(User::first(), function ($user) {
    $user->name = 'taylor';

    $user->save();
});
```

If no closure is passed to the `tap` function, you may call any method on the given `$value`. The return value of the method you call will always be `$value`, regardless of what the method actually returns in its definition. For example, the Eloquent `update` method typically returns an integer. However, we can force the method to return the model itself by chaining the `update` method call through the `tap` function:

```
$user = tap($user)->update([
    'name' => $name,
    'email' => $email,
]);
```

To add a `tap` method to a class, you may add the `Illuminate\Support\Traits\Tappable` trait to the class. The `tap` method of this trait accepts a Closure as its only argument. The object instance itself will be passed to the Closure and then be returned by the `tap` method:

```
return $user->tap(function ($user) {
    //
});
```

`throw_if()` {.collection-method}

The `throw_if` function throws the given exception if a given boolean expression evaluates to `true`:

```
throw_if(! Auth::user()->isAdmin(), AuthorizationException::class);

throw_if(
    ! Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page.'
);
```

`throw_unless()` {.collection-method}

The `throw_unless` function throws the given exception if a given boolean expression evaluates to `false`:

```
throw_unless(Auth::user()->isAdmin(), AuthorizationException::class);

throw_unless(
    Auth::user()->isAdmin(),
    AuthorizationException::class,
    'You are not allowed to access this page.'
);
```

`today()` {.collection-method}

The `today` function creates a new `Illuminate\Support\Carbon` instance for the current date:

```
$today = today();
```

`trait_uses_recursive()` {.collection-method}

The `trait_uses_recursive` function returns all traits used by a trait:

```
$traits =
trait_uses_recursive(\Illuminate\Notifications\Notifiable::class);
```

`transform()` {.collection-method}

The `transform` function executes a closure on a given value if the value is not blank and then returns the return value of the closure:

```
$callback = function ($value) {
    return $value * 2;
};

$result = transform(5, $callback);

// 10
```

A default value or closure may be passed as the third argument to the function. This value will be returned if the given value is blank:

```
$result = transform(null, $callback, 'The value is blank');

// The value is blank
```

`validator()` {.collection-method}

The `validator` function creates a new validator instance with the given arguments. You may use it as an alternative to the `Validator` facade:

```
$validator = validator($data, $rules, $messages);
```

`value()` {.collection-method}

The `value` function returns the value it is given. However, if you pass a closure to the function, the closure will be executed and its returned value will be returned:

```
$result = value(true);

// true

$result = value(function () {
    return false;
});

// false
```

`view()` {.collection-method}

The `view` function retrieves a view instance:

```
return view('auth.login');
```

`with()` {.collection-method}

The `with` function returns the value it is given. If a closure is passed as the second argument to the function, the closure will be executed and its returned value will be returned:

```
$callback = function ($value) {
    return (is_numeric($value)) ? $value * 2 : 0;
};

$result = with(5, $callback);

// 10

$result = with(null, $callback);

// 0

$result = with(5, null);

// 5
```

HTTP Client

- Introduction
- Making Requests
    - Request Data
    - Headers
    - Authentication
    - Timeout
    - Retries
    - Error Handling
    - Guzzle Options
- Concurrent Requests
- Testing
    - Faking Responses
    - Inspecting Requests
- Events

Introduction

Laravel provides an expressive, minimal API around the Guzzle HTTP client, allowing you to quickly make outgoing HTTP requests to communicate with other web applications. Laravel's wrapper around Guzzle is focused on its most common use cases and a wonderful developer experience.

Before getting started, you should ensure that you have installed the Guzzle package as a dependency of your application. By default, Laravel automatically includes this dependency. However, if you have previously removed the package, you may install it again via Composer:

```
composer require guzzlehttp/guzzle
```

Making Requests

To make requests, you may use the `get`, `post`, `put`, `patch`, and `delete` methods provided by the `Http` facade. First, let's examine how to make a basic `GET` request to another URL:

```
use Illuminate\Support\Facades\Http;

$response = Http::get('http://example.com');
```

The `get` method returns an instance of `Illuminate\Http\Client\Response`, which provides a variety of methods that may be used to inspect the response:

```
$response->body() : string;
$response->json() : array|mixed;
$response->object() : object;
$response->collect() : Illuminate\Support\Collection;
$response->status() : int;
$response->ok() : bool;
$response->successful() : bool;
$response->failed() : bool;
$response->serverError() : bool;
$response->clientError() : bool;
$response->header($header) : string;
$response->headers() : array;
```

The `Illuminate\Http\Client\Response` object also implements the PHP `ArrayAccess` interface, allowing you to access JSON response data directly on the response:

```
return Http::get('http://example.com/users/1')['name'];
```

Dumping Requests

If you would like to dump the outgoing request instance before it is sent and terminate the script's execution, you may add the `dd` method to the beginning of your request definition:

```
return Http::dd()->get('http://example.com');
```

Request Data

Of course, it is common when making `POST`, `PUT`, and `PATCH` requests to send additional data with your request, so these methods accept an array of data as their second argument. By default, data will be sent using the `application/json` content type:

```
use Illuminate\Support\Facades\Http;

$response = Http::post('http://example.com/users', [
    'name' => 'Steve',
    'role' => 'Network Administrator',
]);
```

GET Request Query Parameters

When making `GET` requests, you may either append a query string to the URL directly or pass an array of key / value pairs as the second argument to the `get` method:

```
$response = Http::get('http://example.com/users', [
    'name' => 'Taylor',
    'page' => 1,
]);
```

Sending Form URL Encoded Requests

If you would like to send data using the `application/x-www-form-urlencoded` content type, you should call the `asForm` method before making your request:

```
$response = Http::asForm()->post('http://example.com/users', [
    'name' => 'Sara',
    'role' => 'Privacy Consultant',
]);
```

Sending A Raw Request Body

You may use the `withBody` method if you would like to provide a raw request body when making a request. The content type may be provided via the method's second argument:

```
$response = Http::withBody(
    base64_encode($photo), 'image/jpeg'
)->post('http://example.com/photo');
```

Multi-Part Requests

If you would like to send files as multi-part requests, you should call the `attach` method before making your request. This method accepts the name of the file and its contents. If needed, you may provide a third argument which will be considered the file's filename:

```
$response = Http::attach(
    'attachment', file_get_contents('photo.jpg'), 'photo.jpg'
)->post('http://example.com/attachments');
```

Instead of passing the raw contents of a file, you may pass a stream resource:

```
$photo = fopen('photo.jpg', 'r');

$response = Http::attach(
    'attachment', $photo, 'photo.jpg'
)->post('http://example.com/attachments');
```

Headers

Headers may be added to requests using the `withHeaders` method. This `withHeaders` method accepts an array of key / value pairs:

```
$response = Http::withHeaders([
    'X-First' => 'foo',
    'X-Second' => 'bar'
])->post('http://example.com/users', [
    'name' => 'Taylor',
]);
```

You may use the `accept` method to specify the content type that your application is expecting in response to your request:

```
$response =
Http::accept('application/json')->get('http://example.com/users');
```

For convenience, you may use the `acceptJson` method to quickly specify that your application expects the `application/json` content type in response to your request:

```
$response = Http::acceptJson()->get('http://example.com/users');
```

Authentication

You may specify basic and digest authentication credentials using the `withBasicAuth` and `withDigestAuth` methods, respectively:

```
// Basic authentication...
$response = Http::withBasicAuth('taylor@laravel.com',
'secret')->post(...);

// Digest authentication...
$response = Http::withDigestAuth('taylor@laravel.com',
'secret')->post(...);
```

Bearer Tokens

If you would like to quickly add a bearer token to the request's `Authorization` header, you may use the `withToken` method:

```
$response = Http::withToken('token')->post(...);
```

Timeout

The `timeout` method may be used to specify the maximum number of seconds to wait for a response:

```
$response = Http::timeout(3)->get(...);
```

If the given timeout is exceeded, an instance of `Illuminate\Http\Client\ConnectionException` will be thrown.

Retries

If you would like HTTP client to automatically retry the request if a client or server error occurs, you may use the `retry` method. The `retry` method accepts the maximum number of times the request should be attempted and the number of milliseconds that Laravel should wait in between attempts:

```
$response = Http::retry(3, 100)->post(...);
```

If needed, you may pass a third argument to the `retry` method. The third argument should be a callable that determines if the retries should actually be attempted. For example, you may wish to only retry the request if the initial request encounters an `ConnectionException`:

```
$response = Http::retry(3, 100, function ($exception) {
    return $exception instanceof ConnectionException;
})->post(...);
```

If all of the requests fail, an instance of `Illuminate\Http\Client\RequestException` will be thrown.

Error Handling

Unlike Guzzle's default behavior, Laravel's HTTP client wrapper does not throw exceptions on client or server errors (`400` and `500` level responses from servers). You may determine if one of these errors was returned using the `successful`, `clientError`, or `serverError` methods:

```
// Determine if the status code is >= 200 and < 300...
$response->successful();

// Determine if the status code is >= 400...
$response->failed();

// Determine if the response has a 400 level status code...
$response->clientError();

// Determine if the response has a 500 level status code...
$response->serverError();
```

Throwing Exceptions

If you have a response instance and would like to throw an instance of `Illuminate\Http\Client\RequestException` if the response status code indicates a client or server error, you may use the `throw` or `throwIf` methods:

```
$response = Http::post(...);

// Throw an exception if a client or server error occurred...
$response->throw();

// Throw an exception if an error occurred and the given condition is
true...
$response->throwIf($condition);

return $response['user']['id'];
```

The `Illuminate\Http\Client\RequestException` instance has a public `$response` property which will allow you to inspect the returned response.

The `throw` method returns the response instance if no error occurred, allowing you to chain other operations onto the `throw` method:

```
return Http::post(...)->throw()->json();
```

If you would like to perform some additional logic before the exception is thrown, you may pass a closure to the `throw` method. The exception will be thrown automatically after the closure is invoked, so you do not need to re-throw the exception from within the closure:

```
return Http::post(...)->throw(function ($response, $e) {
    //
})->json();
```

Guzzle Options

You may specify additional Guzzle request options using the `withOptions` method. The `withOptions` method accepts an array of key / value pairs:

```
$response = Http::withOptions([
    'debug' => true,
])->get('http://example.com/users');
```

Concurrent Requests

Sometimes, you may wish to make multiple HTTP requests concurrently. In other words, you want several requests to be dispatched at the same time instead of issuing the requests sequentially. This can lead to substantial performance improvements when interacting with slow HTTP APIs.

Thankfully, you may accomplish this using the `pool` method. The `pool` method accepts a closure which receives an `Illuminate\Http\Client\Pool` instance, allowing you to easily add requests to the request pool for dispatching:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->get('http://localhost/first'),
    $pool->get('http://localhost/second'),
    $pool->get('http://localhost/third'),
]);

return $responses[0]->ok() &&
       $responses[1]->ok() &&
       $responses[2]->ok();
```

As you can see, each response instance can be accessed based on the order it was added to the pool. If you wish, you can name the requests using the `as` method, which allows you to access the corresponding responses by name:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
    $pool->as('first')->get('http://localhost/first'),
    $pool->as('second')->get('http://localhost/second'),
    $pool->as('third')->get('http://localhost/third'),
]);

return $responses['first']->ok();
```

Testing

Many Laravel services provide functionality to help you easily and expressively write tests, and Laravel's HTTP wrapper is no exception. The `Http` facade's `fake` method allows you to instruct the HTTP client to return stubbed / dummy responses when requests are made.

Faking Responses

For example, to instruct the HTTP client to return empty, `200` status code responses for every request, you may call the `fake` method with no arguments:

```
use Illuminate\Support\Facades\Http;

Http::fake();

$response = Http::post(...);
```

{note} When faking requests, HTTP client middleware are not executed. You should define expectations for faked responses as if these middleware have run correctly.

Faking Specific URLs

Alternatively, you may pass an array to the `fake` method. The array's keys should represent URL patterns that you wish to fake and their associated responses. The `*` character may be used as a wildcard character. Any requests made to URLs that have not been faked will actually be executed. You may use the `Http` facade's `response` method to construct stub / fake responses for these endpoints:

```
Http::fake([
    // Stub a JSON response for GitHub endpoints...
    'github.com/*' => Http::response(['foo' => 'bar'], 200, $headers),

    // Stub a string response for Google endpoints...
    'google.com/*' => Http::response('Hello World', 200, $headers),
]);
```

If you would like to specify a fallback URL pattern that will stub all unmatched URLs, you may use a single `*` character:

```
Http::fake([
    // Stub a JSON response for GitHub endpoints...
    'github.com/*' => Http::response(['foo' => 'bar'], 200,
['Headers']),

    // Stub a string response for all other endpoints...
    '*' => Http::response('Hello World', 200, ['Headers']),
]);
```

Faking Response Sequences

Sometimes you may need to specify that a single URL should return a series of fake responses in a specific order. You may accomplish this using the `Http::sequence` method to build the responses:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
                            ->push('Hello World', 200)
                            ->push(['foo' => 'bar'], 200)
                            ->pushStatus(404),
]);
```

When all of the responses in a response sequence have been consumed, any further requests will cause the response sequence to throw an exception. If you would like to specify a default response that should be returned when a sequence is empty, you may use the `whenEmpty` method:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
                            ->push('Hello World', 200)
                            ->push(['foo' => 'bar'], 200)
                            ->whenEmpty(Http::response()),
]);
```

If you would like to fake a sequence of responses but do not need to specify a specific URL pattern that should be faked, you may use the `Http::fakeSequence` method:

```
Http::fakeSequence()
        ->push('Hello World', 200)
        ->whenEmpty(Http::response());
```

Fake Callback

If you require more complicated logic to determine what responses to return for certain endpoints, you may pass a closure to the `fake` method. This closure will receive an instance of `Illuminate\Http\Client\Request` and should return a response instance. Within your closure, you may perform whatever logic is necessary to determine what type of response to return:

```
Http::fake(function ($request) {
    return Http::response('Hello World', 200);
});
```

Inspecting Requests

When faking responses, you may occasionally wish to inspect the requests the client receives in order to make sure your application is sending the correct data or headers. You may accomplish this by calling the `Http::assertSent` method after calling `Http::fake`.

The `assertSent` method accepts a closure which will receive an `Illuminate\Http\Client\Request` instance and should return a boolean value indicating if the request matches your expectations. In order for the test to pass, at least one request must have been issued matching the given expectations:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::withHeaders([
    'X-First' => 'foo',
])->post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertSent(function (Request $request) {
    return $request->hasHeader('X-First', 'foo') &&
           $request->url() == 'http://example.com/users' &&
           $request['name'] == 'Taylor' &&
           $request['role'] == 'Developer';
});
```

If needed, you may assert that a specific request was not sent using the `assertNotSent` method:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertNotSent(function (Request $request) {
    return $request->url() === 'http://example.com/posts';
});
```

Or, you may use the `assertNothingSent` method to assert that no requests were sent during the test:

```
Http::fake();

Http::assertNothingSent();
```

Events

Laravel fires three events during the process of sending HTTP requests. The `RequestSending` event is fired prior to a request being sent, while the `ResponseReceived` event is fired after a response is received for a given request. The `ConnectionFailed` event is fired if no response is received for a given request.

The `RequestSending` and `ConnectionFailed` events both contain a public `$request` property that you may use to inspect the `Illuminate\Http\Client\Request` instance. Likewise, the `ResponseReceived` event contains a `$request` property as well as a `$response` property which may be used to inspect the `Illuminate\Http\Client\Response` instance. You may register event listeners for this event in your `App\Providers\EventServiceProvider` service provider:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Http\Client\Events\RequestSending' => [
        'App\Listeners\LogRequestSending',
    ],
    'Illuminate\Http\Client\Events\ResponseReceived' => [
        'App\Listeners\LogResponseReceived',
    ],
    'Illuminate\Http\Client\Events\ConnectionFailed' => [
        'App\Listeners\LogConnectionFailed',
    ],
];
```

Mail

- Introduction
    - Configuration
    - Driver Prerequisites
    - Failover Configuration
- Generating Mailables
- Writing Mailables

    - Configuring The Sender
    - Configuring The View
    - View Data
    - Attachments
    - Inline Attachments
    - Customizing The Symfony Message
- Markdown Mailables
    - Generating Markdown Mailables
    - Writing Markdown Messages
    - Customizing The Components
- Sending Mail
    - Queueing Mail
- Rendering Mailables
    - Previewing Mailables In The Browser
- Localizing Mailables
- Testing Mailables
- Mail & Local Development
- Events

Introduction

Sending email doesn't have to be complicated. Laravel provides a clean, simple email API powered by the popular Symfony Mailer component. Laravel and Symfony Mailer provide drivers for sending email via SMTP, Mailgun, Postmark, Amazon SES, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Configuration

Laravel's email services may be configured via your application's `config/mail.php` configuration file. Each mailer configured within this file may have its own unique configuration and even its own unique "transport", allowing your application to use different email services to send certain email messages. For example, your application might use Postmark to send transactional emails while using Amazon SES to send bulk emails.

Within your `mail` configuration file, you will find a `mailers` configuration array. This array contains a sample configuration entry for each of the major mail drivers / transports supported by Laravel, while the `default` configuration value determines which mailer will be used by default when your application needs to send an email message.

Driver / Transport Prerequisites

The API based drivers such as Mailgun and Postmark are often simpler and faster than sending mail via SMTP servers. Whenever possible, we recommend that you use one of these drivers.

Mailgun Driver

To use the Mailgun driver, install Symfony's Mailgun Mailer transport via Composer:

```
composer require symfony/mailgun-mailer
```

Next, set the `default` option in your application's `config/mail.php` configuration file to `mailgun`. After configuring your application's default mailer, verify that your `config/services.php` configuration file contains the following options:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
],
```

If you are not using the United States Mailgun region, you may define your region's endpoint in the `services` configuration file:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
],
```

Postmark Driver

To use the Postmark driver, install Symfony's Postmark Mailer transport via Composer:

```
composer require symfony/postmark-mailer
```

Next, set the `default` option in your application's `config/mail.php` configuration file to `postmark`. After configuring your application's default mailer, verify that your `config/services.php` configuration file contains the following options:

```
'postmark' => [
    'token' => env('POSTMARK_TOKEN'),
],
```

If you would like to specify the Postmark message stream that should be used by a given mailer, you may add the `message_stream_id` configuration option to the mailer's configuration array. This configuration array can be found in your application's `config/mail.php` configuration file:

```
'postmark' => [
    'transport' => 'postmark',
    'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
],
```

This way you are also able to set up multiple Postmark mailers with different message streams.

SES Driver

To use the SES driver, install Symfony's Amazon Mailer transport via Composer:

```
composer require symfony/amazon-mailer
```

Next, set the `default` option in your `config/mail.php` configuration file to `ses` and verify that your `config/services.php` configuration file contains the following options:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
],
```

To utilize AWS temporary credentials via a session token, you may add a `token` key to your application's SES configuration:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'token' => env('AWS_SESSION_TOKEN'),
],
```

If you would like to define additional options that Laravel should pass to the AWS SDK's `SendEmail` method when sending an email, you may define an `options` array within your `ses` configuration:

```
'ses' => [
    'key' => env('AWS_ACCESS_KEY_ID'),
    'secret' => env('AWS_SECRET_ACCESS_KEY'),
    'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
    'options' => [
        'ConfigurationSetName' => 'MyConfigurationSet',
        'EmailTags' => [
            ['Name' => 'foo', 'Value' => 'bar'],
        ],
    ],
],
```

Failover Configuration

Sometimes, an external service you have configured to send your application's mail may be down. In these cases, it can be useful to define one or more backup mail delivery configurations that will be used in case your primary delivery driver is down.

To accomplish this, you should define a mailer within your application's `mail` configuration file that uses the `failover` transport. The configuration array for your application's `failover` mailer should contain an array of `mailers` that reference the order in which mail drivers should be chosen for delivery:

```
'mailers' => [
    'failover' => [
        'transport' => 'failover',
        'mailers' => [
            'postmark',
            'mailgun',
            'sendmail',
        ],
    ],

    // ...
],
```

Once your failover mailer has been defined, you should set this mailer as the default mailer used by your application by specifying its name as the value of the `default` configuration key within your application's `mail` configuration file:

```
'default' => env('MAIL_MAILER', 'failover'),
```

Generating Mailables

When building Laravel applications, each type of email sent by your application is represented as a "mailable" class. These classes are stored in the `app/Mail` directory. Don't worry if you don't see this directory in your application, since it will be generated for you when you create your first mailable class using the `make:mail` Artisan command:

```
php artisan make:mail OrderShipped
```

Writing Mailables

Once you have generated a mailable class, open it up so we can explore its contents. First, note that all of a mailable class' configuration is done in the `build` method. Within this method, you may call various methods such as `from`, `subject`, `view`, and `attach` to configure the email's presentation and delivery.

{tip} You may type-hint dependencies on the mailable's `build` method. The Laravel service container automatically injects these dependencies.

Configuring The Sender

Using The `from` Method

First, let's explore configuring the sender of the email. Or, in other words, who the email is going to be "from". There are two ways to configure the sender. First, you may use the `from` method within your mailable class' `build` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com', 'Example')
                ->view('emails.orders.shipped');
}
```

Using A Global `from` Address

However, if your application uses the same "from" address for all of its emails, it can become cumbersome to call the `from` method in each mailable class you generate. Instead, you may specify a global "from" address in your `config/mail.php` configuration file. This address will be used if no other "from" address is specified within the mailable class:

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

In addition, you may define a global "reply_to" address within your `config/mail.php` configuration file:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App
Name'],
```

Configuring The View

Within a mailable class' `build` method, you may use the `view` method to specify which template should be used when rendering the email's contents. Since each email typically uses a Blade template to render its contents, you have the full power and convenience of the Blade templating engine when building your email's HTML:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped');
}
```

{tip} You may wish to create a `resources/views/emails` directory to house all of your email templates; however, you are free to place them wherever you wish within your `resources/views` directory.

Plain Text Emails

If you would like to define a plain-text version of your email, you may use the `text` method. Like the `view` method, the `text` method accepts a template name which will be used to render the contents of the email. You are free to define both an HTML and plain-text version of your message:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->text('emails.orders.shipped_plain');
}
```

View Data

Via Public Properties

Typically, you will want to pass some data to your view that you can utilize when rendering the email's HTML. There are two ways you may make data available to your view. First, any public property defined on your mailable class will automatically be made available to the view. So, for example, you may pass data into your mailable class' constructor and set that data to public properties defined on the class:

```php
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    public $order;

    /**
     * Create a new message instance.
     *
     * @param  \App\Models\Order  $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped');
    }
}
```

Once the data has been set to a public property, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
    Price: {{ $order->price }}
</div>
```

Via The `with` Method:

If you would like to customize the format of your email's data before it is sent to the template, you may manually pass your data to the view via the `with` method. Typically, you will still pass data via the mailable class' constructor; however, you should set this data to `protected` or `private` properties so the data is not automatically made available to the template. Then, when calling the `with` method, pass an array of data that you wish to make available to the template:

```
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * The order instance.
     *
     * @var \App\Models\Order
     */
    protected $order;

    /**
     * Create a new message instance.
     *
     * @param  \App\Models\Order  $order
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }
```

```
    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped')
                    ->with([
                        'orderName' => $this->order->name,
                        'orderPrice' => $this->order->price,
                    ]);
    }
}
```

Once the data has been passed to the `with` method, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

```
<div>
    Price: {{ $orderPrice }}
</div>
```

Attachments

To add attachments to an email, use the `attach` method within the mailable class' `build` method. The `attach` method accepts the full path to the file as its first argument:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attach('/path/to/file');
}
```

When attaching files to a message, you may also specify the display name and / or MIME type by passing an `array` as the second argument to the `attach` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attach('/path/to/file', [
                    'as' => 'name.pdf',
                    'mime' => 'application/pdf',
                ]);
}
```

Attaching Files From Disk

If you have stored a file on one of your filesystem disks, you may attach it to the email using the `attachFromStorage` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
   return $this->view('emails.orders.shipped')
               ->attachFromStorage('/path/to/file');
}
```

If necessary, you may specify the file's attachment name and additional options using the second and third arguments to the `attachFromStorage` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
   return $this->view('emails.orders.shipped')
               ->attachFromStorage('/path/to/file', 'name.pdf', [
                   'mime' => 'application/pdf'
               ]);
}
```

The `attachFromStorageDisk` method may be used if you need to specify a storage disk other than your default disk:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
   return $this->view('emails.orders.shipped')
               ->attachFromStorageDisk('s3', '/path/to/file');
}
```

Raw Data Attachments

The `attachData` method may be used to attach a raw string of bytes as an attachment. For example, you might use this method if you have generated a PDF in memory and want to attach it to the email without writing it to disk. The `attachData` method accepts the raw data bytes as its first argument, the name of the file as its second argument, and an array of options as its third argument:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attachData($this->pdf, 'name.pdf', [
                    'mime' => 'application/pdf',
                ]);
}
```

Inline Attachments

Embedding inline images into your emails is typically cumbersome; however, Laravel provides a convenient way to attach images to your emails. To embed an inline image, use the `embed` method on the `$message` variable within your email template. Laravel automatically makes the `$message` variable available to all of your email templates, so you don't need to worry about passing it in manually:

```
<body>
    Here is an image:

    <img src="{{ $message->embed($pathToImage) }}">
</body>
```

{note} The `$message` variable is not available in plain-text message templates since plain-text messages do not utilize inline attachments.

Embedding Raw Data Attachments

If you already have a raw image data string you wish to embed into an email template, you may call the `embedData` method on the `$message` variable. When calling the `embedData` method, you will need to provide a filename that should be assigned to the embedded image:

```
<body>
    Here is an image from raw data:

    <img src="{{ $message->embedData($data, 'example-image.jpg') }}">
</body>
```

Customizing The Symfony Message

The `withSymfonyMessage` method of the `Mailable` base class allows you to register a closure which will be invoked with the Symfony Message instance before sending the message. This gives you an opportunity to deeply customize the message before it is delivered:

```
use Symfony\Component\Mime\Email;

/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    $this->view('emails.orders.shipped');

    $this->withSymfonyMessage(function (Email $message) {
        $message->getHeaders()->addTextHeader(
            'Custom-Header', 'Header Value'
        );
    });
    return $this;
}
```

Markdown Mailables

Markdown mailable messages allow you to take advantage of the pre-built templates and components of mail notifications in your mailables. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating Markdown Mailables

To generate a mailable with a corresponding Markdown template, you may use the `--markdown` option of the `make:mail` Artisan command:

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

Then, when configuring the mailable within its `build` method, call the `markdown` method instead of the `view` method. The `markdown` method accepts the name of the Markdown template and an optional array of data to make available to the template:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
                ->markdown('emails.orders.shipped', [
                    'url' => $this->orderUrl,
                ]);
}
```

Writing Markdown Messages

Markdown mailables use a combination of Blade components and Markdown syntax which allow you to easily construct mail messages while leveraging Laravel's pre-built email UI

components:

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

{tip} Do not use excess indentation when writing Markdown emails. Per Markdown standards, Markdown parsers will render indented content as code blocks.

Button Component

The button component renders a centered button link. The component accepts two arguments, a `url` and an optional `color`. Supported colors are `primary`, `success`, and `error`. You may add as many button components to a message as you wish:

```
@component('mail::button', ['url' => $url, 'color' => 'success'])
View Order
@endcomponent
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the message. This allows you to draw attention to a given block of text:

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
@component('mail::table')
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

Customizing The Components

You may export all of the Markdown mail components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

```
php artisan vendor:publish --tag=laravel-mail
```

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing The CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be converted to inline CSS styles within the HTML representations of your Markdown mail messages.

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of your application's `config/mail.php` configuration file to match the name of your new theme.

To customize the theme for an individual mailable, you may set the `$theme` property of the mailable class to the name of the theme that should be used when sending that mailable.

Sending Mail

To send a message, use the `to` method on the `Mail` facade. The `to` method accepts an email address, a user instance, or a collection of users. If you pass an object or collection of objects, the mailer will automatically use their `email` and `name` properties when determining the email's recipients, so make sure these attributes are available on your objects. Once you have specified your recipients, you may pass an instance of your mailable class to the `send` method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $order = Order::findOrFail($request->order_id);

        // Ship the order...

        Mail::to($request->user())->send(new OrderShipped($order));
    }
}
```

You are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients by chaining their respective methods together:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new OrderShipped($order));
```

Looping Over Recipients

Occasionally, you may need to send a mailable to a list of recipients by iterating over an array of recipients / email addresses. However, since the `to` method appends email addresses to the mailable's list of recipients, each iteration through the loop will send another email to every previous recipient. Therefore, you should always re-create the mailable instance for each recipient:

```
foreach (['taylor@example.com', 'dries@example.com'] as $recipient) {
    Mail::to($recipient)->send(new OrderShipped($order));
}
```

Sending Mail Via A Specific Mailer

By default, Laravel will send email using the mailer configured as the `default` mailer in your application's `mail` configuration file. However, you may use the `mailer` method to send a message using a specific mailer configuration:

```
Mail::mailer('postmark')
        ->to($request->user())
        ->send(new OrderShipped($order));
```

Queueing Mail

Queueing A Mail Message

Since sending email messages can negatively impact the response time of your application, many developers choose to queue email messages for background sending. Laravel makes this easy using its built-in unified queue API. To queue a mail message, use the `queue` method on the `Mail` facade after specifying the message's recipients:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue(new OrderShipped($order));
```

This method will automatically take care of pushing a job onto the queue so the message is sent in the background. You will need to configure your queues before using this feature.

Delayed Message Queueing

If you wish to delay the delivery of a queued email message, you may use the `later` method. As its first argument, the `later` method accepts a `DateTime` instance indicating when the message should be sent:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later(now()->addMinutes(10), new OrderShipped($order));
```

Pushing To Specific Queues

Since all mailable classes generated using the `make:mail` command make use of the `Illuminate\Bus\Queueable` trait, you may call the `onQueue` and `onConnection` methods on any mailable class instance, allowing you to specify the connection and queue name for the message:

```
$message = (new OrderShipped($order))
                ->onConnection('sqs')
                ->onQueue('emails');

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->queue($message);
```

Queueing By Default

If you have mailable classes that you want to always be queued, you may implement the

`ShouldQueue` contract on the class. Now, even if you call the `send` method when mailing, the mailable will still be queued since it implements the contract:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
    //
}
```

Queued Mailables & Database Transactions

When queued mailables are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your mailable depends on these models, unexpected errors can occur when the job that sends the queued mailable is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued mailable should be dispatched after all open database transactions have been committed by calling the `afterCommit` method when sending the mail message:

```
Mail::to($request->user())->send(
    (new OrderShipped($order))->afterCommit()
);
```

Alternatively, you may call the `afterCommit` method from your mailable's constructor:

```php
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable implements ShouldQueue
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

{tip} To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Rendering Mailables

Sometimes you may wish to capture the HTML content of a mailable without sending it. To accomplish this, you may call the `render` method of the mailable. This method will return the evaluated HTML content of the mailable as a string:

```
use App\Mail\InvoicePaid;
use App\Models\Invoice;

$invoice = Invoice::find(1);

return (new InvoicePaid($invoice))->render();
```

Previewing Mailables In The Browser

When designing a mailable's template, it is convenient to quickly preview the rendered mailable in your browser like a typical Blade template. For this reason, Laravel allows you to return any mailable directly from a route closure or controller. When a mailable is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

```
Route::get('/mailable', function () {
    $invoice = App\Models\Invoice::find(1);

    return new App\Mail\InvoicePaid($invoice);
});
```

{note} Inline attachments will not be rendered when a mailable is previewed in your browser. To preview these mailables, you should send them to an email testing application such as MailHog or HELO.

Localizing Mailables

Laravel allows you to send mailables in a locale other than the request's current locale, and will even remember this locale if the mail is queued.

To accomplish this, the `Mail` facade offers a `locale` method to set the desired language. The application will change into this locale when the mailable's template is being evaluated and then revert back to the previous locale when evaluation is complete:

```
Mail::to($request->user())->locale('es')->send(
    new OrderShipped($order)
);
```

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on one or more of your models, you may instruct Laravel to use this stored locale when sending mail:

```
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     *
     * @return string
     */
    public function preferredLocale()
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending mailables and notifications to the model. Therefore, there is no need to call the `locale` method when using this interface:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

Testing Mailables

Laravel provides several convenient methods for testing that your mailables contain the content that you expect. These methods are: `assertSeeInHtml`, `assertDontSeeInHtml`, `assertSeeInText`, and `assertDontSeeInText`.

As you might expect, the "HTML" assertions assert that the HTML version of your mailable contains a given string, while the "text" assertions assert that the plain-text version of your mailable contains a given string:

```
use App\Mail\InvoicePaid;
use App\Models\User;

public function test_mailable_content()
{
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);

    $mailable->assertSeeInHtml($user->email);
    $mailable->assertSeeInHtml('Invoice Paid');

    $mailable->assertSeeInText($user->email);
    $mailable->assertSeeInText('Invoice Paid');
}
```

Testing Mailable Sending

We suggest testing the content of your mailables separately from your tests that assert that a given mailable was "sent" to a specific user. To learn how to test that mailables were sent, check out our documentation on the Mail fake.

Mail & Local Development

When developing an application that sends email, you probably don't want to actually send emails to live email addresses. Laravel provides several ways to "disable" the actual sending of emails during local development.

Log Driver

Instead of sending your emails, the `log` mail driver will write all email messages to your log files for inspection. Typically, this driver would only be used during local development. For more information on configuring your application per environment, check out the configuration documentation.

HELO / Mailtrap / MailHog

Finally, you may use a service like HELO or Mailtrap and the `smtp` driver to send your email messages to a "dummy" mailbox where you may view them in a true email client. This approach has the benefit of allowing you to actually inspect the final emails in Mailtrap's message viewer.

If you are using Laravel Sail, you may preview your messages using MailHog. When Sail is running, you may access the MailHog interface at: `http://localhost:8025`.

Events

Laravel fires two events during the process of sending mail messages. The `MessageSending` event is fired prior to a message being sent, while the `MessageSent` event is fired after a message has been sent. Remember, these events are fired when the mail is being *sent*, not when it is queued. You may register event listeners for this event in your `App\Providers\EventServiceProvider` service provider:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
    'Illuminate\Mail\Events\MessageSending' => [
        'App\Listeners\LogSendingMessage',
    ],
    'Illuminate\Mail\Events\MessageSent' => [
        'App\Listeners\LogSentMessage',
    ],
];
```

Database: Pagination

- Introduction
- Basic Usage
    - Paginating Query Builder Results
    - Paginating Eloquent Results
    - Cursor Pagination
    - Manually Creating A Paginator
    - Customizing Pagination URLs
- Displaying Pagination Results
    - Adjusting The Pagination Link Window
    - Converting Results To JSON
- Customizing The Pagination View
    - Using Bootstrap
- Paginator and LengthAwarePaginator Instance Methods
- Cursor Paginator Instance Methods

Introduction

In other frameworks, pagination can be very painful. We hope Laravel's approach to pagination will be a breath of fresh air. Laravel's paginator is integrated with the query builder and Eloquent ORM and provides convenient, easy-to-use pagination of database records with zero configuration.

By default, the HTML generated by the paginator is compatible with the Tailwind CSS framework; however, Bootstrap pagination support is also available.

Basic Usage

Paginating Query Builder Results

There are several ways to paginate items. The simplest is by using the `paginate` method on the query builder or an Eloquent query. The `paginate` method automatically takes care of setting the query's "limit" and "offset" based on the current page being viewed by the user. By default, the current page is detected by the value of the `page` query string argument on the HTTP request. This value is automatically detected by Laravel, and is also automatically inserted into links generated by the paginator.

In this example, the only argument passed to the `paginate` method is the number of items you would like displayed "per page". In this case, let's specify that we would like to display `15` items per page:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show all application users.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return view('user.index', [
            'users' => DB::table('users')->paginate(15)
        ]);
    }
}
```

Simple Pagination

The `paginate` method counts the total number of records matched by the query before retrieving the records from the database. This is done so that the paginator knows how many pages of records there are in total. However, if you do not plan to show the total number of pages in your application's UI then the record count query is unnecessary.

Therefore, if you only need to display simple "Next" and "Previous" links in your application's UI, you may use the `simplePaginate` method to perform a single, efficient query:

```
$users = DB::table('users')->simplePaginate(15);
```

Paginating Eloquent Results

You may also paginate Eloquent queries. In this example, we will paginate the `App\Models\User` model and indicate that we plan to display 15 records per page. As you can see, the syntax is nearly identical to paginating query builder results:

```
use App\Models\User;

$users = User::paginate(15);
```

Of course, you may call the `paginate` method after setting other constraints on the query, such as `where` clauses:

```
$users = User::where('votes', '>', 100)->paginate(15);
```

You may also use the `simplePaginate` method when paginating Eloquent models:

```
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

Similarly, you may use the `cursorPaginate` method to cursor paginate Eloquent models:

```
$users = User::where('votes', '>', 100)->cursorPaginate(15);
```

Multiple Paginator Instances Per Page

Sometimes you may need to render two separate paginators on a single screen that is rendered by your application. However, if both paginator instances use the `page` query string parameter to store the current page, the two paginator's will conflict. To resolve this conflict, you may pass the name of the query string parameter you wish to use to store the paginator's current page via the third argument provided to the `paginate`, `simplePaginate`, and `cursorPaginate` methods:

```
use App\Models\User;

$users = User::where('votes', '>', 100)->paginate(
    $perPage = 15, $columns = ['*'], $pageName = 'users'
);
```

Cursor Pagination

While `paginate` and `simplePaginate` create queries using the SQL "offset" clause, cursor pagination works by constructing "where" clauses that compare the values of the ordered columns contained in the query, providing the most efficient database performance available amongst all of Laravel's pagination methods. This method of pagination is particularly well-suited for large data-sets and "infinite" scrolling user interfaces.

Unlike offset based pagination, which includes a page number in the query string of the URLs generated by the paginator, cursor based pagination places a "cursor" string in the query string. The cursor is an encoded string containing the location that the next paginated query should start paginating and the direction that it should paginate:

```
http://localhost/users?cursor=eyJpZCI6MTUsIl9wb2ludHNUb05leHRJdGVtcyI6dH
J1ZX0
```

You may create a cursor based paginator instance via the `cursorPaginate` method offered by the query builder. This method returns an instance of `Illuminate\Pagination\CursorPaginator`:

```
$users = DB::table('users')->orderBy('id')->cursorPaginate(15);
```

Once you have retrieved a cursor paginator instance, you may display the pagination results as you typically would when using the `paginate` and `simplePaginate` methods. For more information on the instance methods offered by the cursor paginator, please consult the cursor paginator instance method documentation.

{note} Your query must contain an "order by" clause in order to take advantage of cursor pagination.

Cursor vs. Offset Pagination

To illustrate the differences between offset pagination and cursor pagination, let's examine some example SQL queries. Both of the following queries will both display the "second page" of results for a `users` table ordered by `id`:

```
# Offset Pagination...
select * from users order by id asc limit 15 offset 15;

# Cursor Pagination...
select * from users where id > 15 order by id asc limit 15;
```

The cursor pagination query offers the following advantages over offset pagination:

- For large data-sets, cursor pagination will offer better performance if the "order by" columns are indexed. This is because the "offset" clause scans through all previously matched data.
- For data-sets with frequent writes, offset pagination may skip records or show duplicates if results have been recently added to or deleted from the page a user is currently viewing.

However, cursor pagination has the following limitations:

- Like `simplePaginate`, cursor pagination can only be used to display "Next" and "Previous" links and does not support generating links with page numbers.
- It requires that the ordering is based on at least one unique column or a combination of columns that are unique. Columns with `null` values are not supported.
- Query expressions in "order by" clauses are supported only if they are aliased and added to the "select" clause as well.

Manually Creating A Paginator

Sometimes you may wish to create a pagination instance manually, passing it an array of items that you already have in memory. You may do so by creating either an `Illuminate\Pagination\Paginator`, `Illuminate\Pagination\LengthAwarePaginator` or `Illuminate\Pagination\CursorPaginator` instance, depending on your needs.

The `Paginator` and `CursorPaginator` classes do not need to know the total number of items in the result set; however, because of this, these classes do not have methods for retrieving the index of the last page. The `LengthAwarePaginator` accepts almost the same arguments as the `Paginator`; however, it requires a count of the total number of items in the result set.

In other words, the `Paginator` corresponds to the `simplePaginate` method on the query builder, the `CursorPaginator` corresponds to the `cursorPaginate` method, and the `LengthAwarePaginator` corresponds to the `paginate` method.

{note} When manually creating a paginator instance, you should manually "slice" the array of results you pass to the paginator. If you're unsure how to do this, check out the array_slice PHP function.

Customizing Pagination URLs

By default, links generated by the paginator will match the current request's URI. However, the paginator's `withPath` method allows you to customize the URI used by the paginator when generating links. For example, if you want the paginator to generate links like `http://example.com/admin/users?page=N`, you should pass `/admin/users` to the `withPath` method:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->withPath('/admin/users');

    //
});
```

Appending Query String Values

You may append to the query string of pagination links using the `appends` method. For example, to append `sort=votes` to each pagination link, you should make the following call to `appends`:

```
use App\Models\User;

Route::get('/users', function () {
    $users = User::paginate(15);

    $users->appends(['sort' => 'votes']);

    //
});
```

You may use the `withQueryString` method if you would like to append all of the current request's query string values to the pagination links:

```
$users = User::paginate(15)->withQueryString();
```

Appending Hash Fragments

If you need to append a "hash fragment" to URLs generated by the paginator, you may use the `fragment` method. For example, to append `#users` to the end of each pagination link, you should invoke the `fragment` method like so:

```
$users = User::paginate(15)->fragment('users');
```

Displaying Pagination Results

When calling the `paginate` method, you will receive an instance of `Illuminate\Pagination\LengthAwarePaginator`, while calling the `simplePaginate` method returns an instance of `Illuminate\Pagination\Paginator`. And, finally, calling the `cursorPaginate` method returns an instance of `Illuminate\Pagination\CursorPaginator`.

These objects provide several methods that describe the result set. In addition to these helpers methods, the paginator instances are iterators and may be looped as an array. So, once you have retrieved the results, you may display the results and render the page links using Blade:

```
<div class="container">
    @foreach ($users as $user)
        {{ $user->name }}
    @endforeach
</div>

{{ $users->links() }}
```

The `links` method will render the links to the rest of the pages in the result set. Each of these links will already contain the proper `page` query string variable. Remember, the HTML generated by the `links` method is compatible with the Tailwind CSS framework.

Adjusting The Pagination Link Window

When the paginator displays pagination links, the current page number is displayed as well as links for the three pages before and after the current page. Using the `onEachSide` method, you may control how many additional links are displayed on each side of the current page within the middle, sliding window of links generated by the paginator:

```
{{ $users->onEachSide(5)->links() }}
```

Converting Results To JSON

The Laravel paginator classes implement the `Illuminate\Contracts\Support\Jsonable` Interface contract and expose the `toJson` method, so it's very easy to convert your pagination results to JSON. You may also convert a paginator instance to JSON by returning it from a route or controller action:

```
use App\Models\User;

Route::get('/users', function () {
    return User::paginate();
});
```

The JSON from the paginator will include meta information such as `total`, `current_page`, `last_page`, and more. The result records are available via the `data` key in the JSON array. Here is an example of the JSON created by returning a paginator instance from a route:

```
{
   "total": 50,
   "per_page": 15,
   "current_page": 1,
   "last_page": 4,
   "first_page_url": "http://laravel.app?page=1",
   "last_page_url": "http://laravel.app?page=4",
   "next_page_url": "http://laravel.app?page=2",
   "prev_page_url": null,
   "path": "http://laravel.app",
   "from": 1,
   "to": 15,
   "data":[
        {
            // Record...
        },
        {
            // Record...
        }
   ]
}
```

Customizing The Pagination View

By default, the views rendered to display the pagination links are compatible with the Tailwind CSS framework. However, if you are not using Tailwind, you are free to define your own views to render these links. When calling the `links` method on a paginator instance, you may pass the view name as the first argument to the method:

```
{{ $paginator->links('view.name') }}

// Passing additional data to the view...
{{ $paginator->links('view.name', ['foo' => 'bar']) }}
```

However, the easiest way to customize the pagination views is by exporting them to your `resources/views/vendor` directory using the `vendor:publish` command:

```
php artisan vendor:publish --tag=laravel-pagination
```

This command will place the views in your application's `resources/views/vendor/pagination` directory. The `tailwind.blade.php` file within this directory corresponds to the default pagination view. You may edit this file to modify the pagination HTML.

If you would like to designate a different file as the default pagination view, you may invoke the paginator's `defaultView` and `defaultSimpleView` methods within the `boot` method of your `App\Providers\AppServiceProvider` class:

```php
<?php

namespace App\Providers;

use Illuminate\Pagination\Paginator;
use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Paginator::defaultView('view-name');

        Paginator::defaultSimpleView('view-name');
    }
}
```

Using Bootstrap

Laravel includes pagination views built using Bootstrap CSS. To use these views instead of the default Tailwind views, you may call the paginator's `useBootstrap` method within the `boot` method of your `App\Providers\AppServiceProvider` class:

```
use Illuminate\Pagination\Paginator;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Paginator::useBootstrap();
}
```

Paginator / LengthAwarePaginator Instance Methods

Each paginator instance provides additional pagination information via the following methods:

Method	Description
`$paginator->count()`	Get the number of items for the current page.
`$paginator->currentPage()`	Get the current page number.
`$paginator->firstItem()`	Get the result number of the first item in the results.
`$paginator->getOptions()`	Get the paginator options.
`$paginator->getUrlRange($start, $end)`	Create a range of pagination URLs.
`$paginator->hasPages()`	Determine if there are enough items to split into multiple pages.
`$paginator->hasMorePages()`	Determine if there are more items in the data store.
`$paginator->items()`	Get the items for the current page.
`$paginator->lastItem()`	Get the result number of the last item in the results.
`$paginator->lastPage()`	Get the page number of the last available page. (Not available when using `simplePaginate`).
`$paginator->nextPageUrl()`	Get the URL for the next page.
`$paginator->onFirstPage()`	Determine if the paginator is on the first page.
`$paginator->perPage()`	The number of items to be shown per page.
`$paginator->previousPageUrl()`	Get the URL for the previous page.

Method	Description
`$paginator->total()`	Determine the total number of matching items in the data store. (Not available when using `simplePaginate`).
`$paginator->url($page)`	Get the URL for a given page number.
`$paginator->getPageName()`	Get the query string variable used to store the page.
`$paginator->setPageName($name)`	Set the query string variable used to store the page.

Cursor Paginator Instance Methods

Each cursor paginator instance provides additional pagination information via the following methods:

Method	Description
`$paginator->count()`	Get the number of items for the current page.
`$paginator->cursor()`	Get the current cursor instance.
`$paginator->getOptions()`	Get the paginator options.
`$paginator->hasPages()`	Determine if there are enough items to split into multiple pages.
`$paginator->hasMorePages()`	Determine if there are more items in the data store.
`$paginator->getCursorName()`	Get the query string variable used to store the cursor.
`$paginator->items()`	Get the items for the current page.
`$paginator->nextCursor()`	Get the cursor instance for the next set of items.
`$paginator->nextPageUrl()`	Get the URL for the next page.
`$paginator->onFirstPage()`	Determine if the paginator is on the first page.
`$paginator->perPage()`	The number of items to be shown per page.
`$paginator->previousCursor()`	Get the cursor instance for the previous set of items.
`$paginator->previousPageUrl()`	Get the URL for the previous page.
`$paginator->setCursorName()`	Set the query string variable used to store the cursor.
`$paginator->url($cursor)`	Get the URL for a given cursor instance.

Database: Query Builder

- Introduction
- Running Database Queries
    - Chunking Results
    - Streaming Results Lazily
    - Aggregates
- Select Statements
- Raw Expressions
- Joins
- Unions
- Basic Where Clauses
    - Where Clauses
    - Or Where Clauses
    - JSON Where Clauses
    - Additional Where Clauses
    - Logical Grouping
- Advanced Where Clauses
    - Where Exists Clauses
    - Subquery Where Clauses
- Ordering, Grouping, Limit & Offset
    - Ordering
    - Grouping
    - Limit & Offset
- Conditional Clauses
- Insert Statements
    - Upserts
- Update Statements
    - Updating JSON Columns
    - Increment & Decrement
- Delete Statements
- Pessimistic Locking
- Debugging

Introduction

Laravel's database query builder provides a convenient, fluent interface to creating and running database queries. It can be used to perform most database operations in your application and works perfectly with all of Laravel's supported database systems.

The Laravel query builder uses PDO parameter binding to protect your application against SQL injection attacks. There is no need to clean or sanitize strings passed to the query builder as query bindings.

{note} PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns.

Running Database Queries

Retrieving All Rows From A Table

You may use the `table` method provided by the `DB` facade to begin a query. The `table` method returns a fluent query builder instance for the given table, allowing you to chain more constraints onto the query and then finally retrieve the results of the query using the `get` method:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
    /**
     * Show a list of all of the application's users.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

The `get` method returns an `Illuminate\Support\Collection` instance containing the results of the query where each result is an instance of the PHP `stdClass` object. You may access each column's value by accessing the column as a property of the object:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->get();

foreach ($users as $user) {
    echo $user->name;
}
```

{tip} Laravel collections provide a variety of extremely powerful methods for mapping and reducing data. For more information on Laravel collections, check out the collection documentation.

Retrieving A Single Row / Column From A Table

If you just need to retrieve a single row from a database table, you may use the `DB` facade's `first` method. This method will return a single `stdClass` object:

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

If you don't need an entire row, you may extract a single value from a record using the `value` method. This method will return the value of the column directly:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

To retrieve a single row by its `id` column value, use the `find` method:

```
$user = DB::table('users')->find(3);
```

Retrieving A List Of Column Values

If you would like to retrieve an `Illuminate\Support\Collection` instance containing the values of a single column, you may use the `pluck` method. In this example, we'll retrieve a collection of user titles:

```
use Illuminate\Support\Facades\DB;

$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
    echo $title;
}
```

You may specify the column that the resulting collection should use as its keys by providing a second argument to the `pluck` method:

```
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
    echo $title;
}
```

Chunking Results

If you need to work with thousands of database records, consider using the `chunk` method provided by the `DB` facade. This method retrieves a small chunk of results at a time and feeds each chunk into a closure for processing. For example, let's retrieve the entire `users` table in chunks of 100 records at a time:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    foreach ($users as $user) {
        //
    }
});
```

You may stop further chunks from being processed by returning `false` from the closure:

```
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
    // Process the records...

    return false;
});
```

If you are updating database records while chunking results, your chunk results could change in unexpected ways. If you plan to update the retrieved records while chunking, it is always best to use the `chunkById` method instead. This method will automatically paginate the results based on the record's primary key:

```
DB::table('users')->where('active', false)
    ->chunkById(100, function ($users) {
        foreach ($users as $user) {
            DB::table('users')
                ->where('id', $user->id)
                ->update(['active' => true]);
        }
    });
```

{note} When updating or deleting records inside the chunk callback, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the chunked results.

Streaming Results Lazily

The `lazy` method works similarly to the chunk method in the sense that it executes the query in chunks. However, instead of passing each chunk into a callback, the `lazy()` method returns a `LazyCollection`, which lets you interact with the results as a single stream:

```
use Illuminate\Support\Facades\DB;

DB::table('users')->orderBy('id')->lazy()->each(function ($user) {
    //
});
```

Once again, if you plan to update the retrieved records while iterating over them, it is best to use the `lazyById` or `lazyByIdDesc` methods instead. These methods will automatically paginate the results based on the record's primary key:

```
DB::table('users')->where('active', false)
    ->lazyById()->each(function ($user) {
        DB::table('users')
            ->where('id', $user->id)
            ->update(['active' => true]);
    });
```

{note} When updating or deleting records while iterating over them, any changes to the primary key or foreign keys could affect the chunk query. This could potentially result in records not being included in the results.

Aggregates

The query builder also provides a variety of methods for retrieving aggregate values like `count`, `max`, `min`, `avg`, and `sum`. You may call any of these methods after constructing your query:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Of course, you may combine these methods with other clauses to fine-tune how your aggregate value is calculated:

```
$price = DB::table('orders')
                ->where('finalized', 1)
                ->avg('price');
```

Determining If Records Exist

Instead of using the `count` method to determine if any records exist that match your

query's constraints, you may use the `exists` and `doesntExist` methods:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {
    // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
    // ...
}
```

Select Statements

Specifying A Select Clause

You may not always want to select all columns from a database table. Using the `select` method, you can specify a custom "select" clause for the query:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->select('name', 'email as user_email')
            ->get();
```

The `distinct` method allows you to force the query to return distinct results:

```
$users = DB::table('users')->distinct()->get();
```

If you already have a query builder instance and you wish to add a column to its existing select clause, you may use the `addSelect` method:

```
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Raw Expressions

Sometimes you may need to insert an arbitrary string into a query. To create a raw string expression, you may use the `raw` method provided by the `DB` facade:

```
$users = DB::table('users')
             ->select(DB::raw('count(*) as user_count, status'))
             ->where('status', '<>', 1)
             ->groupBy('status')
             ->get();
```

{note} Raw statements will be injected into the query as strings, so you should be extremely careful to avoid creating SQL injection vulnerabilities.

Raw Methods

Instead of using the `DB::raw` method, you may also use the following methods to insert a raw expression into various parts of your query. **Remember, Laravel can not guarantee that any query using raw expressions is protected against SQL injection vulnerabilities.**

`selectRaw`

The `selectRaw` method can be used in place of `addSelect(DB::raw(...))`. This method accepts an optional array of bindings as its second argument:

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as price_with_tax', [1.0825])
                ->get();
```

`whereRaw / orWhereRaw`

The `whereRaw` and `orWhereRaw` methods can be used to inject a raw "where" clause into your query. These methods accept an optional array of bindings as their second argument:

```
$orders = DB::table('orders')
                ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
                ->get();
```

havingRaw / orHavingRaw

The `havingRaw` and `orHavingRaw` methods may be used to provide a raw string as the value of the "having" clause. These methods accept an optional array of bindings as their second argument:

```
$orders = DB::table('orders')
                ->select('department', DB::raw('SUM(price) as
total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

orderByRaw

The `orderByRaw` method may be used to provide a raw string as the value of the "order by" clause:

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at DESC')
                ->get();
```

groupByRaw

The `groupByRaw` method may be used to provide a raw string as the value of the `group by` clause:

```
$orders = DB::table('orders')
                ->select('city', 'state')
                ->groupByRaw('city, state')
                ->get();
```

Joins

Inner Join Clause

The query builder may also be used to add join clauses to your queries. To perform a basic "inner join", you may use the `join` method on a query builder instance. The first argument passed to the `join` method is the name of the table you need to join to, while the remaining arguments specify the column constraints for the join. You may even join multiple tables in a single query:

```
use Illuminate\Support\Facades\DB;

$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

Left Join / Right Join Clause

If you would like to perform a "left join" or "right join" instead of an "inner join", use the `leftJoin` or `rightJoin` methods. These methods have the same signature as the `join` method:

```
$users = DB::table('users')
            ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();

$users = DB::table('users')
            ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
            ->get();
```

Cross Join Clause

You may use the `crossJoin` method to perform a "cross join". Cross joins generate a cartesian product between the first table and the joined table:

```
$sizes = DB::table('sizes')
            ->crossJoin('colors')
            ->get();
```

Advanced Join Clauses

You may also specify more advanced join clauses. To get started, pass a closure as the second argument to the `join` method. The closure will receive a `Illuminate\Database\Query\JoinClause` instance which allows you to specify constraints on the "join" clause:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
        })
        ->get();
```

If you would like to use a "where" clause on your joins, you may use the `where` and `orWhere` methods provided by the `JoinClause` instance. Instead of comparing two columns, these methods will compare the column against a value:

```
DB::table('users')
        ->join('contacts', function ($join) {
            $join->on('users.id', '=', 'contacts.user_id')
                 ->where('contacts.user_id', '>', 5);
        })
        ->get();
```

Subquery Joins

You may use the `joinSub`, `leftJoinSub`, and `rightJoinSub` methods to join a query to a subquery. Each of these methods receives three arguments: the subquery, its table alias, and a closure that defines the related columns. In this example, we will retrieve a collection of users where each user record also contains the `created_at` timestamp of the user's most recently published blog post:

```
$latestPosts = DB::table('posts')
                   ->select('user_id', DB::raw('MAX(created_at) as
last_post_created_at'))
                   ->where('is_published', true)
                   ->groupBy('user_id');

$users = DB::table('users')
        ->joinSub($latestPosts, 'latest_posts', function ($join) {
            $join->on('users.id', '=', 'latest_posts.user_id');
        })->get();
```

Unions

The query builder also provides a convenient method to "union" two or more queries together. For example, you may create an initial query and use the `union` method to union it with more queries:

```
use Illuminate\Support\Facades\DB;

$first = DB::table('users')
            ->whereNull('first_name');

$users = DB::table('users')
            ->whereNull('last_name')
            ->union($first)
            ->get();
```

In addition to the `union` method, the query builder provides a `unionAll` method. Queries that are combined using the `unionAll` method will not have their duplicate results removed. The `unionAll` method has the same method signature as the `union` method.

Basic Where Clauses

Where Clauses

You may use the query builder's `where` method to add "where" clauses to the query. The most basic call to the `where` method requires three arguments. The first argument is the name of the column. The second argument is an operator, which can be any of the database's supported operators. The third argument is the value to compare against the column's value.

For example, the following query retrieves users where the value of the `votes` column is equal to `100` and the value of the `age` column is greater than `35`:

```
$users = DB::table('users')
                ->where('votes', '=', 100)
                ->where('age', '>', 35)
                ->get();
```

For convenience, if you want to verify that a column is `=` to a given value, you may pass the value as the second argument to the `where` method. Laravel will assume you would like to use the `=` operator:

```
$users = DB::table('users')->where('votes', 100)->get();
```

As previously mentioned, you may use any operator that is supported by your database system:

```
$users = DB::table('users')
                ->where('votes', '>=', 100)
                ->get();

$users = DB::table('users')
                ->where('votes', '<>', 100)
                ->get();

$users = DB::table('users')
                ->where('name', 'like', 'T%')
                ->get();
```

You may also pass an array of conditions to the `where` function. Each element of the array should be an array containing the three arguments typically passed to the `where` method:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

{note} PDO does not support binding column names. Therefore, you should never allow user input to dictate the column names referenced by your queries, including "order by" columns.

Or Where Clauses

When chaining together calls to the query builder's `where` method, the "where" clauses will be joined together using the `and` operator. However, you may use the `orWhere` method to join a clause to the query using the `or` operator. The `orWhere` method accepts the same arguments as the `where` method:

```
$users = DB::table('users')
                    ->where('votes', '>', 100)
                    ->orWhere('name', 'John')
                    ->get();
```

If you need to group an "or" condition within parentheses, you may pass a closure as the first argument to the `orWhere` method:

```
$users = DB::table('users')
            ->where('votes', '>', 100)
            ->orWhere(function($query) {
                $query->where('name', 'Abigail')
                      ->where('votes', '>', 50);
            })
            ->get();
```

The example above will produce the following SQL:

```
select * from users where votes > 100 or (name = 'Abigail' and votes >
50)
```

{note} You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

JSON Where Clauses

Laravel also supports querying JSON column types on databases that provide support for JSON column types. Currently, this includes MySQL 5.7+, PostgreSQL, SQL Server 2016, and SQLite 3.9.0 (with the JSON1 extension). To query a JSON column, use the `->` operator:

```
$users = DB::table('users')
                ->where('preferences->dining->meal', 'salad')
                ->get();
```

You may use `whereJsonContains` to query JSON arrays. This feature is not supported by the SQLite database:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', 'en')
                ->get();
```

If your application uses the MySQL or PostgreSQL databases, you may pass an array of values to the `whereJsonContains` method:

```
$users = DB::table('users')
                ->whereJsonContains('options->languages', ['en', 'de'])
                ->get();
```

You may use `whereJsonLength` method to query JSON arrays by their length:

```
$users = DB::table('users')
                ->whereJsonLength('options->languages', 0)
                ->get();

$users = DB::table('users')
                ->whereJsonLength('options->languages', '>', 1)
                ->get();
```

Additional Where Clauses

whereBetween / orWhereBetween

The `whereBetween` method verifies that a column's value is between two values:

```
$users = DB::table('users')
           ->whereBetween('votes', [1, 100])
           ->get();
```

whereNotBetween / orWhereNotBetween

The `whereNotBetween` method verifies that a column's value lies outside of two values:

```
$users = DB::table('users')
                    ->whereNotBetween('votes', [1, 100])
                    ->get();
```

whereIn / whereNotIn / orWhereIn / orWhereNotIn

The `whereIn` method verifies that a given column's value is contained within the given array:

```
$users = DB::table('users')
                    ->whereIn('id', [1, 2, 3])
                    ->get();
```

The `whereNotIn` method verifies that the given column's value is not contained in the given array:

```
$users = DB::table('users')
                    ->whereNotIn('id', [1, 2, 3])
                    ->get();
```

{note} If you are adding a large array of integer bindings to your query, the `whereIntegerInRaw` or `whereIntegerNotInRaw` methods may be used to greatly reduce your memory usage.

whereNull / whereNotNull / orWhereNull / orWhereNotNull

The `whereNull` method verifies that the value of the given column is `NULL`:

```
$users = DB::table('users')
                ->whereNull('updated_at')
                ->get();
```

The `whereNotNull` method verifies that the column's value is not `NULL`:

```
$users = DB::table('users')
                ->whereNotNull('updated_at')
                ->get();
```

whereDate / whereMonth / whereDay / whereYear / whereTime

The `whereDate` method may be used to compare a column's value against a date:

```
$users = DB::table('users')
                ->whereDate('created_at', '2016-12-31')
                ->get();
```

The `whereMonth` method may be used to compare a column's value against a specific month:

```
$users = DB::table('users')
                ->whereMonth('created_at', '12')
                ->get();
```

The `whereDay` method may be used to compare a column's value against a specific day of the month:

```
$users = DB::table('users')
                ->whereDay('created_at', '31')
                ->get();
```

The `whereYear` method may be used to compare a column's value against a specific year:

```
$users = DB::table('users')
                ->whereYear('created_at', '2016')
                ->get();
```

The `whereTime` method may be used to compare a column's value against a specific time:

```
$users = DB::table('users')
                ->whereTime('created_at', '=', '11:20:45')
                ->get();
```

whereColumn / orWhereColumn

The `whereColumn` method may be used to verify that two columns are equal:

```
$users = DB::table('users')
                ->whereColumn('first_name', 'last_name')
                ->get();
```

You may also pass a comparison operator to the `whereColumn` method:

```
$users = DB::table('users')
                ->whereColumn('updated_at', '>', 'created_at')
                ->get();
```

You may also pass an array of column comparisons to the `whereColumn` method. These conditions will be joined using the `and` operator:

```
$users = DB::table('users')
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at'],
                ])->get();
```

Logical Grouping

Sometimes you may need to group several "where" clauses within parentheses in order to achieve your query's desired logical grouping. In fact, you should generally always group calls to the `orWhere` method in parentheses in order to avoid unexpected query behavior. To accomplish this, you may pass a closure to the `where` method:

```
$users = DB::table('users')
           ->where('name', '=', 'John')
           ->where(function ($query) {
               $query->where('votes', '>', 100)
                     ->orWhere('title', '=', 'Admin');
           })
           ->get();
```

As you can see, passing a closure into the `where` method instructs the query builder to begin a constraint group. The closure will receive a query builder instance which you can use

to set the constraints that should be contained within the parenthesis group. The example above will produce the following SQL:

```
select * from users where name = 'John' and (votes > 100 or title =
'Admin')
```

{note} You should always group `orWhere` calls in order to avoid unexpected behavior when global scopes are applied.

Advanced Where Clauses

Where Exists Clauses

The `whereExists` method allows you to write "where exists" SQL clauses. The `whereExists` method accepts a closure which will receive a query builder instance, allowing you to define the query that should be placed inside of the "exists" clause:

```
$users = DB::table('users')
           ->whereExists(function ($query) {
               $query->select(DB::raw(1))
                     ->from('orders')
                     ->whereColumn('orders.user_id', 'users.id');
           })
           ->get();
```

The query above will produce the following SQL:

```
select * from users
where exists (
    select 1
    from orders
    where orders.user_id = users.id
)
```

Subquery Where Clauses

Sometimes you may need to construct a "where" clause that compares the results of a subquery to a given value. You may accomplish this by passing a closure and a value to the `where` method. For example, the following query will retrieve all users who have a recent "membership" of a given type;

```
use App\Models\User;

$users = User::where(function ($query) {
    $query->select('type')
        ->from('membership')
        ->whereColumn('membership.user_id', 'users.id')
        ->orderByDesc('membership.start_date')
        ->limit(1);
}, 'Pro')->get();
```

Or, you may need to construct a "where" clause that compares a column to the results of a subquery. You may accomplish this by passing a column, operator, and closure to the `where` method. For example, the following query will retrieve all income records where the amount is less than average;

```
use App\Models\Income;

$incomes = Income::where('amount', '<', function ($query) {
    $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

Ordering, Grouping, Limit & Offset

Ordering

The `orderBy` Method

The `orderBy` method allows you to sort the results of the query by a given column. The first argument accepted by the `orderBy` method should be the column you wish to sort by, while the second argument determines the direction of the sort and may be either `asc` or `desc`:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->get();
```

To sort by multiple columns, you may simply invoke `orderBy` as many times as necessary:

```
$users = DB::table('users')
                ->orderBy('name', 'desc')
                ->orderBy('email', 'asc')
                ->get();
```

The `latest` & `oldest` Methods

The `latest` and `oldest` methods allow you to easily order results by date. By default, the result will be ordered by the table's `created_at` column. Or, you may pass the column name that you wish to sort by:

```
$user = DB::table('users')
                ->latest()
                ->first();
```

Random Ordering

The `inRandomOrder` method may be used to sort the query results randomly. For example, you may use this method to fetch a random user:

```
$randomUser = DB::table('users')
                ->inRandomOrder()
                ->first();
```

Removing Existing Orderings

The `reorder` method removes all of the "order by" clauses that have previously been applied to the query:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

You may pass a column and direction when calling the `reorder` method in order to remove all existing "order by" clauses and apply an entirely new order to the query:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

Grouping

The `groupBy` & `having` Methods

As you might expect, the `groupBy` and `having` methods may be used to group the query results. The `having` method's signature is similar to that of the `where` method:

```
$users = DB::table('users')
                ->groupBy('account_id')
                ->having('account_id', '>', 100)
                ->get();
```

You can use the `havingBetween` method to filter the results within a given range:

```
$report = DB::table('orders')
                ->selectRaw('count(id) as number_of_orders,
customer_id')
                ->groupBy('customer_id')
                ->havingBetween('number_of_orders', [5, 15])
                ->get();
```

You may pass multiple arguments to the `groupBy` method to group by multiple columns:

```
$users = DB::table('users')
                ->groupBy('first_name', 'status')
                ->having('account_id', '>', 100)
                ->get();
```

To build more advanced `having` statements, see the `havingRaw` method.

Limit & Offset

The `skip` & `take` Methods

You may use the `skip` and `take` methods to limit the number of results returned from the query or to skip a given number of results in the query:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Alternatively, you may use the `limit` and `offset` methods. These methods are functionally equivalent to the `take` and `skip` methods, respectively:

```
$users = DB::table('users')
                ->offset(10)
                ->limit(5)
                ->get();
```

Conditional Clauses

Sometimes you may want certain query clauses to apply to a query based on another condition. For instance, you may only want to apply a `where` statement if a given input value is present on the incoming HTTP request. You may accomplish this using the `when` method:

```
$role = $request->input('role');

$users = DB::table('users')
                ->when($role, function ($query, $role) {
                    return $query->where('role_id', $role);
                })
                ->get();
```

The `when` method only executes the given closure when the first argument is `true`. If the first argument is `false`, the closure will not be executed. So, in the example above, the closure given to the `when` method will only be invoked if the `role` field is present on the incoming request and evaluates to `true`.

You may pass another closure as the third argument to the `when` method. This closure will only execute if the first argument evaluates as `false`. To illustrate how this feature may be used, we will use it to configure the default ordering of a query:

```
$sortByVotes = $request->input('sort_by_votes');

$users = DB::table('users')
                ->when($sortByVotes, function ($query, $sortByVotes) {
                    return $query->orderBy('votes');
                }, function ($query) {
                    return $query->orderBy('name');
                })
                ->get();
```

Insert Statements

The query builder also provides an `insert` method that may be used to insert records into the database table. The `insert` method accepts an array of column names and values:

```
DB::table('users')->insert([
    'email' => 'kayla@example.com',
    'votes' => 0
]);
```

You may insert several records at once by passing an array of arrays. Each array represents a record that should be inserted into the table:

```
DB::table('users')->insert([
    ['email' => 'picard@example.com', 'votes' => 0],
    ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

The `insertOrIgnore` method will ignore errors while inserting records into the database:

```
DB::table('users')->insertOrIgnore([
    ['id' => 1, 'email' => 'sisko@example.com'],
    ['id' => 2, 'email' => 'archer@example.com'],
]);
```

{note} `insertOrIgnore` will ignore duplicate records and also may ignore other types of errors depending on the database engine. For example, `insertOrIgnore` will bypass MySQL's strict mode.

Auto-Incrementing IDs

If the table has an auto-incrementing id, use the `insertGetId` method to insert a record and then retrieve the ID:

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

{note} When using PostgreSQL the `insertGetId` method expects the auto-incrementing column to be named `id`. If you would like to retrieve the ID from a different "sequence", you may pass the column name as the second parameter to the `insertGetId` method.

Upserts

The `upsert` method will insert records that do not exist and update the records that already exist with new values that you may specify. The method's first argument consists of the values to insert or update, while the second argument lists the column(s) that uniquely identify records within the associated table. The method's third and final argument is an array of columns that should be updated if a matching record already exists in the database:

```
DB::table('flights')->upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' =>
99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' =>
150]
], ['departure', 'destination'], ['price']);
```

In the example above, Laravel will attempt to insert two records. If a record already exists with the same `departure` and `destination` column values, Laravel will update that record's `price` column.

{note} All databases except SQL Server require the columns in the second argument of the `upsert` method to have a "primary" or "unique" index. In addition, the MySQL database driver ignores the second argument of the `upsert` method and always uses the "primary" and "unique" indexes of the table to detect existing records.

Update Statements

In addition to inserting records into the database, the query builder can also update existing records using the `update` method. The `update` method, like the `insert` method, accepts an array of column and value pairs indicating the columns to be updated. You may constrain the `update` query using `where` clauses:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['votes' => 1]);
```

Update Or Insert

Sometimes you may want to update an existing record in the database or create it if no matching record exists. In this scenario, the `updateOrInsert` method may be used. The `updateOrInsert` method accepts two arguments: an array of conditions by which to find the record, and an array of column and value pairs indicating the columns to be updated.

The `updateOrInsert` method will attempt to locate a matching database record using the first argument's column and value pairs. If the record exists, it will be updated with the values in the second argument. If the record can not be found, a new record will be inserted with the merged attributes of both arguments:

```
DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

Updating JSON Columns

When updating a JSON column, you should use `->` syntax to update the appropriate key in the JSON object. This operation is supported on MySQL 5.7+ and PostgreSQL 9.5+:

```
$affected = DB::table('users')
              ->where('id', 1)
              ->update(['options->enabled' => true]);
```

Increment & Decrement

The query builder also provides convenient methods for incrementing or decrementing the value of a given column. Both of these methods accept at least one argument: the column to modify. A second argument may be provided to specify the amount by which the column should be incremented or decremented:

```
DB::table('users')->increment('votes');

DB::table('users')->increment('votes', 5);

DB::table('users')->decrement('votes');

DB::table('users')->decrement('votes', 5);
```

You may also specify additional columns to update during the operation:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Delete Statements

The query builder's `delete` method may be used to delete records from the table. You may constrain `delete` statements by adding "where" clauses before calling the `delete` method:

```
DB::table('users')->delete();

DB::table('users')->where('votes', '>', 100)->delete();
```

If you wish to truncate an entire table, which will remove all records from the table and reset the auto-incrementing ID to zero, you may use the `truncate` method:

```
DB::table('users')->truncate();
```

Table Truncation & PostgreSQL

When truncating a PostgreSQL database, the `CASCADE` behavior will be applied. This means that all foreign key related records in other tables will be deleted as well.

Pessimistic Locking

The query builder also includes a few functions to help you achieve "pessimistic locking" when executing your `select` statements. To execute a statement with a "shared lock", you may call the `sharedLock` method. A shared lock prevents the selected rows from being modified until your transaction is committed:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->sharedLock()
        ->get();
```

Alternatively, you may use the `lockForUpdate` method. A "for update" lock prevents the selected records from being modified or from being selected with another shared lock:

```
DB::table('users')
        ->where('votes', '>', 100)
        ->lockForUpdate()
        ->get();
```

Debugging

You may use the `dd` and `dump` methods while building a query to dump the current query bindings and SQL. The `dd` method will display the debug information and then stop executing the request. The `dump` method will display the debug information but allow the request to continue executing:

```
DB::table('users')->where('votes', '>', 100)->dd();

DB::table('users')->where('votes', '>', 100)->dump();
```

HTTP Requests

- Introduction
- Interacting With The Request
    - Accessing The Request
    - Request Path & Method
    - Request Headers
    - Request IP Address
    - Content Negotiation
    - PSR-7 Requests
- Input
    - Retrieving Input
    - Determining If Input Is Present
    - Old Input
    - Cookies
    - Input Trimming & Normalization
- Files
    - Retrieving Uploaded Files
    - Storing Uploaded Files
- Configuring Trusted Proxies
- Configuring Trusted Hosts

Introduction

Laravel's `Illuminate\Http\Request` class provides an object-oriented way to interact with the current HTTP request being handled by your application as well as retrieve the input, cookies, and files that were submitted with the request.

Interacting With The Request

Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the `Illuminate\Http\Request` class on your route closure or controller method. The incoming request instance will automatically be injected by the Laravel service container:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

As mentioned, you may also type-hint the `Illuminate\Http\Request` class on a route closure. The service container will automatically inject the incoming request into the closure when it is executed:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    //
});
```

Dependency Injection & Route Parameters

If your controller method is also expecting input from a route parameter you should list your route parameters after your other dependencies. For example, if your route is defined like so:

```
use App\Http\Controllers\UserController;

Route::put('/user/{id}', [UserController::class, 'update']);
```

You may still type-hint the `Illuminate\Http\Request` and access your `id` route parameter by defining your controller method as follows:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  string  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Request Path & Method

The `Illuminate\Http\Request` instance provides a variety of methods for examining the incoming HTTP request and extends the `Symfony\Component\HttpFoundation\Request` class. We will discuss a few of the most important methods below.

Retrieving The Request Path

The `path` method returns the request's path information. So, if the incoming request is targeted at `http://example.com/foo/bar`, the `path` method will return `foo/bar`:

```php
$uri = $request->path();
```

Inspecting The Request Path / Route

The `is` method allows you to verify that the incoming request path matches a given pattern.

You may use the `*` character as a wildcard when utilizing this method:

```
if ($request->is('admin/*')) {
    //
}
```

Using the `routeIs` method, you may determine if the incoming request has matched a named route:

```
if ($request->routeIs('admin.*')) {
    //
}
```

Retrieving The Request URL

To retrieve the full URL for the incoming request you may use the `url` or `fullUrl` methods. The `url` method will return the URL without the query string, while the `fullUrl` method includes the query string:

```
$url = $request->url();

$urlWithQueryString = $request->fullUrl();
```

If you would like to append query string data to the current URL, you may call the `fullUrlWithQuery` method. This method merges the given array of query string variables with the current query string:

```
$request->fullUrlWithQuery(['type' => 'phone']);
```

Retrieving The Request Method

The `method` method will return the HTTP verb for the request. You may use the `isMethod` method to verify that the HTTP verb matches a given string:

```
$method = $request->method();

if ($request->isMethod('post')) {
    //
}
```

Request Headers

You may retrieve a request header from the `Illuminate\Http\Request` instance using the `header` method. If the header is not present on the request, `null` will be returned. However, the `header` method accepts an optional second argument that will be returned if the header is not present on the request:

```
$value = $request->header('X-Header-Name');

$value = $request->header('X-Header-Name', 'default');
```

The `hasHeader` method may be used to determine if the request contains a given header:

```
if ($request->hasHeader('X-Header-Name')) {
    //
}
```

For convenience, the `bearerToken` method may be used to retrieve a bearer token from the `Authorization` header. If no such header is present, an empty string will be returned:

```
$token = $request->bearerToken();
```

Request IP Address

The `ip` method may be used to retrieve the IP address of the client that made the request to your application:

```
$ipAddress = $request->ip();
```

Content Negotiation

Laravel provides several methods for inspecting the incoming request's requested content types via the `Accept` header. First, the `getAcceptableContentTypes` method will return an array containing all of the content types accepted by the request:

```
$contentTypes = $request->getAcceptableContentTypes();
```

The `accepts` method accepts an array of content types and returns `true` if any of the content types are accepted by the request. Otherwise, `false` will be returned:

```
if ($request->accepts(['text/html', 'application/json'])) {
    // ...
}
```

You may use the `prefers` method to determine which content type out of a given array of content types is most preferred by the request. If none of the provided content types are accepted by the request, `null` will be returned:

```
$preferred = $request->prefers(['text/html', 'application/json']);
```

Since many applications only serve HTML or JSON, you may use the `expectsJson` method to quickly determine if the incoming request expects a JSON response:

```
if ($request->expectsJson()) {
    // ...
}
```

PSR-7 Requests

The PSR-7 standard specifies interfaces for HTTP messages, including requests and responses. If you would like to obtain an instance of a PSR-7 request instead of a Laravel request, you will first need to install a few libraries. Laravel uses the *Symfony HTTP Message Bridge* component to convert typical Laravel requests and responses into PSR-7 compatible implementations:

```
composer require symfony/psr-http-message-bridge
composer require nyholm/psr7
```

Once you have installed these libraries, you may obtain a PSR-7 request by type-hinting the request interface on your route closure or controller method:

```
use Psr\Http\Message\ServerRequestInterface;

Route::get('/', function (ServerRequestInterface $request) {
    //
});
```

{tip} If you return a PSR-7 response instance from a route or controller, it will automatically be converted back to a Laravel response instance and be displayed by the framework.

Input

Retrieving Input

Retrieving All Input Data

You may retrieve all of the incoming request's input data as an `array` using the `all` method. This method may be used regardless of whether the incoming request is from an HTML form or is an XHR request:

```
$input = $request->all();
```

Using the `collect` method, you may retrieve all of the incoming request's input data as a collection:

```
$input = $request->collect();
```

The `collect` method also allows you to retrieve a subset of the incoming request input as a collection:

```
$request->collect('users')->each(function ($user) {
    // ...
});
```

Retrieving An Input Value

Using a few simple methods, you may access all of the user input from your `Illuminate\Http\Request` instance without worrying about which HTTP verb was used for the request. Regardless of the HTTP verb, the `input` method may be used to retrieve user input:

```
$name = $request->input('name');
```

You may pass a default value as the second argument to the `input` method. This value will be returned if the requested input value is not present on the request:

```
$name = $request->input('name', 'Sally');
```

When working with forms that contain array inputs, use "dot" notation to access the arrays:

```
$name = $request->input('products.0.name');

$names = $request->input('products.*.name');
```

You may call the `input` method without any arguments in order to retrieve all of the input values as an associative array:

```
$input = $request->input();
```

Retrieving Input From The Query String

While the `input` method retrieves values from the entire request payload (including the query string), the `query` method will only retrieve values from the query string:

```
$name = $request->query('name');
```

If the requested query string value data is not present, the second argument to this method will be returned:

```
$name = $request->query('name', 'Helen');
```

You may call the `query` method without any arguments in order to retrieve all of the query string values as an associative array:

```
$query = $request->query();
```

Retrieving JSON Input Values

When sending JSON requests to your application, you may access the JSON data via the `input` method as long as the `Content-Type` header of the request is properly set to `application/json`. You may even use "dot" syntax to retrieve values that are nested within JSON arrays:

```
$name = $request->input('user.name');
```

Retrieving Boolean Input Values

When dealing with HTML elements like checkboxes, your application may receive "truthy" values that are actually strings. For example, "true" or "on". For convenience, you may use the `boolean` method to retrieve these values as booleans. The `boolean` method returns `true` for 1, "1", true, "true", "on", and "yes". All other values will return `false`:

```
$archived = $request->boolean('archived');
```

Retrieving Input Via Dynamic Properties

You may also access user input using dynamic properties on the `Illuminate\Http\Request` instance. For example, if one of your application's forms contains a `name` field, you may access the value of the field like so:

```
$name = $request->name;
```

When using dynamic properties, Laravel will first look for the parameter's value in the request payload. If it is not present, Laravel will search for the field in the matched route's parameters.

Retrieving A Portion Of The Input Data

If you need to retrieve a subset of the input data, you may use the `only` and `except`

methods. Both of these methods accept a single `array` or a dynamic list of arguments:

```
$input = $request->only(['username', 'password']);

$input = $request->only('username', 'password');

$input = $request->except(['credit_card']);

$input = $request->except('credit_card');
```

{note} The `only` method returns all of the key / value pairs that you request; however, it will not return key / value pairs that are not present on the request.

Determining If Input Is Present

You may use the `has` method to determine if a value is present on the request. The `has` method returns `true` if the value is present on the request:

```
if ($request->has('name')) {
    //
}
```

When given an array, the `has` method will determine if all of the specified values are present:

```
if ($request->has(['name', 'email'])) {
    //
}
```

The `whenHas` method will execute the given closure if a value is present on the request:

```
$request->whenHas('name', function ($input) {
    //
});
```

A second closure may be passed to the `whenHas` method that will be executed if the specified value is not present on the request:

```
$request->whenHas('name', function ($input) {
    // The "name" value is present...
}, function () {
    // The "name" value is not present...
});
```

The `hasAny` method returns `true` if any of the specified values are present:

```
if ($request->hasAny(['name', 'email'])) {
    //
}
```

If you would like to determine if a value is present on the request and is not empty, you may use the `filled` method:

```
if ($request->filled('name')) {
    //
}
```

The `whenFilled` method will execute the given closure if a value is present on the request and is not empty:

```
$request->whenFilled('name', function ($input) {
    //
});
```

A second closure may be passed to the `whenFilled` method that will be executed if the specified value is not "filled":

```
$request->whenFilled('name', function ($input) {
    // The "name" value is filled...
}, function () {
    // The "name" value is not filled...
});
```

To determine if a given key is absent from the request, you may use the `missing` method:

```
if ($request->missing('name')) {
    //
}
```

Old Input

Laravel allows you to keep input from one request during the next request. This feature is particularly useful for re-populating forms after detecting validation errors. However, if you are using Laravel's included validation features, it is possible that you will not need to manually use these session input flashing methods directly, as some of Laravel's built-in validation facilities will call them automatically.

Flashing Input To The Session

The `flash` method on the `Illuminate\Http\Request` class will flash the current input to the session so that it is available during the user's next request to the application:

```
$request->flash();
```

You may also use the `flashOnly` and `flashExcept` methods to flash a subset of the request data to the session. These methods are useful for keeping sensitive information such as passwords out of the session:

```
$request->flashOnly(['username', 'email']);

$request->flashExcept('password');
```

Flashing Input Then Redirecting

Since you often will want to flash input to the session and then redirect to the previous page, you may easily chain input flashing onto a redirect using the `withInput` method:

```
return redirect('form')->withInput();

return redirect()->route('user.create')->withInput();

return redirect('form')->withInput(
    $request->except('password')
);
```

Retrieving Old Input

To retrieve flashed input from the previous request, invoke the `old` method on an instance of `Illuminate\Http\Request`. The `old` method will pull the previously flashed input data from the session:

```
$username = $request->old('username');
```

Laravel also provides a global `old` helper. If you are displaying old input within a Blade template, it is more convenient to use the `old` helper to repopulate the form. If no old input exists for the given field, `null` will be returned:

```
<input type="text" name="username" value="{{ old('username') }}">
```

Cookies

Retrieving Cookies From Requests

All cookies created by the Laravel framework are encrypted and signed with an authentication code, meaning they will be considered invalid if they have been changed by the client. To retrieve a cookie value from the request, use the `cookie` method on an `Illuminate\Http\Request` instance:

```
$value = $request->cookie('name');
```

Input Trimming & Normalization

By default, Laravel includes the `App\Http\Middleware\TrimStrings` and `App\Http\Middleware\ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middleware are listed in the global middleware stack by the `App\Http\Kernel` class. These middleware will automatically trim all incoming string fields on the request, as well as convert any empty string fields to `null`. This allows you to not have to worry about these normalization concerns in your routes and controllers.

If you would like to disable this behavior, you may remove the two middleware from your application's middleware stack by removing them from the `$middleware` property of your `App\Http\Kernel` class.

Files

Retrieving Uploaded Files

You may retrieve uploaded files from an `Illuminate\Http\Request` instance using the `file` method or using dynamic properties. The `file` method returns an instance of the `Illuminate\Http\UploadedFile` class, which extends the PHP `SplFileInfo` class and provides a variety of methods for interacting with the file:

```
$file = $request->file('photo');

$file = $request->photo;
```

You may determine if a file is present on the request using the `hasFile` method:

```
if ($request->hasFile('photo')) {
    //
}
```

Validating Successful Uploads

In addition to checking if the file is present, you may verify that there were no problems uploading the file via the `isValid` method:

```
if ($request->file('photo')->isValid()) {
    //
}
```

File Paths & Extensions

The `UploadedFile` class also contains methods for accessing the file's fully-qualified path and its extension. The `extension` method will attempt to guess the file's extension based on its contents. This extension may be different from the extension that was supplied by the client:

```
$path = $request->photo->path();

$extension = $request->photo->extension();
```

Other File Methods

There are a variety of other methods available on `UploadedFile` instances. Check out the API documentation for the class for more information regarding these methods.

Storing Uploaded Files

To store an uploaded file, you will typically use one of your configured filesystems. The `UploadedFile` class has a `store` method that will move an uploaded file to one of your disks, which may be a location on your local filesystem or a cloud storage location like Amazon S3.

The `store` method accepts the path where the file should be stored relative to the filesystem's configured root directory. This path should not contain a filename, since a unique ID will automatically be generated to serve as the filename.

The `store` method also accepts an optional second argument for the name of the disk that should be used to store the file. The method will return the path of the file relative to the disk's root:

```
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

If you do not want a filename to be automatically generated, you may use the `storeAs` method, which accepts the path, filename, and disk name as its arguments:

```
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

{tip} For more information about file storage in Laravel, check out the complete file storage documentation.

Configuring Trusted Proxies

When running your applications behind a load balancer that terminates TLS / SSL certificates, you may notice your application sometimes does not generate HTTPS links when using the `url` helper. Typically this is because your application is being forwarded traffic from your load balancer on port 80 and does not know it should generate secure links.

To solve this, you may use the `App\Http\Middleware\TrustProxies` middleware that is included in your Laravel application, which allows you to quickly customize the load balancers or proxies that should be trusted by your application. Your trusted proxies should be listed as an array on the `$proxies` property of this middleware. In addition to configuring the trusted proxies, you may configure the proxy `$headers` that should be trusted:

```
<?php

namespace App\Http\Middleware;

use Illuminate\Http\Middleware\TrustProxies as Middleware;
use Illuminate\Http\Request;

class TrustProxies extends Middleware
{
    /**
     * The trusted proxies for this application.
     *
     * @var string|array
     */
    protected $proxies = [
        '192.168.1.1',
        '192.168.1.2',
    ];

    /**
     * The headers that should be used to detect proxies.
     *
     * @var int
     */
    protected $headers = Request::HEADER_X_FORWARDED_FOR |
Request::HEADER_X_FORWARDED_HOST | Request::HEADER_X_FORWARDED_PORT |
Request::HEADER_X_FORWARDED_PROTO;
}
```

{tip} If you are using AWS Elastic Load Balancing, your `$headers` value should be `Request::HEADER_X_FORWARDED_AWS_ELB`. For more information on the constants that may be used in the `$headers` property, check out Symfony's documentation on trusting proxies.

Trusting All Proxies

If you are using Amazon AWS or another "cloud" load balancer provider, you may not know the IP addresses of your actual balancers. In this case, you may use `*` to trust all proxies:

```
/**
 * The trusted proxies for this application.
 *
 * @var string|array
 */
protected $proxies = '*';
```

Configuring Trusted Hosts

By default, Laravel will respond to all requests it receives regardless of the content of the HTTP request's `Host` header. In addition, the `Host` header's value will be used when generating absolute URLs to your application during a web request.

Typically, you should configure your web server, such as Nginx or Apache, to only send requests to your application that match a given host name. However, if you do not have the ability to customize your web server directly and need to instruct Laravel to only respond to certain host names, you may do so by enabling the `App\Http\Middleware\TrustHosts` middleware for your application.

The `TrustHosts` middleware is already included in the `$middleware` stack of your application; however, you should uncomment it so that it becomes active. Within this middleware's `hosts` method, you may specify the host names that your application should respond to. Incoming requests with other `Host` value headers will be rejected:

```
/**
 * Get the host patterns that should be trusted.
 *
 * @return array
 */
public function hosts()
{
    return [
        'laravel.test',
        $this->allSubdomainsOfApplicationUrl(),
    ];
}
```

The `allSubdomainsOfApplicationUrl` helper method will return a regular expression matching all subdomains of your application's `app.url` configuration value. This helper method provides a convenient way to allow all of your application's subdomains when building an application that utilizes wildcard subdomains.

HTTP Responses

- Creating Responses
    - Attaching Headers To Responses

    - Attaching Cookies To Responses
    - Cookies & Encryption
- Redirects
    - Redirecting To Named Routes
    - Redirecting To Controller Actions
    - Redirecting To External Domains
    - Redirecting With Flashed Session Data
- Other Response Types
    - View Responses
    - JSON Responses
    - File Downloads
    - File Responses
- Response Macros

Creating Responses

Strings & Arrays

All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is returning a string from a route or controller. The framework will automatically convert the string into a full HTTP response:

```
Route::get('/', function () {
    return 'Hello World';
});
```

In addition to returning strings from your routes and controllers, you may also return arrays. The framework will automatically convert the array into a JSON response:

```
Route::get('/', function () {
    return [1, 2, 3];
});
```

{tip} Did you know you can also return Eloquent collections from your routes or controllers? They will automatically be converted to JSON. Give it a shot!

Response Objects

Typically, you won't just be returning simple strings or arrays from your route actions. Instead, you will be returning full `Illuminate\Http\Response` instances or views.

Returning a full `Response` instance allows you to customize the response's HTTP status code and headers. A `Response` instance inherits from the `Symfony\Component\HttpFoundation\Response` class, which provides a variety of methods for building HTTP responses:

```
Route::get('/home', function () {
    return response('Hello World', 200)
                  ->header('Content-Type', 'text/plain');
});
```

Eloquent Models & Collections

You may also return Eloquent ORM models and collections directly from your routes and controllers. When you do, Laravel will automatically convert the models and collections to JSON responses while respecting the model's hidden attributes:

```
use App\Models\User;

Route::get('/user/{user}', function (User $user) {
    return $user;
});
```

Attaching Headers To Responses

Keep in mind that most response methods are chainable, allowing for the fluent construction of response instances. For example, you may use the `header` method to add a series of headers to the response before sending it back to the user:

```
return response($content)
            ->header('Content-Type', $type)
            ->header('X-Header-One', 'Header Value')
            ->header('X-Header-Two', 'Header Value');
```

Or, you may use the `withHeaders` method to specify an array of headers to be added to the response:

```
return response($content)
            ->withHeaders([
                'Content-Type' => $type,
                'X-Header-One' => 'Header Value',
                'X-Header-Two' => 'Header Value',
            ]);
```

Cache Control Middleware

Laravel includes a `cache.headers` middleware, which may be used to quickly set the `Cache-Control` header for a group of routes. Directives should be provided using the "snake case" equivalent of the corresponding cache-control directive and should be separated by a semicolon. If `etag` is specified in the list of directives, an MD5 hash of the response content will automatically be set as the ETag identifier:

```
Route::middleware('cache.headers:public;max_age=2628000;etag')->group(fu
nction () {
    Route::get('/privacy', function () {
        // ...
    });

    Route::get('/terms', function () {
        // ...
    });
});
```

Attaching Cookies To Responses

You may attach a cookie to an outgoing `Illuminate\Http\Response` instance using the `cookie` method. You should pass the name, value, and the number of minutes the cookie should be considered valid to this method:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes
);
```

The `cookie` method also accepts a few more arguments which are used less frequently.

Generally, these arguments have the same purpose and meaning as the arguments that would be given to PHP's native setcookie method:

```
return response('Hello World')->cookie(
    'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
);
```

If you would like to ensure that a cookie is sent with the outgoing response but you do not yet have an instance of that response, you can use the `Cookie` facade to "queue" cookies for attachment to the response when it is sent. The `queue` method accepts the arguments needed to create a cookie instance. These cookies will be attached to the outgoing response before it is sent to the browser:

```
use Illuminate\Support\Facades\Cookie;

Cookie::queue('name', 'value', $minutes);
```

Generating Cookie Instances

If you would like to generate a `Symfony\Component\HttpFoundation\Cookie` instance that can be attached to a response instance at a later time, you may use the global `cookie` helper. This cookie will not be sent back to the client unless it is attached to a response instance:

```
$cookie = cookie('name', 'value', $minutes);

return response('Hello World')->cookie($cookie);
```

Expiring Cookies Early

You may remove a cookie by expiring it via the `withoutCookie` method of an outgoing response:

```
return response('Hello World')->withoutCookie('name');
```

If you do not yet have an instance of the outgoing response, you may use the `Cookie` facade's `expire` method to expire a cookie:

```
Cookie::expire('name');
```

Cookies & Encryption

By default, all cookies generated by Laravel are encrypted and signed so that they can't be modified or read by the client. If you would like to disable encryption for a subset of cookies generated by your application, you may use the `$except` property of the `App\Http\Middleware\EncryptCookies` middleware, which is located in the `app/Http/Middleware` directory:

```
/**
 * The names of the cookies that should not be encrypted.
 *
 * @var array
 */
protected $except = [
    'cookie_name',
];
```

Redirects

Redirect responses are instances of the `Illuminate\Http\RedirectResponse` class, and contain the proper headers needed to redirect the user to another URL. There are several ways to generate a `RedirectResponse` instance. The simplest method is to use the global `redirect` helper:

```
Route::get('/dashboard', function () {
    return redirect('home/dashboard');
});
```

Sometimes you may wish to redirect the user to their previous location, such as when a submitted form is invalid. You may do so by using the global `back` helper function. Since this feature utilizes the session, make sure the route calling the `back` function is using the `web` middleware group:

```
Route::post('/user/profile', function () {
    // Validate the request...

    return back()->withInput();
});
```

Redirecting To Named Routes

When you call the `redirect` helper with no parameters, an instance of `Illuminate\Routing\Redirector` is returned, allowing you to call any method on the `Redirector` instance. For example, to generate a `RedirectResponse` to a named route, you may use the `route` method:

```
return redirect()->route('login');
```

If your route has parameters, you may pass them as the second argument to the `route` method:

```
// For a route with the following URI: /profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

Populating Parameters Via Eloquent Models

If you are redirecting to a route with an "ID" parameter that is being populated from an Eloquent model, you may pass the model itself. The ID will be extracted automatically:

```
// For a route with the following URI: /profile/{id}

return redirect()->route('profile', [$user]);
```

If you would like to customize the value that is placed in the route parameter, you can specify the column in the route parameter definition (`/profile/{id:slug}`) or you can override the `getRouteKey` method on your Eloquent model:

```
/**
 * Get the value of the model's route key.
 *
 * @return mixed
 */
public function getRouteKey()
{
    return $this->slug;
}
```

Redirecting To Controller Actions

You may also generate redirects to controller actions. To do so, pass the controller and action name to the `action` method:

```
use App\Http\Controllers\UserController;

return redirect()->action([UserController::class, 'index']);
```

If your controller route requires parameters, you may pass them as the second argument to the `action` method:

```
return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

Redirecting To External Domains

Sometimes you may need to redirect to a domain outside of your application. You may do so by calling the `away` method, which creates a `RedirectResponse` without any additional URL encoding, validation, or verification:

```
return redirect()->away('https://www.google.com');
```

Redirecting With Flashed Session Data

Redirecting to a new URL and flashing data to the session are usually done at the same time. Typically, this is done after successfully performing an action when you flash a success message to the session. For convenience, you may create a `RedirectResponse` instance and flash data to the session in a single, fluent method chain:

```
Route::post('/user/profile', function () {
    // ...

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

After the user is redirected, you may display the flashed message from the session. For example, using Blade syntax:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

Redirecting With Input

You may use the `withInput` method provided by the `RedirectResponse` instance to flash the current request's input data to the session before redirecting the user to a new location. This is typically done if the user has encountered a validation error. Once the input has been flashed to the session, you may easily retrieve it during the next request to repopulate the form:

```
return back()->withInput();
```

Other Response Types

The `response` helper may be used to generate other types of response instances. When the `response` helper is called without arguments, an implementation of the `Illuminate\Contracts\Routing\ResponseFactory` contract is returned. This contract provides several helpful methods for generating responses.

View Responses

If you need control over the response's status and headers but also need to return a view as the response's content, you should use the `view` method:

```
return response()
            ->view('hello', $data, 200)
            ->header('Content-Type', $type);
```

Of course, if you do not need to pass a custom HTTP status code or custom headers, you may use the global `view` helper function.

JSON Responses

The `json` method will automatically set the `Content-Type` header to `application/json`, as well as convert the given array to JSON using the `json_encode` PHP function:

```
return response()->json([
    'name' => 'Abigail',
    'state' => 'CA',
]);
```

If you would like to create a JSONP response, you may use the `json` method in combination with the `withCallback` method:

```
return response()
            ->json(['name' => 'Abigail', 'state' => 'CA'])
            ->withCallback($request->input('callback'));
```

File Downloads

The `download` method may be used to generate a response that forces the user's browser to download the file at the given path. The `download` method accepts a filename as the second argument to the method, which will determine the filename that is seen by the user downloading the file. Finally, you may pass an array of HTTP headers as the third argument to the method:

```
return response()->download($pathToFile);

return response()->download($pathToFile, $name, $headers);
```

{note} Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII filename.

Streamed Downloads

Sometimes you may wish to turn the string response of a given operation into a downloadable response without having to write the contents of the operation to disk. You may use the `streamDownload` method in this scenario. This method accepts a callback, filename, and an optional array of headers as its arguments:

```
use App\Services\GitHub;

return response()->streamDownload(function () {
    echo GitHub::api('repo')
                ->contents()
                ->readme('laravel', 'laravel')['contents'];
}, 'laravel-readme.md');
```

File Responses

The `file` method may be used to display a file, such as an image or PDF, directly in the user's browser instead of initiating a download. This method accepts the path to the file as its first argument and an array of headers as its second argument:

```
return response()->file($pathToFile);

return response()->file($pathToFile, $headers);
```

Response Macros

If you would like to define a custom response that you can re-use in a variety of your routes and controllers, you may use the `macro` method on the `Response` facade. Typically, you should call this method from the `boot` method of one of your application's service providers, such as the `App\Providers\AppServiceProvider` service provider:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Response::macro('caps', function ($value) {
            return Response::make(strtoupper($value));
        });
    }
}
```

The `macro` function accepts a name as its first argument and a closure as its second argument. The macro's closure will be executed when calling the macro name from a `ResponseFactory` implementation or the `response` helper:

```
return response()->caps('foo');
```

Routing

- Basic Routing
    - Redirect Routes
    - View Routes
- Route Parameters
    - Required Parameters
    - Optional Parameters
    - Regular Expression Constraints
- Named Routes
- Route Groups
    - Middleware
    - Subdomain Routing
    - Route Prefixes
    - Route Name Prefixes
- Route Model Binding
    - Implicit Binding
    - Explicit Binding
- Fallback Routes
- Rate Limiting
    - Defining Rate Limiters
    - Attaching Rate Limiters To Routes
- Form Method Spoofing
- Accessing The Current Route
- Cross-Origin Resource Sharing (CORS)
- Route Caching

Basic Routing

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files:

```
use Illuminate\Support\Facades\Route;

Route::get('/greeting', function () {
    return 'Hello World';
});
```

The Default Route Files

All Laravel routes are defined in your route files, which are located in the `routes` directory. These files are automatically loaded by your application's `App\Providers\RouteServiceProvider`. The `routes/web.php` file defines routes that are for your web interface. These routes are assigned the `web` middleware group, which provides features like session state and CSRF protection. The routes in `routes/api.php` are stateless and are assigned the `api` middleware group.

For most applications, you will begin by defining routes in your `routes/web.php` file. The routes defined in `routes/web.php` may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to `http://example.com/user` in your browser:

```
use App\Http\Controllers\UserController;

Route::get('/user', [UserController::class, 'index']);
```

Routes defined in the `routes/api.php` file are nested within a route group by the `RouteServiceProvider`. Within this group, the `/api` URI prefix is automatically applied so you do not need to manually apply it to every route in the file. You may modify the prefix and other route group options by modifying your `RouteServiceProvider` class.

Available Router Methods

The router allows you to register routes that respond to any HTTP verb:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the `match` method. Or, you may even register a route that responds to all HTTP verbs using the `any` method:

```
Route::match(['get', 'post'], '/', function () {
    //
});

Route::any('/', function () {
    //
});
```

{tip} When defining multiple routes that share the same URI, routes using the `get`, `post`, `put`, `patch`, `delete`, and `options` methods should be defined before routes using the `any`, `match`, and `redirect` methods. This ensures the incoming request is matched with the correct route.

Dependency Injection

You may type-hint any dependencies required by your route in your route's callback signature. The declared dependencies will automatically be resolved and injected into the callback by the Laravel service container. For example, you may type-hint the `Illuminate\Http\Request` class to have the current HTTP request automatically injected into your route callback:

```
use Illuminate\Http\Request;

Route::get('/users', function (Request $request) {
    // ...
});
```

CSRF Protection

Remember, any HTML forms pointing to `POST`, `PUT`, `PATCH`, or `DELETE` routes that are defined in the `web` routes file should include a CSRF token field. Otherwise, the request will be rejected. You can read more about CSRF protection in the CSRF documentation:

```
<form method="POST" action="/profile">
    @csrf
    ...
</form>
```

Redirect Routes

If you are defining a route that redirects to another URI, you may use the `Route::redirect` method. This method provides a convenient shortcut so that you do not have to define a full route or controller for performing a simple redirect:

```
Route::redirect('/here', '/there');
```

By default, `Route::redirect` returns a `302` status code. You may customize the status code using the optional third parameter:

```
Route::redirect('/here', '/there', 301);
```

Or, you may use the `Route::permanentRedirect` method to return a `301` status code:

```
Route::permanentRedirect('/here', '/there');
```

{note} When using route parameters in redirect routes, the following parameters are reserved by Laravel and cannot be used: `destination` and `status`.

View Routes

If your route only needs to return a view, you may use the `Route::view` method. Like the `redirect` method, this method provides a simple shortcut so that you do not have to define a full route or controller. The `view` method accepts a URI as its first argument and a view name as its second argument. In addition, you may provide an array of data to pass to the view as an optional third argument:

```
Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

{note} When using route parameters in view routes, the following parameters are reserved by Laravel and cannot be used: `view`, `data`, `status`, and `headers`.

Route Parameters

Required Parameters

Sometimes you will need to capture segments of the URI within your route. For example, you may need to capture a user's ID from the URL. You may do so by defining route parameters:

```
Route::get('/user/{id}', function ($id) {
    return 'User '.$id;
});
```

You may define as many route parameters as required by your route:

```
Route::get('/posts/{post}/comments/{comment}', function ($postId,
$commentId) {
    //
});
```

Route parameters are always encased within {} braces and should consist of alphabetic characters. Underscores (_) are also acceptable within route parameter names. Route parameters are injected into route callbacks / controllers based on their order - the names of the route callback / controller arguments do not matter.

Parameters & Dependency Injection

If your route has dependencies that you would like the Laravel service container to automatically inject into your route's callback, you should list your route parameters after your dependencies:

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, $id) {
    return 'User '.$id;
});
```

Optional Parameters

Occasionally you may need to specify a route parameter that may not always be present in the URI. You may do so by placing a `?` mark after the parameter name. Make sure to give the route's corresponding variable a default value:

```
Route::get('/user/{name?}', function ($name = null) {
    return $name;
});

Route::get('/user/{name?}', function ($name = 'John') {
    return $name;
});
```

Regular Expression Constraints

You may constrain the format of your route parameters using the `where` method on a route instance. The `where` method accepts the name of the parameter and a regular expression defining how the parameter should be constrained:

```
Route::get('/user/{name}', function ($name) {
    //
})->where('name', '[A-Za-z]+');

Route::get('/user/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

For convenience, some commonly used regular expression patterns have helper methods that allow you to quickly add pattern constraints to your routes:

```
Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->whereNumber('id')->whereAlpha('name');

Route::get('/user/{name}', function ($name) {
    //
})->whereAlphaNumeric('name');

Route::get('/user/{id}', function ($id) {
    //
})->whereUuid('id');
```

If the incoming request does not match the route pattern constraints, a 404 HTTP response will be returned.

Global Constraints

If you would like a route parameter to always be constrained by a given regular expression, you may use the `pattern` method. You should define these patterns in the `boot` method of your `App\Providers\RouteServiceProvider` class:

```
/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');
}
```

Once the pattern has been defined, it is automatically applied to all routes using that parameter name:

```
Route::get('/user/{id}', function ($id) {
    // Only executed if {id} is numeric...
});
```

Encoded Forward Slashes

The Laravel routing component allows all characters except `/` to be present within route parameter values. You must explicitly allow `/` to be part of your placeholder using a `where` condition regular expression:

```
Route::get('/search/{search}', function ($search) {
    return $search;
})->where('search', '.*');
```

{note} Encoded forward slashes are only supported within the last route segment.

Named Routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the `name` method onto the route definition:

```
Route::get('/user/profile', function () {
    //
})->name('profile');
```

You may also specify route names for controller actions:

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

{note} Route names should always be unique.

Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via Laravel's `route` and `redirect` helper functions:

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');
```

If the named route defines parameters, you may pass the parameters as the second argument to the `route` function. The given parameters will automatically be inserted into the generated URL in their correct positions:

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1]);
```

If you pass additional parameters in the array, those key / value pairs will automatically be added to the generated URL's query string:

```
Route::get('/user/{id}/profile', function ($id) {
    //
})->name('profile');

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

{tip} Sometimes, you may wish to specify request-wide default values for URL parameters, such as the current locale. To accomplish this, you may use the `URL::defaults` method.

Inspecting The Current Route

If you would like to determine if the current request was routed to a given named route, you may use the `named` method on a Route instance. For example, you may check the current route name from a route middleware:

```
/**
 * Handle an incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Closure  $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {
        //
    }

    return $next($request);
}
```

Route Groups

Route groups allow you to share route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and `where` conditions are merged while names and prefixes are appended. Namespace delimiters and slashes in URI prefixes are automatically added where appropriate.

Middleware

To assign middleware to all routes within a group, you may use the `middleware` method before defining the group. Middleware are executed in the order they are listed in the array:

```
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second middleware...
    });

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

Subdomain Routing

Route groups may also be used to handle subdomain routing. Subdomains may be assigned route parameters just like route URIs, allowing you to capture a portion of the subdomain for usage in your route or controller. The subdomain may be specified by calling the `domain` method before defining the group:

```
Route::domain('{account}.example.com')->group(function () {
    Route::get('user/{id}', function ($account, $id) {
        //
    });
});
```

{note} In order to ensure your subdomain routes are reachable, you should register subdomain routes before registering root domain routes. This will prevent root domain routes from overwriting subdomain routes which have the same URI path.

Route Prefixes

The `prefix` method may be used to prefix each route in the group with a given URI. For example, you may want to prefix all route URIs within the group with `admin`:

```
Route::prefix('admin')->group(function () {
    Route::get('/users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Route Name Prefixes

The `name` method may be used to prefix each route name in the group with a given string. For example, you may want to prefix all of the grouped route's names with `admin`. The given string is prefixed to the route name exactly as it is specified, so we will be sure to provide the trailing `.` character in the prefix:

```
Route::name('admin.')->group(function () {
    Route::get('/users', function () {
        // Route assigned name "admin.users"...
    })->name('users');
});
```

Route Model Binding

When injecting a model ID to a route or controller action, you will often query the database to retrieve the model that corresponds to that ID. Laravel route model binding provides a convenient way to automatically inject the model instances directly into your routes. For example, instead of injecting a user's ID, you can inject the entire `User` model instance that matches the given ID.

Implicit Binding

Laravel automatically resolves Eloquent models defined in routes or controller actions whose type-hinted variable names match a route segment name. For example:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
});
```

Since the `$user` variable is type-hinted as the `App\Models\User` Eloquent model and the variable name matches the `{user}` URI segment, Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI. If a matching model instance is not found in the database, a 404 HTTP response will automatically be generated.

Of course, implicit binding is also possible when using controller methods. Again, note the `{user}` URI segment matches the `$user` variable in the controller which contains an `App\Models\User` type-hint:

```
use App\Http\Controllers\UserController;
use App\Models\User;

// Route definition...
Route::get('/users/{user}', [UserController::class, 'show']);

// Controller method definition...
public function show(User $user)
{
    return view('user.profile', ['user' => $user]);
}
```

Soft Deleted Models

Typically, implicit model binding will not retrieve models that have been soft deleted. However, you may instruct the implicit binding to retrieve these models by chaining the `withTrashed` method onto your route's definition:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    return $user->email;
})->withTrashed();
```

Customizing The Key

Sometimes you may wish to resolve Eloquent models using a column other than `id`. To do so, you may specify the column in the route parameter definition:

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

If you would like model binding to always use a database column other than `id` when retrieving a given model class, you may override the `getRouteKeyName` method on the Eloquent model:

```
/**
 * Get the route key for the model.
 *
 * @return string
 */
public function getRouteKeyName()
{
    return 'slug';
}
```

Custom Keys & Scoping

When implicitly binding multiple Eloquent models in a single route definition, you may wish to scope the second Eloquent model such that it must be a child of the previous Eloquent model. For example, consider this route definition that retrieves a blog post by slug for a specific user:

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post
$post) {
    return $post;
});
```

When using a custom keyed implicit binding as a nested route parameter, Laravel will automatically scope the query to retrieve the nested model by its parent using conventions to guess the relationship name on the parent. In this case, it will be assumed that the `User` model has a relationship named `posts` (the plural form of the route parameter name) which can be used to retrieve the `Post` model.

If you wish, you may instruct Laravel to scope "child" bindings even when a custom key is not provided. To do so, you may invoke the `scopeBindings` method when defining your route:

```
use App\Models\Post;
use App\Models\User;

Route::get('/users/{user}/posts/{post}', function (User $user, Post
$post) {
    return $post;
})->scopeBindings();
```

Or, you may instruct an entire group of route definitions to use scoped bindings:

```
Route::scopeBindings()->group(function () {
    Route::get('/users/{user}/posts/{post}', function (User $user, Post
$post) {
        return $post;
    });
});
```

Customizing Missing Model Behavior

Typically, a 404 HTTP response will be generated if an implicitly bound model is not found. However, you may customize this behavior by calling the `missing` method when defining your route. The `missing` method accepts a closure that will be invoked if an implicitly bound model can not be found:

```
use App\Http\Controllers\LocationsController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::get('/locations/{location:slug}', [LocationsController::class,
'show'])
        ->name('locations.view')
        ->missing(function (Request $request) {
            return Redirect::route('locations.index');
        });
```

Explicit Binding

You are not required to use Laravel's implicit, convention based model resolution in order to use model binding. You can also explicitly define how route parameters correspond to models. To register an explicit binding, use the router's `model` method to specify the class for a given parameter. You should define your explicit model bindings at the beginning of the `boot` method of your `RouteServiceProvider` class:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::model('user', User::class);

    // ...
}
```

Next, define a route that contains a `{user}` parameter:

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    //
});
```

Since we have bound all `{user}` parameters to the `App\Models\User` model, an instance of that class will be injected into the route. So, for example, a request to `users/1` will inject the `User` instance from the database which has an ID of `1`.

If a matching model instance is not found in the database, a 404 HTTP response will be automatically generated.

Customizing The Resolution Logic

If you wish to define your own model binding resolution logic, you may use the `Route::bind` method. The closure you pass to the `bind` method will receive the value of the URI segment and should return the instance of the class that should be injected into the route. Again, this customization should take place in the `boot` method of your application's `RouteServiceProvider`:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::bind('user', function ($value) {
        return User::where('name', $value)->firstOrFail();
    });

    // ...
}
```

Alternatively, you may override the `resolveRouteBinding` method on your Eloquent model. This method will receive the value of the URI segment and should return the instance of the class that should be injected into the route:

```
/**
 * Retrieve the model for a bound value.
 *
 * @param  mixed  $value
 * @param  string|null  $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveRouteBinding($value, $field = null)
{
    return $this->where('name', $value)->firstOrFail();
}
```

If a route is utilizing implicit binding scoping, the `resolveChildRouteBinding` method

will be used to resolve the child binding of the parent model:

```
/**
 * Retrieve the child model for a bound value.
 *
 * @param  string  $childType
 * @param  mixed  $value
 * @param  string|null  $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveChildRouteBinding($childType, $value, $field)
{
    return parent::resolveChildRouteBinding($childType, $value, $field);
}
```

Fallback Routes

Using the `Route::fallback` method, you may define a route that will be executed when no other route matches the incoming request. Typically, unhandled requests will automatically render a "404" page via your application's exception handler. However, since you would typically define the `fallback` route within your `routes/web.php` file, all middleware in the `web` middleware group will apply to the route. You are free to add additional middleware to this route as needed:

```
Route::fallback(function () {
    //
});
```

{note} The fallback route should always be the last route registered by your application.

Rate Limiting

Defining Rate Limiters

Laravel includes powerful and customizable rate limiting services that you may utilize to restrict the amount of traffic for a given route or group of routes. To get started, you should define rate limiter configurations that meet your application's needs. Typically, this should be done within the `configureRateLimiting` method of your application's `App\Providers\RouteServiceProvider` class.

Rate limiters are defined using the `RateLimiter` facade's `for` method. The `for` method accepts a rate limiter name and a closure that returns the limit configuration that should apply to routes that are assigned to the rate limiter. Limit configuration are instances of the `Illuminate\Cache\RateLimiting\Limit` class. This class contains helpful "builder" methods so that you can quickly define your limit. The rate limiter name may be any string you wish:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Configure the rate limiters for the application.
 *
 * @return void
 */
protected function configureRateLimiting()
{
    RateLimiter::for('global', function (Request $request) {
        return Limit::perMinute(1000);
    });
}
```

If the incoming request exceeds the specified rate limit, a response with a 429 HTTP status code will automatically be returned by Laravel. If you would like to define your own response that should be returned by a rate limit, you may use the `response` method:

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000)->response(function () {
        return response('Custom response...', 429);
    });
});
```

Since rate limiter callbacks receive the incoming HTTP request instance, you may build the appropriate rate limit dynamically based on the incoming request or authenticated user:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100);
});
```

Segmenting Rate Limits

Sometimes you may wish to segment rate limits by some arbitrary value. For example, you may wish to allow users to access a given route 100 times per minute per IP address. To accomplish this, you may use the by method when building your rate limit:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
                ? Limit::none()
                : Limit::perMinute(100)->by($request->ip());
});
```

To illustrate this feature using another example, we can limit access to the route to 100 times per minute per authenticated user ID or 10 times per minute per IP address for guests:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()
                ? Limit::perMinute(100)->by($request->user()->id)
                : Limit::perMinute(10)->by($request->ip());
});
```

Multiple Rate Limits

If needed, you may return an array of rate limits for a given rate limiter configuration. Each rate limit will be evaluated for the route based on the order they are placed within the array:

```
RateLimiter::for('login', function (Request $request) {
    return [
        Limit::perMinute(500),
        Limit::perMinute(3)->by($request->input('email')),
    ];
});
```

Attaching Rate Limiters To Routes

Rate limiters may be attached to routes or route groups using the `throttle` middleware. The throttle middleware accepts the name of the rate limiter you wish to assign to the route:

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        //
    });

    Route::post('/video', function () {
        //
    });
});
```

Throttling With Redis

Typically, the `throttle` middleware is mapped to the `Illuminate\Routing\Middleware\ThrottleRequests` class. This mapping is defined in your application's HTTP kernel (`App\Http\Kernel`). However, if you are using Redis as your application's cache driver, you may wish to change this mapping to use the `Illuminate\Routing\Middleware\ThrottleRequestsWithRedis` class. This class is more efficient at managing rate limiting using Redis:

```
'throttle' =>
\Illuminate\Routing\Middleware\ThrottleRequestsWithRedis::class,
```

Form Method Spoofing

HTML forms do not support `PUT`, `PATCH`, or `DELETE` actions. So, when defining `PUT`, `PATCH`, or `DELETE` routes that are called from an HTML form, you will need to add a hidden `_method` field to the form. The value sent with the `_method` field will be used as the HTTP request method:

```
<form action="/example" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

For convenience, you may use the `@method` Blade directive to generate the `_method` input field:

```
<form action="/example" method="POST">
    @method('PUT')
    @csrf
</form>
```

Accessing The Current Route

You may use the `current`, `currentRouteName`, and `currentRouteAction` methods on the `Route` facade to access information about the route handling the incoming request:

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

You may refer to the API documentation for both the underlying class of the Route facade and Route instance to review all of the methods that are available on the router and route classes.

Cross-Origin Resource Sharing (CORS)

Laravel can automatically respond to CORS `OPTIONS` HTTP requests with values that you configure. All CORS settings may be configured in your application's `config/cors.php` configuration file. The `OPTIONS` requests will automatically be handled by the `HandleCors` middleware that is included by default in your global middleware stack. Your global middleware stack is located in your application's HTTP kernel (`App\Http\Kernel`).

{tip} For more information on CORS and CORS headers, please consult the MDN web documentation on CORS.

Route Caching

When deploying your application to production, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. To generate a route cache, execute the `route:cache` Artisan command:

```
php artisan route:cache
```

After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the `route:cache` command during your project's deployment.

You may use the `route:clear` command to clear the route cache:

```
php artisan route:clear
```

HTTP Session

- Introduction
    - Configuration
    - Driver Prerequisites
- Interacting With The Session
    - Retrieving Data
    - Storing Data
    - Flash Data
    - Deleting Data
    - Regenerating The Session ID
- Session Blocking
- Adding Custom Session Drivers
    - Implementing The Driver
    - Registering The Driver

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple requests. That user information is typically placed in a persistent store / backend that can be accessed from subsequent requests.

Laravel ships with a variety of session backends that are accessed through an expressive, unified API. Support for popular backends such as Memcached, Redis, and databases is included.

Configuration

Your application's session configuration file is stored at `config/session.php`. Be sure to review the options available to you in this file. By default, Laravel is configured to use the `file` session driver, which will work well for many applications. If your application will be load balanced across multiple web servers, you should choose a centralized store that all servers can access, such as Redis or a database.

The session `driver` configuration option defines where session data will be stored for each request. Laravel ships with several great drivers out of the box:

- `file` - sessions are stored in `storage/framework/sessions`. - `cookie` - sessions are stored in secure, encrypted cookies. - `database` - sessions are stored in a relational database. - `memcached` / `redis` - sessions are stored in one of these fast, cache based stores. - `dynamodb` - sessions are stored in AWS DynamoDB. - `array` - sessions are stored in a PHP array and will not be persisted.
{tip} The array driver is primarily used during testing and prevents the data stored in the session from being persisted.

Driver Prerequisites

Database

When using the `database` session driver, you will need to create a table to contain the session records. An example `Schema` declaration for the table may be found below:

```
Schema::create('sessions', function ($table) {
    $table->string('id')->primary();
    $table->foreignId('user_id')->nullable()->index();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity')->index();
});
```

You may use the `session:table` Artisan command to generate this migration. To learn more about database migrations, you may consult the complete migration documentation:

```
php artisan session:table

php artisan migrate
```

Redis

Before using Redis sessions with Laravel, you will need to either install the PhpRedis PHP extension via PECL or install the `predis/predis` package (~1.0) via Composer. For more information on configuring Redis, consult Laravel's Redis documentation.

{tip} In the `session` configuration file, the `connection` option may be used to specify which Redis connection is used by the session.

Interacting With The Session

Retrieving Data

There are two primary ways of working with session data in Laravel: the global `session` helper and via a `Request` instance. First, let's look at accessing the session via a `Request` instance, which can be type-hinted on a route closure or controller method. Remember, controller method dependencies are automatically injected via the Laravel service container:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  Request  $request
     * @param  int  $id
     * @return Response
     */
    public function show(Request $request, $id)
    {
        $value = $request->session()->get('key');

        //
    }
}
```

When you retrieve an item from the session, you may also pass a default value as the second argument to the `get` method. This default value will be returned if the specified key does not exist in the session. If you pass a closure as the default value to the `get` method and the requested key does not exist, the closure will be executed and its result returned:

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

The Global Session Helper

You may also use the global `session` PHP function to retrieve and store data in the session. When the `session` helper is called with a single, string argument, it will return the value of that session key. When the helper is called with an array of key / value pairs, those values will be stored in the session:

```
Route::get('/home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Specifying a default value...
    $value = session('key', 'default');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

{tip} There is little practical difference between using the session via an HTTP request instance versus using the global `session` helper. Both methods are testable via the `assertSessionHas` method which is available in all of your test cases.

Retrieving All Session Data

If you would like to retrieve all the data in the session, you may use the `all` method:

```
$data = $request->session()->all();
```

Determining If An Item Exists In The Session

To determine if an item is present in the session, you may use the `has` method. The `has`

method returns `true` if the item is present and is not `null`:

```
if ($request->session()->has('users')) {
    //
}
```

To determine if an item is present in the session, even if its value is `null`, you may use the `exists` method:

```
if ($request->session()->exists('users')) {
    //
}
```

To determine if an item is not present in the session, you may use the `missing` method. The `missing` method returns `true` if the item is `null` or if the item is not present:

```
if ($request->session()->missing('users')) {
    //
}
```

Storing Data

To store data in the session, you will typically use the request instance's `put` method or the global `session` helper:

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global "session" helper...
session(['key' => 'value']);
```

Pushing To Array Session Values

The `push` method may be used to push a new value onto a session value that is an array. For example, if the `user.teams` key contains an array of team names, you may push a new

value onto the array like so:

```
$request->session()->push('user.teams', 'developers');
```

Retrieving & Deleting An Item

The `pull` method will retrieve and delete an item from the session in a single statement:

```
$value = $request->session()->pull('key', 'default');
```

Incrementing & Decrementing Session Values

If your session data contains an integer you wish to increment or decrement, you may use the `increment` and `decrement` methods:

```
$request->session()->increment('count');

$request->session()->increment('count', $incrementBy = 2);

$request->session()->decrement('count');

$request->session()->decrement('count', $decrementBy = 2);
```

Flash Data

Sometimes you may wish to store items in the session for the next request. You may do so using the `flash` method. Data stored in the session using this method will be available immediately and during the subsequent HTTP request. After the subsequent HTTP request, the flashed data will be deleted. Flash data is primarily useful for short-lived status messages:

```
$request->session()->flash('status', 'Task was successful!');
```

If you need to persist your flash data for several requests, you may use the `reflash`

method, which will keep all of the flash data for an additional request. If you only need to keep specific flash data, you may use the `keep` method:

```
$request->session()->reflash();

$request->session()->keep(['username', 'email']);
```

To persist your flash data only for the current request, you may use the `now` method:

```
$request->session()->now('status', 'Task was successful!');
```

Deleting Data

The `forget` method will remove a piece of data from the session. If you would like to remove all data from the session, you may use the `flush` method:

```
// Forget a single key...
$request->session()->forget('name');

// Forget multiple keys...
$request->session()->forget(['name', 'status']);

$request->session()->flush();
```

Regenerating The Session ID

Regenerating the session ID is often done in order to prevent malicious users from exploiting a session fixation attack on your application.

Laravel automatically regenerates the session ID during authentication if you are using one of the Laravel application starter kits or Laravel Fortify; however, if you need to manually regenerate the session ID, you may use the `regenerate` method:

```
$request->session()->regenerate();
```

If you need to regenerate the session ID and remove all data from the session in a single statement, you may use the `invalidate` method:

```
$request->session()->invalidate();
```

Session Blocking

{note} To utilize session blocking, your application must be using a cache driver that supports atomic locks. Currently, those cache drivers include the `memcached`, `dynamodb`, `redis`, and `database` drivers. In addition, you may not use the `cookie` session driver.

By default, Laravel allows requests using the same session to execute concurrently. So, for example, if you use a JavaScript HTTP library to make two HTTP requests to your application, they will both execute at the same time. For many applications, this is not a problem; however, session data loss can occur in a small subset of applications that make concurrent requests to two different application endpoints which both write data to the session.

To mitigate this, Laravel provides functionality that allows you to limit concurrent requests for a given session. To get started, you may simply chain the `block` method onto your route definition. In this example, an incoming request to the `/profile` endpoint would acquire a session lock. While this lock is being held, any incoming requests to the `/profile` or `/order` endpoints which share the same session ID will wait for the first request to finish executing before continuing their execution:

```
Route::post('/profile', function () {
    //
})->block($lockSeconds = 10, $waitSeconds = 10)

Route::post('/order', function () {
    //
})->block($lockSeconds = 10, $waitSeconds = 10)
```

The `block` method accepts two optional arguments. The first argument accepted by the `block` method is the maximum number of seconds the session lock should be held for before it is released. Of course, if the request finishes executing before this time the lock will be released earlier.

The second argument accepted by the `block` method is the number of seconds a request should wait while attempting to obtain a session lock. An `Illuminate\Contracts\Cache\LockTimeoutException` will be thrown if the request is unable to obtain a session lock within the given number of seconds.

If neither of these arguments is passed, the lock will be obtained for a maximum of 10 seconds and requests will wait a maximum of 10 seconds while attempting to obtain a lock:

```
Route::post('/profile', function () {
    //
})->block()
```

Adding Custom Session Drivers

Implementing The Driver

If none of the existing session drivers fit your application's needs, Laravel makes it possible to write your own session handler. Your custom session driver should implement PHP's built-in `SessionHandlerInterface`. This interface contains just a few simple methods. A stubbed MongoDB implementation looks like the following:

```
<?php

namespace App\Extensions;

class MongoSessionHandler implements \SessionHandlerInterface
{
    public function open($savePath, $sessionName) {}
    public function close() {}
    public function read($sessionId) {}
    public function write($sessionId, $data) {}
    public function destroy($sessionId) {}
    public function gc($lifetime) {}
}
```

{tip} Laravel does not ship with a directory to contain your extensions. You are free to place them anywhere you like. In this example, we have created an `Extensions` directory to house the `MongoSessionHandler`.

Since the purpose of these methods is not readily understandable, let's quickly cover what each of the methods do:

- The `open` method would typically be used in file based session store systems. Since Laravel ships with a `file` session driver, you will rarely need to put anything in this method. You can simply leave this method empty. - The `close` method, like the `open` method, can also usually be disregarded. For most drivers, it is not needed. - The `read` method should return the string version of the session data associated with the given `$sessionId`. There is no need to do any serialization or other encoding when retrieving or storing session data in your driver, as Laravel will perform the serialization for you. - The `write` method should write the given `$data` string associated with the `$sessionId` to some persistent storage system, such as MongoDB or another storage system of your choice. Again, you should not perform any serialization - Laravel will have already handled that for you. - The `destroy`

method should remove the data associated with the `$sessionId` from persistent storage. - The `gc` method should destroy all session data that is older than the given `$lifetime`, which is a UNIX timestamp. For self-expiring systems like Memcached and Redis, this method may be left empty.

Registering The Driver

Once your driver has been implemented, you are ready to register it with Laravel. To add additional drivers to Laravel's session backend, you may use the `extend` method provided by the `Session` facade. You should call the `extend` method from the `boot` method of a service provider. You may do this from the existing `App\Providers\AppServiceProvider` or create an entirely new provider:

```php
<?php

namespace App\Providers;

use App\Extensions\MongoSessionHandler;
use Illuminate\Support\Facades\Session;
use Illuminate\Support\ServiceProvider;

class SessionServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Session::extend('mongo', function ($app) {
            // Return an implementation of SessionHandlerInterface...
            return new MongoSessionHandler;
        });
    }
}
```

Once the session driver has been registered, you may use the `mongo` driver in your `config/session.php` configuration file.

Directory Structure

- Introduction
- The Root Directory
    - The app Directory

    - The `bootstrap` Directory
    - The `config` Directory
    - The `database` Directory
    - The `public` Directory
    - The `resources` Directory
    - The `routes` Directory
    - The `storage` Directory
    - The `tests` Directory
    - The `vendor` Directory
- The App Directory
    - The `Broadcasting` Directory
    - The `Console` Directory
    - The `Events` Directory
    - The `Exceptions` Directory
    - The `Http` Directory
    - The `Jobs` Directory
    - The `Listeners` Directory
    - The `Mail` Directory
    - The `Models` Directory
    - The `Notifications` Directory
    - The `Policies` Directory
    - The `Providers` Directory
    - The `Rules` Directory

Introduction

The default Laravel application structure is intended to provide a great starting point for both large and small applications. But you are free to organize your application however you like. Laravel imposes almost no restrictions on where any given class is located - as long as Composer can autoload the class.

The Root Directory

The App Directory

The `app` directory contains the core code of your application. We'll explore this directory in more detail soon; however, almost all of the classes in your application will be in this directory.

The Bootstrap Directory

The `bootstrap` directory contains the `app.php` file which bootstraps the framework. This directory also houses a `cache` directory which contains framework generated files for performance optimization such as the route and services cache files. You should not typically need to modify any files within this directory.

The Config Directory

The `config` directory, as the name implies, contains all of your application's configuration files. It's a great idea to read through all of these files and familiarize yourself with all of the options available to you.

The Database Directory

The `database` directory contains your database migrations, model factories, and seeds. If you wish, you may also use this directory to hold an SQLite database.

The Public Directory

The `public` directory contains the `index.php` file, which is the entry point for all requests entering your application and configures autoloading. This directory also houses your assets such as images, JavaScript, and CSS.

The Resources Directory

The `resources` directory contains your views as well as your raw, un-compiled assets such as CSS or JavaScript. This directory also houses all of your language files.

The Routes Directory

The `routes` directory contains all of the route definitions for your application. By default, several route files are included with Laravel: `web.php`, `api.php`, `console.php`, and `channels.php`.

The `web.php` file contains routes that the `RouteServiceProvider` places in the `web` middleware group, which provides session state, CSRF protection, and cookie encryption. If your application does not offer a stateless, RESTful API then it is likely that all of your routes will most likely be defined in the `web.php` file.

The `api.php` file contains routes that the `RouteServiceProvider` places in the `api` middleware group. These routes are intended to be stateless, so requests entering the application through these routes are intended to be authenticated via tokens and will not have access to session state.

The `console.php` file is where you may define all of your closure based console commands. Each closure is bound to a command instance allowing a simple approach to interacting with each command's IO methods. Even though this file does not define HTTP routes, it defines console based entry points (routes) into your application.

The `channels.php` file is where you may register all of the event broadcasting channels that your application supports.

The Storage Directory

The `storage` directory contains your logs, compiled Blade templates, file based sessions, file caches, and other files generated by the framework. This directory is segregated into `app`, `framework`, and `logs` directories. The `app` directory may be used to store any files generated by your application. The `framework` directory is used to store framework generated files and caches. Finally, the `logs` directory contains your application's log files.

The `storage/app/public` directory may be used to store user-generated files, such as profile avatars, that should be publicly accessible. You should create a symbolic link at `public/storage` which points to this directory. You may create the link using the `php artisan storage:link` Artisan command.

The Tests Directory

The `tests` directory contains your automated tests. Example PHPUnit unit tests and feature tests are provided out of the box. Each test class should be suffixed with the word `Test`. You may run your tests using the `phpunit` or `php vendor/bin/phpunit` commands. Or, if you would like a more detailed and beautiful representation of your test results, you may run your tests using the `php artisan test` Artisan command.

The Vendor Directory

The `vendor` directory contains your Composer dependencies.

The App Directory

The majority of your application is housed in the `app` directory. By default, this directory is namespaced under `App` and is autoloaded by Composer using the PSR-4 autoloading standard.

The `app` directory contains a variety of additional directories such as `Console`, `Http`, and `Providers`. Think of the `Console` and `Http` directories as providing an API into the core of your application. The HTTP protocol and CLI are both mechanisms to interact with your application, but do not actually contain application logic. In other words, they are two ways of issuing commands to your application. The `Console` directory contains all of your Artisan commands, while the `Http` directory contains your controllers, middleware, and requests.

A variety of other directories will be generated inside the `app` directory as you use the `make` Artisan commands to generate classes. So, for example, the `app/Jobs` directory will not exist until you execute the `make:job` Artisan command to generate a job class.

{tip} Many of the classes in the `app` directory can be generated by Artisan via commands. To review the available commands, run the `php artisan list make` command in your terminal.

The Broadcasting Directory

The `Broadcasting` directory contains all of the broadcast channel classes for your application. These classes are generated using the `make:channel` command. This directory does not exist by default, but will be created for you when you create your first channel. To learn more about channels, check out the documentation on event broadcasting.

The Console Directory

The `Console` directory contains all of the custom Artisan commands for your application. These commands may be generated using the `make:command` command. This directory also houses your console kernel, which is where your custom Artisan commands are registered and your scheduled tasks are defined.

The Events Directory

This directory does not exist by default, but will be created for you by the `event:generate` and `make:event` Artisan commands. The `Events` directory houses event classes. Events may be used to alert other parts of your application that a given action has occurred, providing a great deal of flexibility and decoupling.

The Exceptions Directory

The `Exceptions` directory contains your application's exception handler and is also a good place to place any exceptions thrown by your application. If you would like to customize how your exceptions are logged or rendered, you should modify the `Handler` class in this directory.

The Http Directory

The `Http` directory contains your controllers, middleware, and form requests. Almost all of the logic to handle requests entering your application will be placed in this directory.

The Jobs Directory

This directory does not exist by default, but will be created for you if you execute the `make:job` Artisan command. The `Jobs` directory houses the queueable jobs for your application. Jobs may be queued by your application or run synchronously within the current request lifecycle. Jobs that run synchronously during the current request are sometimes referred to as "commands" since they are an implementation of the command pattern.

The Listeners Directory

This directory does not exist by default, but will be created for you if you execute the `event:generate` or `make:listener` Artisan commands. The `Listeners` directory contains the classes that handle your events. Event listeners receive an event instance and perform logic in response to the event being fired. For example, a `UserRegistered` event might be handled by a `SendWelcomeEmail` listener.

The Mail Directory

This directory does not exist by default, but will be created for you if you execute the `make:mail` Artisan command. The `Mail` directory contains all of your classes that represent emails sent by your application. Mail objects allow you to encapsulate all of the logic of building an email in a single, simple class that may be sent using the `Mail::send` method.

The Models Directory

The `Models` directory contains all of your Eloquent model classes. The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

The Notifications Directory

This directory does not exist by default, but will be created for you if you execute the `make:notification` Artisan command. The `Notifications` directory contains all of the "transactional" notifications that are sent by your application, such as simple notifications about events that happen within your application. Laravel's notification feature abstracts sending notifications over a variety of drivers such as email, Slack, SMS, or stored in a database.

The Policies Directory

This directory does not exist by default, but will be created for you if you execute the `make:policy` Artisan command. The `Policies` directory contains the authorization policy classes for your application. Policies are used to determine if a user can perform a given action against a resource.

The Providers Directory

The `Providers` directory contains all of the service providers for your application. Service providers bootstrap your application by binding services in the service container, registering events, or performing any other tasks to prepare your application for incoming requests.

In a fresh Laravel application, this directory will already contain several providers. You are free to add your own providers to this directory as needed.

The Rules Directory

This directory does not exist by default, but will be created for you if you execute the `make:rule` Artisan command. The `Rules` directory contains the custom validation rule objects for your application. Rules are used to encapsulate complicated validation logic in a simple object. For more information, check out the validation documentation.

URL Generation

- Introduction
- The Basics
    - Generating URLs
    - Accessing The Current URL
- URLs For Named Routes
    - Signed URLs
- URLs For Controller Actions
- Default Values

Introduction

Laravel provides several helpers to assist you in generating URLs for your application. These helpers are primarily helpful when building links in your templates and API responses, or when generating redirect responses to another part of your application.

The Basics

Generating URLs

The `url` helper may be used to generate arbitrary URLs for your application. The generated URL will automatically use the scheme (HTTP or HTTPS) and host from the current request being handled by the application:

```
$post = App\Models\Post::find(1);

echo url("/posts/{$post->id}");

// http://example.com/posts/1
```

Accessing The Current URL

If no path is provided to the `url` helper, an `Illuminate\Routing\UrlGenerator` instance is returned, allowing you to access information about the current URL:

```
// Get the current URL without the query string...
echo url()->current();

// Get the current URL including the query string...
echo url()->full();

// Get the full URL for the previous request...
echo url()->previous();
```

Each of these methods may also be accessed via the URL facade:

```
use Illuminate\Support\Facades\URL;

echo URL::current();
```

URLs For Named Routes

The `route` helper may be used to generate URLs to named routes. Named routes allow you to generate URLs without being coupled to the actual URL defined on the route. Therefore, if the route's URL changes, no changes need to be made to your calls to the `route` function. For example, imagine your application contains a route defined like the following:

```
Route::get('/post/{post}', function (Post $post) {
    //
})->name('post.show');
```

To generate a URL to this route, you may use the `route` helper like so:

```
echo route('post.show', ['post' => 1]);

// http://example.com/post/1
```

Of course, the `route` helper may also be used to generate URLs for routes with multiple parameters:

```
Route::get('/post/{post}/comment/{comment}', function (Post $post,
Comment $comment) {
    //
})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]);

// http://example.com/post/1/comment/3
```

Any additional array elements that do not correspond to the route's definition parameters will be added to the URL's query string:

```
echo route('post.show', ['post' => 1, 'search' => 'rocket']);

// http://example.com/post/1?search=rocket
```

Eloquent Models

You will often be generating URLs using the route key (typically the primary key) of Eloquent models. For this reason, you may pass Eloquent models as parameter values. The `route` helper will automatically extract the model's route key:

```
echo route('post.show', ['post' => $post]);
```

Signed URLs

Laravel allows you to easily create "signed" URLs to named routes. These URLs have a "signature" hash appended to the query string which allows Laravel to verify that the URL has not been modified since it was created. Signed URLs are especially useful for routes that are publicly accessible yet need a layer of protection against URL manipulation.

For example, you might use signed URLs to implement a public "unsubscribe" link that is emailed to your customers. To create a signed URL to a named route, use the `signedRoute` method of the `URL` facade:

```
use Illuminate\Support\Facades\URL;

return URL::signedRoute('unsubscribe', ['user' => 1]);
```

If you would like to generate a temporary signed route URL that expires after a specified amount of time, you may use the `temporarySignedRoute` method. When Laravel validates a temporary signed route URL, it will ensure that the expiration timestamp that is encoded into the signed URL has not elapsed:

```
use Illuminate\Support\Facades\URL;

return URL::temporarySignedRoute(
    'unsubscribe', now()->addMinutes(30), ['user' => 1]
);
```

Validating Signed Route Requests

To verify that an incoming request has a valid signature, you should call the `hasValidSignature` method on the incoming `Illuminate\Http\Request` instance:

```
use Illuminate\Http\Request;

Route::get('/unsubscribe/{user}', function (Request $request) {
    if (! $request->hasValidSignature()) {
        abort(401);
    }

    // ...
})->name('unsubscribe');
```

Sometimes, you may need to allow your application's frontend to append data to a signed URL, such as when performing client-side pagination. Therefore, you can specify request query parameters that should be ignored when validating a signed URL using the `hasValidSignatureWhileIgnoring` method. Remember, ignoring parameters allows anyone to modify those parameters on the request:

```
if (! $request->hasValidSignatureWhileIgnoring(['page', 'order'])) {
    abort(401);
}
```

Instead of validating signed URLs using the incoming request instance, you may assign the `Illuminate\Routing\Middleware\ValidateSignature` middleware to the route. If it is not already present, you should assign this middleware a key in your HTTP kernel's `routeMiddleware` array:

```
/**
 * The application's route middleware.
 *
 * These middleware may be assigned to groups or used individually.
 *
 * @var array
 */
protected $routeMiddleware = [
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
];
```

Once you have registered the middleware in your kernel, you may attach it to a route. If the incoming request does not have a valid signature, the middleware will automatically return a `403` HTTP response:

```
Route::post('/unsubscribe/{user}', function (Request $request) {
    // ...
})->name('unsubscribe')->middleware('signed');
```

Responding To Invalid Signed Routes

When someone visits a signed URL that has expired, they will receive a generic error page for the `403` HTTP status code. However, you can customize this behavior by defining a custom "renderable" closure for the `InvalidSignatureException` exception in your exception handler. This closure should return an HTTP response:

```
use Illuminate\Routing\Exceptions\InvalidSignatureException;

/**
 * Register the exception handling callbacks for the application.
 *
 * @return void
 */
public function register()
{
    $this->renderable(function (InvalidSignatureException $e) {
        return response()->view('error.link-expired', [], 403);
    });
}
```

URLs For Controller Actions

The `action` function generates a URL for the given controller action:

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

If the controller method accepts route parameters, you may pass an associative array of route parameters as the second argument to the function:

```
$url = action([UserController::class, 'profile'], ['id' => 1]);
```

Default Values

For some applications, you may wish to specify request-wide default values for certain URL parameters. For example, imagine many of your routes define a `{locale}` parameter:

```
Route::get('/{locale}/posts', function () {
    //
})->name('post.index');
```

It is cumbersome to always pass the `locale` every time you call the `route` helper. So, you may use the `URL::defaults` method to define a default value for this parameter that will always be applied during the current request. You may wish to call this method from a route middleware so that you have access to the current request:

```
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\URL;

class SetDefaultLocaleForUrls
{
    /**
     * Handle the incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return \Illuminate\Http\Response
     */
    public function handle($request, Closure $next)
    {
        URL::defaults(['locale' => $request->user()->locale]);

        return $next($request);
    }
}
```

Once the default value for the `locale` parameter has been set, you are no longer required

to pass its value when generating URLs via the `route` helper.

URL Defaults & Middleware Priority

Setting URL default values can interfere with Laravel's handling of implicit model bindings. Therefore, you should prioritize your middleware that set URL defaults to be executed before Laravel's own `SubstituteBindings` middleware. You can accomplish this by making sure your middleware occurs before the `SubstituteBindings` middleware within the `$middlewarePriority` property of your application's HTTP kernel.

The `$middlewarePriority` property is defined in the base `Illuminate\Foundation\Http\Kernel` class. You may copy its definition from that class and overwrite it in your application's HTTP kernel in order to modify it:

```
/**
 * The priority-sorted list of middleware.
 *
 * This forces non-global middleware to always be in the given order.
 *
 * @var array
 */
protected $middlewarePriority = [
    // ...
     \App\Http\Middleware\SetDefaultLocaleForUrls::class,
     \Illuminate\Routing\Middleware\SubstituteBindings::class,
     // ...
];
```

Validation

- Introduction
- Validation Quickstart
    - Defining The Routes
    - Creating The Controller
    - Writing The Validation Logic
    - Displaying The Validation Errors
    - Repopulating Forms
    - A Note On Optional Fields
- Form Request Validation
    - Creating Form Requests
    - Authorizing Form Requests

    - Customizing The Error Messages
    - Preparing Input For Validation
- Manually Creating Validators
    - Automatic Redirection
    - Named Error Bags
    - Customizing The Error Messages
    - After Validation Hook
- Working With Validated Input
- Working With Error Messages
    - Specifying Custom Messages In Language Files
    - Specifying Attributes In Language Files
    - Specifying Values In Language Files
- Available Validation Rules
- Conditionally Adding Rules
- Validating Arrays
    - Excluding Unvalidated Array Keys
    - Validating Nested Array Input
- Validating Passwords
- Custom Validation Rules
    - Using Rule Objects
    - Using Closures
    - Implicit Rules

Introduction

Laravel provides several different approaches to validate your application's incoming data. It is most common to use the `validate` method available on all incoming HTTP requests. However, we will discuss other approaches to validation as well.

Laravel includes a wide variety of convenient validation rules that you may apply to data, even providing the ability to validate if values are unique in a given database table. We'll cover each of these validation rules in detail so that you are familiar with all of Laravel's validation features.

Validation Quickstart

To learn about Laravel's powerful validation features, let's look at a complete example of validating a form and displaying the error messages back to the user. By reading this high-level overview, you'll be able to gain a good general understanding of how to validate incoming request data using Laravel:

Defining The Routes

First, let's assume we have the following routes defined in our `routes/web.php` file:

```
use App\Http\Controllers\PostController;

Route::get('/post/create', [PostController::class, 'create']);
Route::post('/post', [PostController::class, 'store']);
```

The `GET` route will display a form for the user to create a new blog post, while the `POST` route will store the new blog post in the database.

Creating The Controller

Next, let's take a look at a simple controller that handles incoming requests to these routes. We'll leave the `store` method empty for now:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return \Illuminate\View\View
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Validate and store the blog post...
    }
}
```

Writing The Validation Logic

Now we are ready to fill in our `store` method with the logic to validate the new blog post. To do this, we will use the `validate` method provided by the `Illuminate\Http\Request` object. If the validation rules pass, your code will keep executing normally; however, if validation fails, an `Illuminate\Validation\ValidationException` exception will be thrown and the proper error response will automatically be sent back to the user.

If validation fails during a traditional HTTP request, a redirect response to the previous URL will be generated. If the incoming request is an XHR request, a JSON response containing the validation error messages will be returned.

To get a better understanding of the `validate` method, let's jump back into the `store` method:

```
/**
 * Store a new blog post.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $validated = $request->validate([
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid...
}
```

As you can see, the validation rules are passed into the `validate` method. Don't worry - all available validation rules are documented. Again, if the validation fails, the proper response will automatically be generated. If the validation passes, our controller will continue executing normally.

Alternatively, validation rules may be specified as arrays of rules instead of a single `|` delimited string:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

In addition, you may use the `validateWithBag` method to validate a request and store any error messages within a named error bag:

```
$validatedData = $request->validateWithBag('post', [
    'title' => ['required', 'unique:posts', 'max:255'],
    'body' => ['required'],
]);
```

Stopping On First Validation Failure

Sometimes you may wish to stop running validation rules on an attribute after the first validation failure. To do so, assign the `bail` rule to the attribute:

```
$request->validate([
    'title' => 'bail|required|unique:posts|max:255',
    'body' => 'required',
]);
```

In this example, if the `unique` rule on the `title` attribute fails, the `max` rule will not be checked. Rules will be validated in the order they are assigned.

A Note On Nested Attributes

If the incoming HTTP request contains "nested" field data, you may specify these fields in your validation rules using "dot" syntax:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

On the other hand, if your field name contains a literal period, you can explicitly prevent this from being interpreted as "dot" syntax by escaping the period with a backslash:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'v1\.0' => 'required',
]);
```

Displaying The Validation Errors

So, what if the incoming request fields do not pass the given validation rules? As mentioned previously, Laravel will automatically redirect the user back to their previous location. In addition, all of the validation errors and request input will automatically be flashed to the

session.

An `$errors` variable is shared with all of your application's views by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. When this middleware is applied an `$errors` variable will always be available in your views, allowing you to conveniently assume the `$errors` variable is always defined and can be safely used. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

So, in our example, the user will be redirected to our controller's `create` method when validation fails, allowing us to display the error messages in the view:

```
<!-- /resources/views/post/create.blade.php -->

<h1>Create Post</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif

<!-- Create Post Form -->
```

Customizing The Error Messages

Laravel's built-in validation rules each has an error message that is located in your application's `resources/lang/en/validation.php` file. Within this file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another translation language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete localization documentation.

XHR Requests & Validation

In this example, we used a traditional form to send data to the application. However, many

applications receive XHR requests from a JavaScript powered frontend. When using the `validate` method during an XHR request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.

The `@error` Directive

You may use the `@error` Blade directive to quickly determine if validation error messages exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title" type="text" name="title" class="@error('title') is-invalid @enderror">

@error('title')
    <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

If you are using named error bags, you may pass the name of the error bag as the second argument to the `@error` directive:

```
<input ... class="@error('title', 'post') is-invalid @enderror">
```

Repopulating Forms

When Laravel generates a redirect response due to a validation error, the framework will automatically flash all of the request's input to the session. This is done so that you may conveniently access the input during the next request and repopulate the form that the user attempted to submit.

To retrieve flashed input from the previous request, invoke the `old` method on an instance of `Illuminate\Http\Request`. The `old` method will pull the previously flashed input data from the session:

```
$title = $request->old('title');
```

Laravel also provides a global `old` helper. If you are displaying old input within a Blade template, it is more convenient to use the `old` helper to repopulate the form. If no old input exists for the given field, `null` will be returned:

```
<input type="text" name="title" value="{{ old('title') }}">
```

A Note On Optional Fields

By default, Laravel includes the `TrimStrings` and `ConvertEmptyStringsToNull` middleware in your application's global middleware stack. These middleware are listed in the stack by the `App\Http\Kernel` class. Because of this, you will often need to mark your "optional" request fields as `nullable` if you do not want the validator to consider `null` values as invalid. For example:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
    'publish_at' => 'nullable|date',
]);
```

In this example, we are specifying that the `publish_at` field may be either `null` or a valid date representation. If the `nullable` modifier is not added to the rule definition, the validator would consider `null` an invalid date.

Form Request Validation

Creating Form Requests

For more complex validation scenarios, you may wish to create a "form request". Form requests are custom request classes that encapsulate their own validation and authorization logic. To create a form request class, you may use the `make:request` Artisan CLI command:

```
php artisan make:request StorePostRequest
```

The generated form request class will be placed in the `app/Http/Requests` directory. If this directory does not exist, it will be created when you run the `make:request` command. Each form request generated by Laravel has two methods: `authorize` and `rules`.

As you might have guessed, the `authorize` method is responsible for determining if the currently authenticated user can perform the action represented by the request, while the `rules` method returns the validation rules that should apply to the request's data:

```
/**
 * Get the validation rules that apply to the request.
 *
 * @return array
 */
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

{tip} You may type-hint any dependencies you require within the `rules` method's signature. They will automatically be resolved via the Laravel service container.

So, how are the validation rules evaluated? All you need to do is type-hint the request on your controller method. The incoming form request is validated before the controller method

is called, meaning you do not need to clutter your controller with any validation logic:

```
/**
 * Store a new blog post.
 *
 * @param  \App\Http\Requests\StorePostRequest  $request
 * @return Illuminate\Http\Response
 */
public function store(StorePostRequest $request)
{
    // The incoming request is valid...

    // Retrieve the validated input data...
    $validated = $request->validated();

    // Retrieve a portion of the validated input data...
    $validated = $request->safe()->only(['name', 'email']);
    $validated = $request->safe()->except(['name', 'email']);
}
```

If validation fails, a redirect response will be generated to send the user back to their previous location. The errors will also be flashed to the session so they are available for display. If the request was an XHR request, an HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

Adding After Hooks To Form Requests

If you would like to add an "after" validation hook to a form request, you may use the `withValidator` method. This method receives the fully constructed validator, allowing you to call any of its methods before the validation rules are actually evaluated:

```
/**
 * Configure the validator instance.
 *
 * @param  \Illuminate\Validation\Validator  $validator
 * @return void
 */
public function withValidator($validator)
{
    $validator->after(function ($validator) {
        if ($this->somethingElseIsInvalid()) {
            $validator->errors()->add('field', 'Something is wrong with
this field!');
        }
    });
}
```

Stopping On First Validation Failure Attribute

By adding a `stopOnFirstFailure` property to your request class, you may inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
/**
 * Indicates if the validator should stop on the first rule failure.
 *
 * @var bool
 */
protected $stopOnFirstFailure = true;
```

Customizing The Redirect Location

As previously discussed, a redirect response will be generated to send the user back to their previous location when form request validation fails. However, you are free to customize this behavior. To do so, define a `$redirect` property on your form request:

```
/**
 * The URI that users should be redirected to if validation fails.
 *
 * @var string
 */
protected $redirect = '/dashboard';
```

Or, if you would like to redirect users to a named route, you may define a `$redirectRoute` property instead:

```
/**
 * The route that users should be redirected to if validation fails.
 *
 * @var string
 */
protected $redirectRoute = 'dashboard';
```

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may determine if the authenticated user actually has the authority to update a given resource. For example, you may determine if a user actually owns a blog comment they are attempting to update. Most likely, you will interact with your authorization gates and policies within this method:

```
use App\Models\Comment;

/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    $comment = Comment::find($this->route('comment'));

    return $comment && $this->user()->can('update', $comment);
}
```

Since all form requests extend the base Laravel request class, we may use the `user` method to access the currently authenticated user. Also, note the call to the `route` method in the example above. This method grants you access to the URI parameters defined on the route being called, such as the `{comment}` parameter in the example below:

```
Route::post('/comment/{comment}');
```

Therefore, if your application is taking advantage of route model binding, your code may be made even more succinct by accessing the resolved model as a property of the request:

```
return $this->user()->can('update', $this->comment);
```

If the `authorize` method returns `false`, an HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to handle authorization logic for the request in another part of your application, you may simply return `true` from the `authorize` method:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

{tip} You may type-hint any dependencies you need within the `authorize` method's signature. They will automatically be resolved via the Laravel service container.

Customizing The Error Messages

You may customize the error messages used by the form request by overriding the `messages` method. This method should return an array of attribute / rule pairs and their corresponding error messages:

```
/**
 * Get the error messages for the defined validation rules.
 *
 * @return array
 */
public function messages()
{
    return [
        'title.required' => 'A title is required',
        'body.required' => 'A message is required',
    ];
}
```

Customizing The Validation Attributes

Many of Laravel's built-in validation rule error messages contain an `:attribute` placeholder. If you would like the `:attribute` placeholder of your validation message to be replaced with a custom attribute name, you may specify the custom names by overriding the `attributes` method. This method should return an array of attribute / name pairs:

```
/**
 * Get custom attributes for validator errors.
 *
 * @return array
 */
public function attributes()
{
    return [
        'email' => 'email address',
    ];
}
```

Preparing Input For Validation

If you need to prepare or sanitize any data from the request before you apply your validation rules, you may use the `prepareForValidation` method:

```
use Illuminate\Support\Str;

/**
 * Prepare the data for validation.
 *
 * @return void
 */
protected function prepareForValidation()
{
    $this->merge([
        'slug' => Str::slug($this->slug),
    ]);
}
```

Manually Creating Validators

If you do not want to use the `validate` method on the request, you may create a validator instance manually using the `Validator` facade. The `make` method on the facade generates a new validator instance:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Validator;

class PostController extends Controller
{
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        if ($validator->fails()) {
            return redirect('post/create')
                        ->withErrors($validator)
                        ->withInput();
        }

        // Retrieve the validated input...
        $validated = $validator->validated();

        // Retrieve a portion of the validated input...
        $validated = $validator->safe()->only(['name', 'email']);
        $validated = $validator->safe()->except(['name', 'email']);

        // Store the blog post...
    }
}
```

The first argument passed to the `make` method is the data under validation. The second argument is an array of the validation rules that should be applied to the data.

After determining whether the request validation failed, you may use the `withErrors`

method to flash the error messages to the session. When using this method, the `$errors` variable will automatically be shared with your views after redirection, allowing you to easily display them back to the user. The `withErrors` method accepts a validator, a `MessageBag`, or a PHP `array`.

Stopping On First Validation Failure

The `stopOnFirstFailure` method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

Automatic Redirection

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the HTTP request's `validate` method, you may call the `validate` method on an existing validator instance. If validation fails, the user will automatically be redirected or, in the case of an XHR request, a JSON response will be returned:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

You may use the `validateWithBag` method to store the error messages in a named error bag if validation fails:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validateWithBag('post');
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` containing the validation errors, allowing you to retrieve the error messages for a specific form. To achieve this, pass a name as the second argument to `withErrors`:

```
return redirect('register')->withErrors($validator, 'login');
```

You may then access the named `MessageBag` instance from the `$errors` variable:

```
{{ $errors->login->first('email') }}
```

Customizing The Error Messages

If needed, you may provide custom error messages that a validator instance should use instead of the default error messages provided by Laravel. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
$validator = Validator::make($input, $rules, $messages = [
    'required' => 'The :attribute field is required.',
]);
```

In this example, the `:attribute` placeholder will be replaced by the actual name of the field under validation. You may also utilize other placeholders in validation messages. For example:

```
$messages = [
    'same' => 'The :attribute and :other must match.',
    'size' => 'The :attribute must be exactly :size.',
    'between' => 'The :attribute value :input is not between :min -
:max.',
    'in' => 'The :attribute must be one of the following types:
:values',
];
```

Specifying A Custom Message For A Given Attribute

Sometimes you may wish to specify a custom error message only for a specific attribute. You may do so using "dot" notation. Specify the attribute's name first, followed by the rule:

```
$messages = [
    'email.required' => 'We need to know your email address!',
];
```

Specifying Custom Attribute Values

Many of Laravel's built-in error messages include an `:attribute` placeholder that is replaced with the name of the field or attribute under validation. To customize the values used to replace these placeholders for specific fields, you may pass an array of custom attributes as the fourth argument to the `Validator::make` method:

```
$validator = Validator::make($input, $rules, $messages, [
    'email' => 'email address',
]);
```

After Validation Hook

You may also attach callbacks to be run after validation is completed. This allows you to easily perform further validation and even add more error messages to the message collection. To get started, call the `after` method on a validator instance:

```
$validator = Validator::make(...);

$validator->after(function ($validator) {
    if ($this->somethingElseIsInvalid()) {
        $validator->errors()->add(
            'field', 'Something is wrong with this field!'
        );
    }
});

if ($validator->fails()) {
    //
}
```

Working With Validated Input

After validating incoming request data using a form request or a manually created validator instance, you may wish to retrieve the incoming request data that actually underwent validation. This can be accomplished in several ways. First, you may call the `validated` method on a form request or validator instance. This method returns an array of the data that was validated:

```
$validated = $request->validated();

$validated = $validator->validated();
```

Alternatively, you may call the `safe` method on a form request or validator instance. This method returns an instance of `Illuminate\Support\ValidatedInput`. This object exposes `only`, `except`, and `all` methods to retrieve a subset of the validated data or the entire array of validated data:

```
$validated = $request->safe()->only(['name', 'email']);

$validated = $request->safe()->except(['name', 'email']);

$validated = $request->safe()->all();
```

In addition, the `Illuminate\Support\ValidatedInput` instance may be iterated over and accessed like an array:

```
// Validated data may be iterated...
foreach ($request->safe() as $key => $value) {
    //
}

// Validated data may be accessed as an array...
$validated = $request->safe();

$email = $validated['email'];
```

If you would like to add additional fields to the validated data, you may call the `merge`

method:

```
$validated = $request->safe()->merge(['name' => 'Taylor Otwell']);
```

If you would like to retrieve the validated data as a collection instance, you may call the `collect` method:

```
$collection = $request->safe()->collect();
```

Working With Error Messages

After calling the `errors` method on a `Validator` instance, you will receive an `Illuminate\Support\MessageBag` instance, which has a variety of convenient methods for working with error messages. The `$errors` variable that is automatically made available to all views is also an instance of the `MessageBag` class.

Retrieving The First Error Message For A Field

To retrieve the first error message for a given field, use the `first` method:

```
$errors = $validator->errors();

echo $errors->first('email');
```

Retrieving All Error Messages For A Field

If you need to retrieve an array of all the messages for a given field, use the `get` method:

```
foreach ($errors->get('email') as $message) {
    //
}
```

If you are validating an array form field, you may retrieve all of the messages for each of the array elements using the `*` character:

```
foreach ($errors->get('attachments.*') as $message) {
    //
}
```

Retrieving All Error Messages For All Fields

To retrieve an array of all messages for all fields, use the `all` method:

```
foreach ($errors->all() as $message) {
    //
}
```

Determining If Messages Exist For A Field

The `has` method may be used to determine if any error messages exist for a given field:

```
if ($errors->has('email')) {
    //
}
```

Specifying Custom Messages In Language Files

Laravel's built-in validation rules each has an error message that is located in your application's `resources/lang/en/validation.php` file. Within this file, you will find a translation entry for each validation rule. You are free to change or modify these messages based on the needs of your application.

In addition, you may copy this file to another translation language directory to translate the messages for your application's language. To learn more about Laravel localization, check out the complete localization documentation.

Custom Messages For Specific Attributes

You may customize the error messages used for specified attribute and rule combinations within your application's validation language files. To do so, add your message customizations to the `custom` array of your application's `resources/lang/xx/validation.php` language file:

```
'custom' => [
    'email' => [
        'required' => 'We need to know your email address!',
        'max' => 'Your email address is too long!'
    ],
],
```

Specifying Attributes In Language Files

Many of Laravel's built-in error messages include an `:attribute` placeholder that is replaced with the name of the field or attribute under validation. If you would like the `:attribute` portion of your validation message to be replaced with a custom value, you may specify the custom attribute name in the `attributes` array of your `resources/lang/xx/validation.php` language file:

```
'attributes' => [
    'email' => 'email address',
],
```

Specifying Values In Language Files

Some of Laravel's built-in validation rule error messages contain a `:value` placeholder that is replaced with the current value of the request attribute. However, you may occasionally need the `:value` portion of your validation message to be replaced with a custom representation of the value. For example, consider the following rule that specifies that a credit card number is required if the `payment_type` has a value of `cc`:

```
Validator::make($request->all(), [
    'credit_card_number' => 'required_if:payment_type,cc'
]);
```

If this validation rule fails, it will produce the following error message:

```
The credit card number field is required when payment type is cc.
```

Instead of displaying `cc` as the payment type value, you may specify a more user-friendly value representation in your `resources/lang/xx/validation.php` language file by defining a `values` array:

```
'values' => [
    'payment_type' => [
        'cc' => 'credit card'
    ],
],
```

After defining this value, the validation rule will produce the following error message:

```
The credit card number field is required when payment type is credit
card.
```

Available Validation Rules

Below is a list of all available validation rules and their function:

Accepted Accepted If Active URL After (Date) After Or Equal (Date) Alpha Alpha Dash Alpha Numeric Array Bail Before (Date) Before Or Equal (Date) Between Boolean Confirmed Current Password Date Date Equals Date Format Declined Declined If Different Digits Digits Between Dimensions (Image Files) Distinct Email Ends With Enum Exclude Exclude If Exclude Unless Exclude Without Exists (Database) File Filled Greater Than Greater Than Or Equal Image (File) In In Array Integer IP Address JSON Less Than Less Than Or Equal Max MIME Types MIME Type By File Extension Min Multiple Of Not In Not Regex Nullable Numeric Password Present Prohibited Prohibited If Prohibited Unless Prohibits Regular Expression Required Required If Required Unless Required With Required With All Required Without Required Without All Same Size Sometimes Starts With String Timezone Unique (Database) URL UUID

accepted

The field under validation must be `"yes"`, `"on"`, `1`, or `true`. This is useful for validating "Terms of Service" acceptance or similar fields.

accepted_if:anotherfield,value,...

The field under validation must be `"yes"`, `"on"`, `1`, or `true` if another field under validation is equal to a specified value. This is useful for validating "Terms of Service" acceptance or similar fields.

active_url

The field under validation must have a valid A or AAAA record according to the `dns_get_record` PHP function. The hostname of the provided URL is extracted using the `parse_url` PHP function before being passed to `dns_get_record`.

after:*date*

The field under validation must be a value after a given date. The dates will be passed into the `strtotime` PHP function in order to be converted to a valid `DateTime` instance:

```
'start_date' => 'required|date|after:tomorrow'
```

Instead of passing a date string to be evaluated by `strtotime`, you may specify another field to compare against the date:

```
'finish_date' => 'required|date|after:start_date'
```

after_or_equal:*date*

The field under validation must be a value after or equal to the given date. For more information, see the after rule.

alpha

The field under validation must be entirely alphabetic characters.

alpha_dash

The field under validation may have alpha-numeric characters, as well as dashes and underscores.

alpha_num

The field under validation must be entirely alpha-numeric characters.

array

The field under validation must be a PHP `array`.

When additional values are provided to the `array` rule, each key in the input array must be present within the list of values provided to the rule. In the following example, the `admin` key in the input array is invalid since it is not contained in the list of values provided to the `array` rule:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:username,locale',
]);
```

In general, you should always specify the array keys that are allowed to be present within your array. Otherwise, the validator's `validate` and `validated` methods will return all of the validated data, including the array and all of its keys, even if those keys were not validated by other nested array validation rules.

If you would like, you may instruct Laravel's validator to never include unvalidated array keys in the "validated" data it returns, even if you use the `array` rule without specifying a list of allowed keys. To accomplish this, you may call the validator's `excludeUnvalidatedArrayKeys` method in the `boot` method of your application's `AppServiceProvider`. After doing so, the validator will include array keys in the "validated" data it returns only when those keys were specifically validated by nested array rules:

```
use Illuminate\Support\Facades\Validator;

/**
 * Register any application services.
 *
 * @return void
 */
public function boot()
{
    Validator::excludeUnvalidatedArrayKeys();
}
```

bail

Stop running validation rules for the field after the first validation failure.

While the `bail` rule will only stop validating a specific field when it encounters a validation failure, the `stopOnFirstFailure` method will inform the validator that it should stop validating all attributes once a single validation failure has occurred:

```
if ($validator->stopOnFirstFailure()->fails()) {
    // ...
}
```

before:*date*

The field under validation must be a value preceding the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance. In addition, like the `after` rule, the name of another field under validation may be supplied as the value of `date`.

before_or_equal:*date*

The field under validation must be a value preceding or equal to the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance. In addition, like the `after` rule, the name of another field under validation may be supplied as the value of `date`.

between:*min*,*max*

The field under validation must have a size between the given *min* and *max*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

boolean

The field under validation must be able to be cast as a boolean. Accepted input are `true`, `false`, `1`, `0`, `"1"`, and `"0"`.

confirmed

The field under validation must have a matching field of `{field}_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

current_password

The field under validation must match the authenticated user's password. You may specify an authentication guard using the rule's first parameter:

```
'password' => 'current_password:api'
```

date

The field under validation must be a valid, non-relative date according to the `strtotime` PHP function.

date_equals:*date*

The field under validation must be equal to the given date. The dates will be passed into the PHP `strtotime` function in order to be converted into a valid `DateTime` instance.

date_format:*format*

The field under validation must match the given *format*. You should use **either** `date` or `date_format` when validating a field, not both. This validation rule supports all formats supported by PHP's DateTime class.

declined

The field under validation must be `"no"`, `"off"`, `0`, or `false`.

declined_if:anotherfield,value,...

The field under validation must be `"no"`, `"off"`, `0`, or `false` if another field under validation is equal to a specified value.

different:*field*

The field under validation must have a different value than *field*.

digits:*value*

The field under validation must be *numeric* and must have an exact length of *value*.

digits_between:*min*,*max*

The field under validation must be *numeric* and must have a length between the given *min* and *max*.

dimensions

The file under validation must be an image meeting the dimension constraints as specified by the rule's parameters:

```
'avatar' => 'dimensions:min_width=100,min_height=200'
```

Available constraints are: *min_width*, *max_width*, *min_height*, *max_height*, *width*, *height*, *ratio*.

A *ratio* constraint should be represented as width divided by height. This can be specified either by a fraction like `3/2` or a float like `1.5`:

```
'avatar' => 'dimensions:ratio=3/2'
```

Since this rule requires several arguments, you may use the `Rule::dimensions` method to fluently construct the rule:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'avatar' => [
        'required',
        Rule::dimensions()->maxWidth(1000)->maxHeight(500)->ratio(3 /
2),
    ],
]);
```

distinct

When validating arrays, the field under validation must not have any duplicate values:

```
'foo.*.id' => 'distinct'
```

Distinct uses loose variable comparisons by default. To use strict comparisons, you may add the `strict` parameter to your validation rule definition:

```
'foo.*.id' => 'distinct:strict'
```

You may add `ignore_case` to the validation rule's arguments to make the rule ignore capitalization differences:

```
'foo.*.id' => 'distinct:ignore_case'
```

email

The field under validation must be formatted as an email address. This validation rule utilizes the `egulias/email-validator` package for validating the email address. By default, the `RFCValidation` validator is applied, but you can apply other validation styles as well:

```
'email' => 'email:rfc,dns'
```

The example above will apply the `RFCValidation` and `DNSCheckValidation` validations. Here's a full list of validation styles you can apply:

- `rfc`: `RFCValidation` - `strict`: `NoRFCWarningsValidation` - `dns`: `DNSCheckValidation` - `spoof`: `SpoofCheckValidation` - `filter`: `FilterEmailValidation`

The `filter` validator, which uses PHP's `filter_var` function, ships with Laravel and was Laravel's default email validation behavior prior to Laravel version 5.8.

{note} The `dns` and `spoof` validators require the PHP `intl` extension.

ends_with:*foo,bar,...*

The field under validation must end with one of the given values.

enum

The `Enum` rule is a class based rule that validates whether the field under validation contains a valid enum value. The `Enum` rule accepts the name of the enum as its only constructor argument:

```
use App\Enums\ServerStatus;
use Illuminate\Validation\Rules\Enum;

$request->validate([
    'status' => [new Enum(ServerStatus::class)],
]);
```

{note} Enums are only available on PHP 8.1+.

exclude

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods.

exclude_if:*anotherfield*,*value*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is equal to *value*.

exclude_unless:*anotherfield*,*value*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods unless *anotherfield*'s field is equal to *value*. If *value* is `null` (`exclude_unless:name,null`), the field under validation will be excluded unless the comparison field is `null` or the comparison field is missing from the request data.

exclude_without:*anotherfield*

The field under validation will be excluded from the request data returned by the `validate` and `validated` methods if the *anotherfield* field is not present.

exists:*table*,*column*

The field under validation must exist in a given database table.

Basic Usage Of Exists Rule

```
'state' => 'exists:states'
```

If the `column` option is not specified, the field name will be used. So, in this case, the rule will validate that the `states` database table contains a record with a `state` column value matching the request's `state` attribute value.

Specifying A Custom Column Name

You may explicitly specify the database column name that should be used by the validation rule by placing it after the database table name:

```
'state' => 'exists:states,abbreviation'
```

Occasionally, you may need to specify a specific database connection to be used for the `exists` query. You can accomplish this by prepending the connection name to the table name:

```
'email' => 'exists:connection.staff,email'
```

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
'user_id' => 'exists:App\Models\User,id'
```

If you would like to customize the query executed by the validation rule, you may use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit them:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::exists('staff')->where(function ($query) {
            return $query->where('account_id', 1);
        }),
    ],
]);
```

file

The field under validation must be a successfully uploaded file.

filled

The field under validation must not be empty when it is present.

gt:*field*

The field under validation must be greater than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

gte:*field*

The field under validation must be greater than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

image

The file under validation must be an image (jpg, jpeg, png, bmp, gif, svg, or webp).

in:*foo*,*bar*,…

The field under validation must be included in the given list of values. Since this rule often requires you to `implode` an array, the `Rule::in` method may be used to fluently construct

the rule:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'zones' => [
        'required',
        Rule::in(['first-zone', 'second-zone']),
    ],
]);
```

When the `in` rule is combined with the `array` rule, each value in the input array must be present within the list of values provided to the `in` rule. In the following example, the `LAS` airport code in the input array is invalid since it is not contained in the list of airports provided to the `in` rule:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

$input = [
    'airports' => ['NYC', 'LAS'],
];

Validator::make($input, [
    'airports' => [
        'required',
        'array',
        Rule::in(['NYC', 'LIT']),
    ],
]);
```

in_array:*anotherfield*.*

The field under validation must exist in *anotherfield*'s values.

integer

The field under validation must be an integer.

{note} This validation rule does not verify that the input is of the "integer" variable type, only that the input is of a type accepted by PHP's `FILTER_VALIDATE_INT` rule. If you need to validate the input as being a number please use this rule in combination with the `numeric` validation rule.

ip

The field under validation must be an IP address.

ipv4

The field under validation must be an IPv4 address.

ipv6

The field under validation must be an IPv6 address.

json

The field under validation must be a valid JSON string.

lt:*field*

The field under validation must be less than the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

lte:*field*

The field under validation must be less than or equal to the given *field*. The two fields must be of the same type. Strings, numerics, arrays, and files are evaluated using the same conventions as the `size` rule.

max:*value*

The field under validation must be less than or equal to a maximum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

mimetypes:*text/plain*,...

The file under validation must match one of the given MIME types:

```
'video' => 'mimetypes:video/avi,video/mpeg,video/quicktime'
```

To determine the MIME type of the uploaded file, the file's contents will be read and the framework will attempt to guess the MIME type, which may be different from the client's provided MIME type.

mimes:*foo*,*bar*,...

The file under validation must have a MIME type corresponding to one of the listed extensions.

Basic Usage Of MIME Rule

```
'photo' => 'mimes:jpg,bmp,png'
```

Even though you only need to specify the extensions, this rule actually validates the MIME type of the file by reading the file's contents and guessing its MIME type. A full listing of MIME types and their corresponding extensions may be found at the following location:

https://svn.apache.org/repos/asf/httpd/httpd/trunk/docs/conf/mime.types

min:*value*

The field under validation must have a minimum *value*. Strings, numerics, arrays, and files are evaluated in the same fashion as the `size` rule.

multiple_of:*value*

The field under validation must be a multiple of *value*.

{note} The bcmath PHP extension is required in order to use the `multiple_of` rule.

not_in:*foo,bar,...*

The field under validation must not be included in the given list of values. The `Rule::notIn` method may be used to fluently construct the rule:

```
use Illuminate\Validation\Rule;

Validator::make($data, [
    'toppings' => [
        'required',
        Rule::notIn(['sprinkles', 'cherries']),
    ],
]);
```

not_regex:*pattern*

The field under validation must not match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'not_regex:/^.+$/i'`.

{note} When using the `regex` / `not_regex` patterns, it may be necessary to specify your validation rules using an array instead of using `|` delimiters, especially if the regular expression contains a `|` character.

nullable

The field under validation may be `null`.

numeric

The field under validation must be numeric.

password

The field under validation must match the authenticated user's password.

{note} This rule was renamed to `current_password` with the intention of removing it in Laravel 9. Please use the Current Password rule instead.

present

The field under validation must be present in the input data but can be empty.

prohibited

The field under validation must be empty or not present.

prohibited_if:*anotherfield*,*value*,...

The field under validation must be empty or not present if the *anotherfield* field is equal to any *value*.

prohibited_unless:*anotherfield*,*value*,...

The field under validation must be empty or not present unless the *anotherfield* field is equal to any *value*.

prohibits:*anotherfield*,...

If the field under validation is present, no fields in *anotherfield* can be present, even if empty.

regex:*pattern*

The field under validation must match the given regular expression.

Internally, this rule uses the PHP `preg_match` function. The pattern specified should obey the same formatting required by `preg_match` and thus also include valid delimiters. For example: `'email' => 'regex:/^.+@.+$/i'`.

{note} When using the `regex` / `not_regex` patterns, it may be necessary to specify rules in an array instead of using `|` delimiters, especially if the regular expression contains a `|` character.

required

The field under validation must be present in the input data and not empty. A field is considered "empty" if one of the following conditions are true:

- The value is `null`.

- The value is an empty string.
- The value is an empty array or empty `Countable` object.
- The value is an uploaded file with no path.

required_if:*anotherfield*,*value*,...

The field under validation must be present and not empty if the *anotherfield* field is equal to any *value*.

If you would like to construct a more complex condition for the `required_if` rule, you may use the `Rule::requiredIf` method. This method accepts a boolean or a closure. When passed a closure, the closure should return `true` or `false` to indicate if the field under validation is required:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf($request->user()->is_admin),
]);

Validator::make($request->all(), [
    'role_id' => Rule::requiredIf(function () use ($request) {
        return $request->user()->is_admin;
    }),
]);
```

required_unless:*anotherfield*,*value*,...

The field under validation must be present and not empty unless the *anotherfield* field is equal to any *value*. This also means *anotherfield* must be present in the request data unless *value* is `null`. If *value* is `null` (`required_unless:name,null`), the field under validation will be required unless the comparison field is `null` or the comparison field is missing from the request data.

required_with:*foo*,*bar*,...

The field under validation must be present and not empty *only if* any of the other specified fields are present and not empty.

required_with_all:*foo*,*bar*,...

The field under validation must be present and not empty *only if* all of the other specified fields are present and not empty.

required_without:*foo*,*bar*,...

The field under validation must be present and not empty *only when* any of the other specified fields are empty or not present.

required_without_all:*foo*,*bar*,...

The field under validation must be present and not empty *only when* all of the other specified fields are empty or not present.

same:*field*

The given *field* must match the field under validation.

size:*value*

The field under validation must have a size matching the given *value*. For string data, *value* corresponds to the number of characters. For numeric data, *value* corresponds to a given integer value (the attribute must also have the `numeric` or `integer` rule). For an array, *size* corresponds to the `count` of the array. For files, *size* corresponds to the file size in kilobytes. Let's look at some examples:

```
// Validate that a string is exactly 12 characters long...
'title' => 'size:12';

// Validate that a provided integer equals 10...
'seats' => 'integer|size:10';

// Validate that an array has exactly 5 elements...
'tags' => 'array|size:5';

// Validate that an uploaded file is exactly 512 kilobytes...
'image' => 'file|size:512';
```

starts_with:*foo*,*bar*,...

The field under validation must start with one of the given values.

string

The field under validation must be a string. If you would like to allow the field to also be `null`, you should assign the `nullable` rule to the field.

timezone

The field under validation must be a valid timezone identifier according to the `timezone_identifiers_list` PHP function.

unique:*table*,*column*

The field under validation must not exist within the given database table.

Specifying A Custom Table / Column Name:

Instead of specifying the table name directly, you may specify the Eloquent model which should be used to determine the table name:

```
'email' => 'unique:App\Models\User,email_address'
```

The `column` option may be used to specify the field's corresponding database column. If the `column` option is not specified, the name of the field under validation will be used.

```
'email' => 'unique:users,email_address'
```

Specifying A Custom Database Connection

Occasionally, you may need to set a custom connection for database queries made by the Validator. To accomplish this, you may prepend the connection name to the table name:

```
'email' => 'unique:connection.users,email_address'
```

Forcing A Unique Rule To Ignore A Given ID:

Sometimes, you may wish to ignore a given ID during unique validation. For example, consider an "update profile" screen that includes the user's name, email address, and location. You will probably want to verify that the email address is unique. However, if the user only changes the name field and not the email field, you do not want a validation error to be thrown because the user is already the owner of the email address in question.

To instruct the validator to ignore the user's ID, we'll use the `Rule` class to fluently define the rule. In this example, we'll also specify the validation rules as an array instead of using the `|` character to delimit the rules:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rule;

Validator::make($data, [
    'email' => [
        'required',
        Rule::unique('users')->ignore($user->id),
    ],
]);
```

{note} You should never pass any user controlled request input into the `ignore` method. Instead, you should only pass a system generated unique ID such as an auto-incrementing ID or UUID from an Eloquent model instance. Otherwise, your application will be vulnerable to an SQL injection attack.

Instead of passing the model key's value to the `ignore` method, you may also pass the entire model instance. Laravel will automatically extract the key from the model:

```
Rule::unique('users')->ignore($user)
```

If your table uses a primary key column name other than `id`, you may specify the name of the column when calling the `ignore` method:

```
Rule::unique('users')->ignore($user->id, 'user_id')
```

By default, the `unique` rule will check the uniqueness of the column matching the name of the attribute being validated. However, you may pass a different column name as the second argument to the `unique` method:

```
Rule::unique('users', 'email_address')->ignore($user->id),
```

Adding Additional Where Clauses:

You may specify additional query conditions by customizing the query using the `where` method. For example, let's add a query condition that scopes the query to only search records that have an `account_id` column value of `1`:

```
'email' => Rule::unique('users')->where(function ($query) {
    return $query->where('account_id', 1);
})
```

url

The field under validation must be a valid URL.

uuid

The field under validation must be a valid RFC 4122 (version 1, 3, 4, or 5) universally unique identifier (UUID).

Conditionally Adding Rules

Skipping Validation When Fields Have Certain Values

You may occasionally wish to not validate a given field if another field has a given value. You may accomplish this using the `exclude_if` validation rule. In this example, the `appointment_date` and `doctor_name` fields will not be validated if the `has_appointment` field has a value of `false`:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' =>
'exclude_if:has_appointment,false|required|date',
    'doctor_name' => 'exclude_if:has_appointment,false|required|string',
]);
```

Alternatively, you may use the `exclude_unless` rule to not validate a given field unless another field has a given value:

```
$validator = Validator::make($data, [
    'has_appointment' => 'required|boolean',
    'appointment_date' =>
'exclude_unless:has_appointment,true|required|date',
    'doctor_name' =>
'exclude_unless:has_appointment,true|required|string',
]);
```

Validating When Present

In some situations, you may wish to run validation checks against a field **only** if that field is present in the data being validated. To quickly accomplish this, add the `sometimes` rule to your rule list:

```
$v = Validator::make($data, [
    'email' => 'sometimes|required|email',
]);
```

In the example above, the `email` field will only be validated if it is present in the `$data` array.

{tip} If you are attempting to validate a field that should always be present but may be empty, check out this note on optional fields.

Complex Conditional Validation

Sometimes you may wish to add validation rules based on more complex conditional logic. For example, you may wish to require a given field only if another field has a greater value than 100. Or, you may need two fields to have a given value only when another field is present. Adding these validation rules doesn't have to be a pain. First, create a `Validator` instance with your *static rules* that never change:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'email' => 'required|email',
    'games' => 'required|numeric',
]);
```

Let's assume our web application is for game collectors. If a game collector registers with our application and they own more than 100 games, we want them to explain why they own so many games. For example, perhaps they run a game resale shop, or maybe they just enjoy collecting games. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
$validator->sometimes('reason', 'required|max:500', function ($input) {
    return $input->games >= 100;
});
```

The first argument passed to the `sometimes` method is the name of the field we are conditionally validating. The second argument is a list of the rules we want to add. If the closure passed as the third argument returns `true`, the rules will be added. This method makes it a breeze to build complex conditional validations. You may even add conditional

validations for several fields at once:

```
$validator->sometimes(['reason', 'cost'], 'required', function ($input)
{
    return $input->games >= 100;
});
```

{tip} The `$input` parameter passed to your closure will be an instance of `Illuminate\Support\Fluent` and may be used to access your input and files under validation.

Complex Conditional Array Validation

Sometimes you may want to validate a field based on another field in the same nested array whose index you do not know. In these situations, you may allow your closure to receive a second argument which will be the current individual item in the array being validated:

```
$input = [
    'channels' => [
        [
            'type' => 'email',
            'address' => 'abigail@example.com',
        ],
        [
            'type' => 'url',
            'address' => 'https://example.com',
        ],
    ],
];

$validator->sometimes('channels.*.address', 'email', function($input,
$item) {
    return $item->type === 'email';
});

$validator->sometimes('channels.*.address', 'url', function($input,
$item) {
    return $item->type !== 'email';
});
```

Like the `$input` parameter passed to the closure, the `$item` parameter is an instance of

`Illuminate\Support\Fluent` when the attribute data is an array; otherwise, it is a string.

Validating Arrays

As discussed in the `array` validation rule documentation, the `array` rule accepts a list of allowed array keys. If any additional keys are present within the array, validation will fail:

```
use Illuminate\Support\Facades\Validator;

$input = [
    'user' => [
        'name' => 'Taylor Otwell',
        'username' => 'taylorotwell',
        'admin' => true,
    ],
];

Validator::make($input, [
    'user' => 'array:username,locale',
]);
```

In general, you should always specify the array keys that are allowed to be present within your array. Otherwise, the validator's `validate` and `validated` methods will return all of the validated data, including the array and all of its keys, even if those keys were not validated by other nested array validation rules.

Excluding Unvalidated Array Keys

If you would like, you may instruct Laravel's validator to never include unvalidated array keys in the "validated" data it returns, even if you use the `array` rule without specifying a list of allowed keys. To accomplish this, you may call the validator's `excludeUnvalidatedArrayKeys` method in the `boot` method of your application's `AppServiceProvider`. After doing so, the validator will include array keys in the "validated" data it returns only when those keys were specifically validated by nested array rules:

```php
use Illuminate\Support\Facades\Validator;

/**
 * Register any application services.
 *
 * @return void
 */
public function boot()
{
    Validator::excludeUnvalidatedArrayKeys();
}
```

Validating Nested Array Input

Validating nested array based form input fields doesn't have to be a pain. You may use "dot notation" to validate attributes within an array. For example, if the incoming HTTP request contains a `photos[profile]` field, you may validate it like so:

```php
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'photos.profile' => 'required|image',
]);
```

You may also validate each element of an array. For example, to validate that each email in a given array input field is unique, you may do the following:

```php
$validator = Validator::make($request->all(), [
    'person.*.email' => 'email|unique:users',
    'person.*.first_name' => 'required_with:person.*.last_name',
]);
```

Likewise, you may use the * character when specifying custom validation messages in your language files, making it a breeze to use a single validation message for array based fields:

```
'custom' => [
    'person.*.email' => [
        'unique' => 'Each person must have a unique email address',
    ]
],
```

Validating Passwords

To ensure that passwords have an adequate level of complexity, you may use Laravel's `Password` rule object:

```
use Illuminate\Support\Facades\Validator;
use Illuminate\Validation\Rules\Password;

$validator = Validator::make($request->all(), [
    'password' => ['required', 'confirmed', Password::min(8)],
]);
```

The `Password` rule object allows you to easily customize the password complexity requirements for your application, such as specifying that passwords require at least one letter, number, symbol, or characters with mixed casing:

```
// Require at least 8 characters...
Password::min(8)

// Require at least one letter...
Password::min(8)->letters()

// Require at least one uppercase and one lowercase letter...
Password::min(8)->mixedCase()

// Require at least one number...
Password::min(8)->numbers()

// Require at least one symbol...
Password::min(8)->symbols()
```

In addition, you may ensure that a password has not been compromised in a public password data breach leak using the `uncompromised` method:

```
Password::min(8)->uncompromised()
```

Internally, the `Password` rule object uses the k-Anonymity model to determine if a password

has been leaked via the haveibeenpwned.com service without sacrificing the user's privacy or security.

By default, if a password appears at least once in a data leak, it will be considered compromised. You can customize this threshold using the first argument of the `uncompromised` method:

```
// Ensure the password appears less than 3 times in the same data
leak...
Password::min(8)->uncompromised(3);
```

Of course, you may chain all the methods in the examples above:

```
Password::min(8)
    ->letters()
    ->mixedCase()
    ->numbers()
    ->symbols()
    ->uncompromised()
```

Defining Default Password Rules

You may find it convenient to specify the default validation rules for passwords in a single location of your application. You can easily accomplish this using the `Password::defaults` method, which accepts a closure. The closure given to the `defaults` method should return the default configuration of the Password rule. Typically, the `defaults` rule should be called within the `boot` method of one of your application's service providers:

```php
use Illuminate\Validation\Rules\Password;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Password::defaults(function () {
        $rule = Password::min(8);

        return $this->app->isProduction()
                    ? $rule->mixedCase()->uncompromised()
                    : $rule;
    });
}
```

Then, when you would like to apply the default rules to a particular password undergoing validation, you may invoke the `defaults` method with no arguments:

```php
'password' => ['required', Password::defaults()],
```

Custom Validation Rules

Using Rule Objects

Laravel provides a variety of helpful validation rules; however, you may wish to specify some of your own. One method of registering custom validation rules is using rule objects. To generate a new rule object, you may use the `make:rule` Artisan command. Let's use this command to generate a rule that verifies a string is uppercase. Laravel will place the new rule in the `app/Rules` directory. If this directory does not exist, Laravel will create it when you execute the Artisan command to create your rule:

```
php artisan make:rule Uppercase
```

Once the rule has been created, we are ready to define its behavior. A rule object contains two methods: `passes` and `message`. The `passes` method receives the attribute value and name, and should return `true` or `false` depending on whether the attribute value is valid or not. The `message` method should return the validation error message that should be used when validation fails:

```php
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;

class Uppercase implements Rule
{
    /**
     * Determine if the validation rule passes.
     *
     * @param  string  $attribute
     * @param  mixed  $value
     * @return bool
     */
    public function passes($attribute, $value)
    {
        return strtoupper($value) === $value;
    }

    /**
     * Get the validation error message.
     *
     * @return string
     */
    public function message()
    {
        return 'The :attribute must be uppercase.';
    }
}
```

You may call the `trans` helper from your `message` method if you would like to return an error message from your translation files:

```
/**
 * Get the validation error message.
 *
 * @return string
 */
public function message()
{
    return trans('validation.uppercase');
}
```

Once the rule has been defined, you may attach it to a validator by passing an instance of the rule object with your other validation rules:

```
use App\Rules\Uppercase;

$request->validate([
    'name' => ['required', 'string', new Uppercase],
]);
```

Accessing Additional Data

If your custom validation rule class needs to access all of the other data undergoing validation, your rule class may implement the `Illuminate\Contracts\Validation\DataAwareRule` interface. This interface requires your class to define a `setData` method. This method will automatically be invoked by Laravel (before validation proceeds) with all of the data under validation:

```php
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\DataAwareRule;

class Uppercase implements Rule, DataAwareRule
{
    /**
     * All of the data under validation.
     *
     * @var array
     */
    protected $data = [];

    // ...

    /**
     * Set the data under validation.
     *
     * @param  array  $data
     * @return $this
     */
    public function setData($data)
    {
        $this->data = $data;

        return $this;
    }
}
```

Or, if your validation rule requires access to the validator instance performing the validation, you may implement the `ValidatorAwareRule` interface:

```php
<?php

namespace App\Rules;

use Illuminate\Contracts\Validation\Rule;
use Illuminate\Contracts\Validation\ValidatorAwareRule;

class Uppercase implements Rule, ValidatorAwareRule
{
    /**
     * The validator instance.
     *
     * @var \Illuminate\Validation\Validator
     */
    protected $validator;

    // ...

    /**
     * Set the current validator.
     *
     * @param  \Illuminate\Validation\Validator  $validator
     * @return $this
     */
    public function setValidator($validator)
    {
        $this->validator = $validator;

        return $this;
    }
}
```

Using Closures

If you only need the functionality of a custom rule once throughout your application, you may use a closure instead of a rule object. The closure receives the attribute's name, the attribute's value, and a `$fail` callback that should be called if validation fails:

```
use Illuminate\Support\Facades\Validator;

$validator = Validator::make($request->all(), [
    'title' => [
        'required',
        'max:255',
        function ($attribute, $value, $fail) {
            if ($value === 'foo') {
                $fail('The '.$attribute.' is invalid.');
            }
        },
    ],
]);
```

Implicit Rules

By default, when an attribute being validated is not present or contains an empty string, normal validation rules, including custom rules, are not run. For example, the `unique` rule will not be run against an empty string:

```
use Illuminate\Support\Facades\Validator;

$rules = ['name' => 'unique:users,name'];

$input = ['name' => ''];

Validator::make($input, $rules)->passes(); // true
```

For a custom rule to run even when an attribute is empty, the rule must imply that the attribute is required. To create an "implicit" rule, implement the `Illuminate\Contracts\Validation\ImplicitRule` interface. This interface serves as a "marker interface" for the validator; therefore, it does not contain any additional methods you need to implement beyond the methods required by the typical `Rule` interface.

To generate a new implicit rule object, you may use the `make:rule` Artisan command with the `--implicit` option :

```
php artisan make:rule Uppercase --implicit
```

{note} An "implicit" rule only *implies* that the attribute is required. Whether it actually invalidates a missing or empty attribute is up to you.

Views

- Introduction
- Creating & Rendering Views
    - Nested View Directories
    - Creating The First Available View
    - Determining If A View Exists
- Passing Data To Views
    - Sharing Data With All Views
- View Composers
    - View Creators
- Optimizing Views

Introduction

Of course, it's not practical to return entire HTML documents strings directly from your routes and controllers. Thankfully, views provide a convenient way to place all of our HTML in separate files. Views separate your controller / application logic from your presentation logic and are stored in the `resources/views` directory. A simple view might look something like this:

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
    <body>
        <h1>Hello, {{ $name }}</h1>
    </body>
</html>
```

Since this view is stored at `resources/views/greeting.blade.php`, we may return it using the global `view` helper like so:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

{tip} Looking for more information on how to write Blade templates? Check out the full Blade documentation to get started.

Creating & Rendering Views

You may create a view by placing a file with the `.blade.php` extension in your application's `resources/views` directory. The `.blade.php` extension informs the framework that the file contains a Blade template. Blade templates contain HTML as well as Blade directives that allow you to easily echo values, create "if" statements, iterate over data, and more.

Once you have created a view, you may return it from one of your application's routes or controllers using the global `view` helper:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

Views may also be returned using the `View` facade:

```
use Illuminate\Support\Facades\View;

return View::make('greeting', ['name' => 'James']);
```

As you can see, the first argument passed to the `view` helper corresponds to the name of the view file in the `resources/views` directory. The second argument is an array of data that should be made available to the view. In this case, we are passing the `name` variable, which is displayed in the view using Blade syntax.

Nested View Directories

Views may also be nested within subdirectories of the `resources/views` directory. "Dot" notation may be used to reference nested views. For example, if your view is stored at `resources/views/admin/profile.blade.php`, you may return it from one of your application's routes / controllers like so:

```
return view('admin.profile', $data);
```

{note} View directory names should not contain the `.` character.

Creating The First Available View

Using the `View` facade's `first` method, you may create the first view that exists in a given array of views. This may be useful if your application or package allows views to be customized or overwritten:

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

Determining If A View Exists

If you need to determine if a view exists, you may use the `View` facade. The `exists` method will return `true` if the view exists:

```
use Illuminate\Support\Facades\View;

if (View::exists('emails.customer')) {
    //
}
```

Passing Data To Views

As you saw in the previous examples, you may pass an array of data to views to make that data available to the view:

```
return view('greetings', ['name' => 'Victoria']);
```

When passing information in this manner, the data should be an array with key / value pairs. After providing data to a view, you can then access each value within your view using the data's keys, such as `<?php echo $name; ?>`.

As an alternative to passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view. The `with` method returns an instance of the view object so that you can continue chaining methods before returning the view:

```
return view('greeting')
            ->with('name', 'Victoria')
            ->with('occupation', 'Astronaut');
```

Sharing Data With All Views

Occasionally, you may need to share data with all views that are rendered by your application. You may do so using the `View` facade's `share` method. Typically, you should place calls to the `share` method within a service provider's `boot` method. You are free to add them to the `App\Providers\AppServiceProvider` class or generate a separate service provider to house them:

```php
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        View::share('key', 'value');
    }
}
```

View Composers

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location. View composers may prove particularly useful if the same view is returned by multiple routes or controllers within your application and always needs a particular piece of data.

Typically, view composers will be registered within one of your application's service providers. In this example, we'll assume that we have created a new `App\Providers\ViewServiceProvider` to house this logic.

We'll use the `View` facade's `composer` method to register the view composer. Laravel does not include a default directory for class based view composers, so you are free to organize them however you wish. For example, you could create an `app/View/Composers` directory to house all of your application's view composers:

```php
<?php

namespace App\Providers;

use App\View\Composers\ProfileComposer;
use Illuminate\Support\Facades\View;
use Illuminate\Support\ServiceProvider;

class ViewServiceProvider extends ServiceProvider
{
    /**
     * Register any application services.
     *
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        // Using class based composers...
        View::composer('profile', ProfileComposer::class);

        // Using closure based composers...
        View::composer('dashboard', function ($view) {
            //
        });
    }
}
```

{note} Remember, if you create a new service provider to contain your view composer registrations, you will need to add the service provider to the `providers` array in the `config/app.php` configuration file.

Now that we have registered the composer, the `compose` method of the `App\View\Composers\ProfileComposer` class will be executed each time the `profile` view is being rendered. Let's take a look at an example of the composer class:

```php
<?php

namespace App\View\Composers;

use App\Repositories\UserRepository;
use Illuminate\View\View;

class ProfileComposer
{
    /**
     * The user repository implementation.
     *
     * @var \App\Repositories\UserRepository
     */
    protected $users;

    /**
     * Create a new profile composer.
     *
     * @param  \App\Repositories\UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        // Dependencies are automatically resolved by the service
container...
        $this->users = $users;
    }

    /**
     * Bind data to the view.
     *
     * @param  \Illuminate\View\View  $view
     * @return void
     */
    public function compose(View $view)
    {
        $view->with('count', $this->users->count());
    }
}
```

As you can see, all view composers are resolved via the service container, so you may type-hint any dependencies you need within a composer's constructor.

Attaching A Composer To Multiple Views

You may attach a view composer to multiple views at once by passing an array of views as the first argument to the `composer` method:

```
use App\Views\Composers\MultiComposer;

View::composer(
    ['profile', 'dashboard'],
    MultiComposer::class
);
```

The `composer` method also accepts the `*` character as a wildcard, allowing you to attach a composer to all views:

```
View::composer('*', function ($view) {
    //
});
```

View Creators

View "creators" are very similar to view composers; however, they are executed immediately after the view is instantiated instead of waiting until the view is about to render. To register a view creator, use the `creator` method:

```
use App\View\Creators\ProfileCreator;
use Illuminate\Support\Facades\View;

View::creator('profile', ProfileCreator::class);
```

Optimizing Views

By default, Blade template views are compiled on demand. When a request is executed that renders a view, Laravel will determine if a compiled version of the view exists. If the file exists, Laravel will then determine if the uncompiled view has been modified more recently than the compiled view. If the compiled view either does not exist, or the uncompiled view has been modified, Laravel will recompile the view.

Compiling views during the request may have a small negative impact on performance, so Laravel provides the `view:cache` Artisan command to precompile all of the views utilized by your application. For increased performance, you may wish to run this command as part of your deployment process:

```
php artisan view:cache
```

You may use the `view:clear` command to clear the view cache:

```
php artisan view:clear
```