Laravel 6 Documentation

Sumardi Shukor

Laravel 6 Documentation

Sumardi Shukor

Contents

Installation

Server Requirements

The Laravel framework has a few system requirements. All of these requirements are satisfied by the Laravel Homestead virtual machine, so it's highly recommended that you use Homestead as your local Laravel development environment.

However, if you are not using Homestead, you will need to make sure your server meets the following requirements:

- PHP >= 7.2.0
- BCMath PHP Extension
- Ctype PHP Extension
- JSON PHP Extension
- Mbstring PHP Extension
- OpenSSL PHP Extension
- PDO PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

Installing Laravel

Laravel utilizes Composer[1] to manage its dependencies. So, before using Laravel, make sure you have Composer installed on your machine.

Via Laravel Installer

First, download the Laravel installer using Composer:

```
composer global require laravel/installer
```

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 and GNU / Linux Distributions: `$HOME/.composer/vendor/bin`

[1]https://getcomposer.org

- Windows: `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin`

Once installed, the `laravel new` command will create a fresh Laravel installation in the directory you specify. For instance, `laravel new blog` will create a directory named `blog` containing a fresh Laravel installation with all of Laravel's dependencies already installed:

```
laravel new blog
```

Via Composer Create-Project

Alternatively, you may also install Laravel by issuing the Composer `create-project` command in your terminal:

```
composer create-project --prefer-dist laravel/laravel blog
```

Local Development Server

If you have PHP installed locally and you would like to use PHP's built-in development server to serve your application, you may use the `serve` Artisan command. This command will start a development server at `http://localhost:8000`:

```
php artisan serve
```

Routing

Basic Routing

The most basic Laravel routes accept a URI and a `Closure`, providing a very simple and expressive method of defining routes:

```
Route::get('foo', 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 the framework. 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://your-app.test/user` in your browser:

```
Route::get('/user', 'UserController@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);
```

CSRF Protection

Any HTML forms pointing to `POST`, `PUT`, 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[2]:

```
<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');
```

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']);
```

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:

[2]/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/csrf

```
    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, and may not contain a `-` character. Instead of using the `-` character, use an underscore (_). Route parameters are injected into route callbacks / controllers based on their order - the names of the callback / controller arguments do not matter.

Optional Parameters

Occasionally you may need to specify a route parameter, but make the presence of that route parameter optional. 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;
    });
```

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]+');
```

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:

```php
Route::get('user/profile', function () {
    //
})->name('profile');
```

You may also specify route names for controller actions:

```php
Route::get('user/profile', 'UserProfileController@show')->name('profile');
```

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 the global `route` function:

```php
// 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 URL in their correct positions:

```php
Route::get('user/{id}/profile', function ($id) {
    //
})->name('profile');
$url = route('profile', ['id' => 1]);
```

Route Groups

Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each individual route. Shared attributes are specified in an array format as the first parameter to the `Route::group` method.

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and `where` conditions are merged while names, namespaces, 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:

```php
Route::middleware(['first', 'second'])->group(function () {
    Route::get('/', function () {
        // Uses first & second Middleware
    });
    Route::get('user/profile', function () {
        // Uses first & second Middleware
    });
});
```

Namespaces

Another common use-case for route groups is assigning the same PHP namespace to a group of controllers using the `namespace` method:

```php
Route::namespace('Admin')->group(function () {
    // Controllers Within The "App\Http\Controllers\Admin" Namespace
});
```

Remember, by default, the `RouteServiceProvider` includes your route files within a namespace group, allowing you to register controller routes without specifying the full `App\Http\Controllers` namespace prefix. So, you only need to specify the portion of the namespace that comes after the base `App\Http\Controllers` namespace.

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`:

```php
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 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:

```
Route::get('api/users/{user}', function (App\User $user) {
    return $user->email;
});
```

Since the `$user` variable is type-hinted as the `App\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.

Customizing The Key Name

If you would like model binding to 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';
    }
```

Explicit Binding

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 in the `boot` method of the `RouteServiceProvider` class:

```
    public function boot()
    {
        parent::boot();
        Route::model('user', App\User::class);
    }
```

Next, define a route that contains a `{user}` parameter:

```
    Route::get('profile/{user}', function (App\User $user) {
        //
    });
```

Since we have bound all `{user}` parameters to the `App\User` model, a `User` instance will be injected into the route. So, for example, a request to `profile/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.

Controllers

Introduction

Instead of defining all of your request handling logic as Closures in route files, you may wish to organize this behavior using Controller classes. Controllers can group related request handling logic into a single class. Controllers are stored in the `app/Http/Controllers` directory.

Basic Controllers

Defining Controllers

Below is an example of a basic controller class. Note that the controller extends the base controller class included with Laravel. The base class provides a few convenience methods such as the `middleware` method, which may be used to attach middleware to controller actions:

```php
<?php
namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return View
     */
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

You can define a route to this controller action like so:

```php
Route::get('user/{id}', 'UserController@show');
```

Now, when a request matches the specified route URI, the `show` method on the `UserController` class will be executed. The route parameters will also be passed to the method.

> Controllers are not **required** to extend a base class. However, you will not have access to convenience features such as the `middleware`, `validate`, and `dispatch` methods.

Controllers & Namespaces

It is very important to note that we did not need to specify the full controller namespace when defining the controller route. Since the `RouteServiceProvider` loads your route files within a route group that contains the namespace, we only specified the portion of the class name that comes after the `App\Http\Controllers` portion of the namespace.

If you choose to nest your controllers deeper into the `App\Http\Controllers` directory, use the specific class name relative to the `App\Http\Controllers` root namespace. So, if your full controller class is `App\Http\Controllers\Photos\AdminController`, you should register routes to the controller like so:

```php
Route::get('foo', 'Photos\AdminController@method');
```

Single Action Controllers

If you would like to define a controller that only handles a single action, you may place a single `__invoke` method on the controller:

```php
<?php

namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class ShowProfile extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return View
     */
    public function __invoke($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

When registering routes for single action controllers, you do not need to specify a method:

```
Route::get('user/{id}', 'ShowProfile');
```

You may generate an invokable controller by using the `--invokable` option of the `make:controller` Artisan command:

```
php artisan make:controller ShowProfile --invokable
```

Controller Middleware

Middleware may be assigned to the controller's routes in your route files:

```
Route::get('profile', 'UserController@show')->middleware('auth');
```

However, it is more convenient to specify middleware within your controller's constructor. Using the `middleware` method from your controller's constructor, you may easily assign middleware to the controller's action. You may even restrict the middleware to only certain methods on the controller class:

```
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 a middleware for a single controller without defining an entire middleware class:

```
$this->middleware(function ($request, $next) {
    // ...

    return $next($request);
});
```

You may assign middleware to a subset of controller actions; however, it may indicate your controller is growing too large. Instead, consider breaking your controller into multiple, smaller controllers.

Resource Controllers

Laravel resource routing assigns the typical "CRUD" routes to a controller with a single line of code. For example, you may wish to create a controller that handles all HTTP requests for "photos" stored by your application. Using the `make:controller` Artisan command, we can quickly create such a controller:

```
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 resourceful route to the controller:

```
Route::resource('photos', 'PhotoController');
```

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, including notes informing you of the HTTP verbs and URIs they handle.

You may register many resource controllers at once by passing an array to the `resources` method:

```
Route::resources([
    'photos' => 'PhotoController',
    'posts' => 'PostController'
]);
```

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

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 --resource --model=Photo
```

Spoofing Form Methods

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>
```

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:

```
Route::resource('photos', 'PhotoController')->only([
    'index', 'show'
]);

Route::resource('photos', 'PhotoController')->except([
    'create', 'store', 'update', 'destroy'
]);
```

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
<?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  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }
}
```

You may also type-hint any Laravel contract. If the container can resolve it, you can type-hint it. Depending on your application, injecting your dependencies into your controller may provide better testability.

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
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return 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:

```php
Route::put('user/{id}', 'UserController@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  Request  $request
```

```
         * @param  string  $id
         * @return Response
         */
        public function update(Request $request, $id)
        {
            //
        }
    }
```

Route Caching

> Closure based routes cannot be cached. To use route caching, you must convert any Closure routes to controller classes.

If your application is exclusively using controller based routes, 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. In some cases, your route registration may even be up to 100x faster. To generate a route cache, just 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 Requests

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 controller method. The incoming request instance will automatically be injected by the service container:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

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:

```php
Route::put('user/{id}', 'UserController@update');
```

You may still type-hint the `Illuminate\Http\Request` and access your route parameter `id` 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  Request  $request
     * @param  string  $id
     * @return Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

Accessing The Request Via Route Closures

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:

```php
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
    //
});
```

Request Path & Method

The `Illuminate\Http\Request` instance provides a variety of methods for examining the HTTP request for your application 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://domain.com/foo/bar`, the `path` method will return `foo/bar`:

```
    $uri = $request->path();
```

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/*')) {
        //
    }
```

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:

```
    // Without Query String...
    $url = $request->url();

    // With Query String...
    $url = $request->fullUrl();
```

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')) {
        //
    }
```

Input Trimming & Normalization

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. 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.

Retrieving Input

Retrieving All Input Data

You may also retrieve all of the input data as an `array` using the `all` method:

```
    $input = $request->all();
```

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 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:

```php
    $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:

```php
    $query = $request->query();
```

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:

```php
    $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 route parameters.

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 dig into JSON arrays:

```php
    $name = $request->input('user.name');
```

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:

```php
    $input = $request->only(['username', 'password']);

    $input = $request->only('username', 'password');

    $input = $request->except(['credit_card']);

    $input = $request->except('credit_card');
```

> 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 An Input Value Is Present

You should 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'])) {
        //
    }
```

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')) {
        //
    }
```

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 unlikely you will need to manually use these methods, 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('form')->withInput(
    $request->except('password')
);
```

Retrieving Old Input

To retrieve flashed input from the previous request, use the `old` method on the `Request` instance. The `old` method will pull the previously flashed input data from the session[3]:

```
$username = $request->old('username');
```

Laravel also provides a global `old` helper. If you are displaying old input within a Blade template[4], it is more convenient to use the `old` helper. 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 a `Illuminate\Http\Request` instance:

```
$value = $request->cookie('name');
```

Alternatively, you may use the `Cookie` facade to access cookie values:

```
use Illuminate\Support\Facades\Cookie;

$value = Cookie::get('name');
```

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 number of minutes the cookie should be considered valid to this method:

[3]/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/session
[4]/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/blade

```
    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[5] method:

```
    return response('Hello World')->cookie(
        'name', 'value', $minutes, $path, $domain, $secure, $httpOnly
    );
```

Generating Cookie Instances

If you would like to generate a `Symfony\Component\HttpFoundation\Cookie` instance that can be given 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);
```

Files

Retrieving Uploaded Files

You may access uploaded files from a `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:

[5] https://secure.php.net/manual/en/function.setcookie.php

```php
    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:

```php
    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:

```php
    $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[6] 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 which will move an uploaded file to one of your disks, which may be a location on your local filesystem or even 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 file name, since a unique ID will automatically be generated to serve as the file name.

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:

[6]https://api.symfony.com/3.0/Symfony/Component/HttpFoundation/File/UploadedFile.html

```php
$path = $request->photo->store('images');

$path = $request->photo->store('images', 's3');
```

If you do not want a file name to be automatically generated, you may use the `storeAs` method, which accepts the path, file name, and disk name as its arguments:

```php
$path = $request->photo->storeAs('images', 'filename.jpg');

$path = $request->photo->storeAs('images', 'filename.jpg', 's3');
```

HTTP Responses

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];
});
```

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');
});
```

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',
            ]);
```

Attaching Cookies To Responses

The `cookie` method on response instances allows you to easily attach cookies to the response. For example, you may use the `cookie` method to generate a cookie and fluently attach it to the response instance like so:

```
return response($content)
                ->header('Content-Type', $type)
                ->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[7] method:

[7]https://secure.php.net/manual/en/function.setcookie.php

```
    ->cookie($name, $value, $minutes, $path, $domain, $secure, $httpOnly)
```

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 or has all of the session middleware applied:

```
    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 should 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. Remember, you do not need to specify the full namespace to the controller since Laravel's `RouteServiceProvider` will automatically set the base controller namespace:

```
    return redirect()->action('HomeController@index');
```

If your controller route requires parameters, you may pass them as the second argument to the `action` method:

```
return redirect()->action(
    'UserController@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 () {
    // Update the user's profile...

    return redirect('dashboard')->with('status', 'Profile updated!');
});
```

After the user is redirected, you may display the flashed message from the session[8]. For example, using Blade syntax[9]:

```
@if (session('status'))
    <div class="alert alert-success">
        {{ session('status') }}
    </div>
@endif
```

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[10] is returned. This contract provides several helpful methods for generating responses.

[8] /docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/session
[9] /docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/blade
[10] /docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/contracts

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 should 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 file name as the second argument to the method, which will determine the file name 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);

return response()->download($pathToFile)->deleteFileAfterSend();
```

Symfony HttpFoundation, which manages file downloads, requires the file being downloaded to have an ASCII file name.

Views

Creating Views

> Looking for more information on how to write Blade templates? Check out the full Blade documentation to get started.

Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views 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']);
});
```

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.

Views may also be nested within sub-directories 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.b` you may reference it like so:

```
return view('admin.profile', $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')) {
    //
}
```

Creating The First Available View

Using the `first` method, you may create the first view that exists in a given array of views. This is useful if your application or package allows views to be customized or overwritten:

```
return view()->first(['custom.admin', 'admin'], $data);
```

You may also call this method via the `View` facade:

```
use Illuminate\Support\Facades\View;

return View::first(['custom.admin', 'admin'], $data);
```

Passing Data To Views

As you saw in the previous examples, you may pass an array of data to views:

```
return view('greetings', ['name' => 'Victoria']);
```

When passing information in this manner, the data should be an array with key / value pairs. Inside your view, you can then access each value using its corresponding key, such as `<?php echo $key; ?>`. 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:

```
return view('greeting')->with('name', 'Victoria');
```

Sharing Data With All Views

Occasionally, you may need to share a piece of 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 `share` within a service provider's `boot` method. You are free to add them to the `AppServiceProvider` 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');
    }
}
```

URL Generation

Introduction

Laravel provides several helpers to assist you in generating URLs for your application. These are mainly helpful when building links in your templates and API responses, or when generating redirect responses to another part of your application.

The Basics

Generating Basic 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:

```
$post = App\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, a `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 `route` function calls. For example, imagine your application contains a route defined like the following:

```
Route::get('/post/{post}', function () {
    //
})->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
```

You will often be generating URLs using 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 primary key:

```
echo route('post.show', ['post' => $post]);
```

The `route` helper may also be used to generate URLs for routes with multiple parameters:

```
Route::get('/post/{post}/comment/{comment}', function () {
    //
})->name('comment.show');

echo route('comment.show', ['post' => 1, 'comment' => 3]);

// http://example.com/post/1/comment/3
```

URLs For Controller Actions

The `action` function generates a URL for the given controller action. You do not need to pass the full namespace of the controller. Instead, pass the controller class name relative to the `App\Http\Controllers` namespace:

```
$url = action('HomeController@index');
```

You may also reference actions with a “callable” array syntax:

```
use App\Http\Controllers\HomeController;

$url = action([HomeController::class, 'index']);
```

If the controller method accepts route parameters, you may pass them as the second argument to the function:

```
$url = action('UserController@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
{
    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.

HTTP Session

Introduction

Since HTTP driven applications are stateless, sessions provide a way to store information about the user across multiple requests. Laravel ships with a variety of session backends that are accessed through an expressive, unified API. Support for popular backends such as Memcached[11], Redis[12], and databases is included out of the box.

Configuration

The 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. In production applications, you may consider using the `memcached` or `redis` drivers for even faster session performance.

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.
- `array` - sessions are stored in a PHP array and will not be persisted.

> The array driver is 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 items. Below is an example `Schema` declaration for the table:

[11]https://memcached.org
[12]https://redis.io

```
Schema::create('sessions', function ($table) {
    $table->string('id')->unique();
    $table->unsignedInteger('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
});
```

You may use the `session:table` Artisan command to generate this migration:

```
php artisan session:table


php artisan migrate
```

Redis

Before using Redis sessions with Laravel, you will need to install the `predis/predis` package (~1.0) via Composer. You may configure your Redis connections in the `database` configuration file. In the `session` configuration file, the `connection` option may be used to specify which Redis connection is used by the session.

Using 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 controller method. Remember, controller method dependencies are automatically injected via the Laravel service container:

```
<?php

namespace App\Http\Controllers;

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

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']);
});
```

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:

```php
$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`:

```php
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. The `exists` method returns `true` if the item is present:

```php
if ($request->session()->exists('users')) {
    //
}
```

Storing Data

To store data in the session, you will typically use the `put` method or the `session` helper:

```php
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global 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');
```

Flash Data

Sometimes you may wish to store items in the session only for the next request. You may do so using the `flash` method. Data stored in the session using this method will only be available during the subsequent HTTP request, and then will be deleted. Flash data is primarily useful for short-lived status messages:

```
$request->session()->flash('status', 'Task was successful!');
```

If you need to keep your flash data around 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']);
```

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('key');

// Forget multiple keys...
$request->session()->forget(['key1', 'key2']);

$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[13] attack on your application.

Laravel automatically regenerates the session ID during authentication if you are using the built-in `LoginController`; however, if you need to manually regenerate the session ID, you may use the `regenerate` method.

```
    $request->session()->regenerate();
```

[13]https://en.wikipedia.org/wiki/Session_fixation

Validation

Introduction

Laravel provides several different approaches to validate your application's incoming data. By default, Laravel's base controller class uses a `ValidatesRequests` trait which provides a convenient method to validate incoming HTTP request with a variety of powerful validation rules.

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.

Defining The Routes

First, let's assume we have the following routes defined in our `routes/web.php` file:

```
Route::get('post/create', 'PostController@create');

Route::post('post', 'PostController@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 these routes. We'll leave the `store` method empty for now:

```php
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Show the form to create a new blog post.
     *
     * @return Response
     */
    public function create()
    {
        return view('post.create');
    }

    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return 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 exception will be thrown and the proper error response will automatically be sent back to the user. In the case of a traditional HTTP request, a redirect response will be generated, while a JSON response will be sent for AJAX requests.

To get a better understanding of the `validate` method, let's jump back into the `store` method:

```
    /**
     * Store a new blog post.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $validatedData = $request->validate([
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ]);

        // The blog post is valid...
    }
```

As you can see, we pass the desired validation rules into the `validate` method. 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'],
    ]);
```

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 your HTTP request contains "nested" parameters, you may specify them in your validation rules using "dot" syntax:

```
$request->validate([
    'title' => 'required|unique:posts|max:255',
    'author.name' => 'required',
    'author.description' => 'required',
]);
```

Displaying The Validation Errors

So, what if the incoming request parameters 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 will automatically be flashed to the session.

Again, notice that we did not have to explicitly bind the error messages to the view in our `GET` route. This is because Laravel will check for errors in the session data, and automatically bind them to the view if they are available. The `$errors` variable will be an instance of `Illuminate\Support\MessageBag`. For more information on working with this object, check out its documentation.

> The `$errors` variable is bound to the view by the `Illuminate\View\Middleware\ShareErrorsFromSession` middleware, which is provided by the `web` middleware group. **When this middleware is applied an `$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.

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 -->
```

The `@error` Directive

You may also use the `@error` Blade directive 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
```

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.

AJAX Requests & Validation

In this example, we used a traditional form to send data to the application. However, many applications use AJAX requests. When using the `validate` method during an AJAX 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.

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 contain validation logic. To create a form request class, use the `make:request` Artisan CLI command:

```
php artisan make:request StoreBlogPost
```

The generated 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. Let's add a few validation rules to the `rules` method:

```
    /**
     * Get the validation rules that apply to the request.
     *
     * @return array
     */
    public function rules()
    {
        return [
            'title' => 'required|unique:posts|max:255',
            'body' => 'required',
        ];
    }
```

You may type-hint any dependencies you need within the `rules` method's signature. They will automatically be resolved via the Laravel service container[14].

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:

[14]/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/container

```
    /**
     * Store the incoming blog post.
     *
     * @param  StoreBlogPost  $request
     * @return Response
     */
    public function store(StoreBlogPost $request)
    {
        // The incoming request is valid...

        // Retrieve the validated input data...
        $validated = $request->validated();
    }
```

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 AJAX request, a HTTP response with a 422 status code will be returned to the user including a JSON representation of the validation errors.

Authorizing Form Requests

The form request class also contains an `authorize` method. Within this method, you may check 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:

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

If the `authorize` method returns `false`, a HTTP response with a 403 status code will automatically be returned and your controller method will not execute.

If you plan to have authorization logic in another part of your application, return `true` from the `authorize` method:

```
/**
 * Determine if the user is authorized to make this request.
 *
 * @return bool
 */
public function authorize()
{
    return true;
}
```

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

If you would like the `:attribute` portion 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',
    ];
}
```

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

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
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();
            }

            // Store the blog post...
        }
    }
```

The first argument passed to the `make` method is the data under validation. The second argument is the validation rules that should be applied to the data.

After checking if 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`.

Automatic Redirection

If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the requests'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 AJAX request, a JSON response will be returned:

```
Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
])->validate();
```

Named Error Bags

If you have multiple forms on a single page, you may wish to name the `MessageBag` of errors, allowing you to retrieve the error messages for a specific form. 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') }}
```

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')) {
        //
    }
```

Custom Error Messages

If needed, you may use custom error messages for validation instead of the defaults. There are several ways to specify custom messages. First, you may pass the custom messages as the third argument to the `Validator::make` method:

```
    $messages = [
        'required' => 'The :attribute field is required.',
    ];

    $validator = Validator::make($input, $rules, $messages);
```

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 field. 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 e-mail address!',
    ];
```

Specifying Custom Messages In Language Files

In most cases, you will probably specify your custom messages in a language file instead of passing them directly to the `Validator`. To do so, add your messages to `custom` array in the `resources/lang/xx/validation.php` language file.

```
    'custom' => [
        'email' => [
            'required' => 'We need to know your e-mail address!',
        ],
    ],
```

Specifying Custom Attributes In Language Files

If you would like the `:attribute` portion of your validation message to be replaced with a custom attribute name, you may specify the custom name in the `attributes` array of your `resources/lang/xx/validation.php` language file:

```
    'attributes' => [
        'email' => 'email address',
    ],
```

Specifying Custom Values In Language Files

Sometimes you may 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`:

```
    $request->validate([
        '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 custom value representation in your `validation` language file by defining a `values` array:

```
    'values' => [
        'payment_type' => [
            'cc' => 'credit card'
        ],
    ],
```

Now if the validation rule fails it will produce the following 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[15]
Active URL[16]
After (Date)[17]
After Or Equal (Date)[18]
Alpha[19]
Alpha Dash[20]
Alpha Numeric[21]
Array[22]
Bail[23]
Before (Date)[24]
Before Or Equal (Date)[25]
Between[26]
Boolean[27]
Confirmed[28]
Date[29]
Date Equals[30]
Date Format[31]
Different[32]
Digits[33]
Digits Between[34]
Dimensions (Image Files)[35]
Distinct[36]
E-Mail[37]

[15]https://laravel.com/docs/6.0/validation#rule-accepted
[16]https://laravel.com/docs/6.0/validation#rule-active-url
[17]https://laravel.com/docs/6.0/validation#rule-after
[18]https://laravel.com/docs/6.0/validation#rule-after-or-equal
[19]https://laravel.com/docs/6.0/validation#rule-alpha
[20]https://laravel.com/docs/6.0/validation#rule-alpha-dash
[21]https://laravel.com/docs/6.0/validation#rule-alpha-num
[22]https://laravel.com/docs/6.0/validation#rule-array
[23]https://laravel.com/docs/6.0/validation#rule-bail
[24]https://laravel.com/docs/6.0/validation#rule-before
[25]https://laravel.com/docs/6.0/validation#rule-before-or-equal
[26]https://laravel.com/docs/6.0/validation#rule-between
[27]https://laravel.com/docs/6.0/validation#rule-boolean
[28]https://laravel.com/docs/6.0/validation#rule-confirmed
[29]https://laravel.com/docs/6.0/validation#rule-date
[30]https://laravel.com/docs/6.0/validation#rule-date-equals
[31]https://laravel.com/docs/6.0/validation#rule-date-format
[32]https://laravel.com/docs/6.0/validation#rule-different
[33]https://laravel.com/docs/6.0/validation#rule-digits
[34]https://laravel.com/docs/6.0/validation#rule-digits-between
[35]https://laravel.com/docs/6.0/validation#rule-dimensions
[36]https://laravel.com/docs/6.0/validation#rule-distinct
[37]https://laravel.com/docs/6.0/validation#rule-email

Ends With[38]
Exists (Database)[39]
File[40]
Filled[41]
Greater Than[42]
Greater Than Or Equal[43]
Image (File)[44]
In[45]
In Array[46]
Integer[47]
IP Address[48]
JSON[49]
Less Than[50]
Less Than Or Equal[51]
Max[52]
MIME Types[53]
MIME Type By File Extension[54]
Min[55]
Not In[56]
Not Regex[57]
Nullable[58]
Numeric[59]
Present[60]
Regular Expression[61]
Required[62]

[38]https://laravel.com/docs/6.0/validation#rule-ends-with
[39]https://laravel.com/docs/6.0/validation#rule-exists
[40]https://laravel.com/docs/6.0/validation#rule-file
[41]https://laravel.com/docs/6.0/validation#rule-filled
[42]https://laravel.com/docs/6.0/validation#rule-gt
[43]https://laravel.com/docs/6.0/validation#rule-gte
[44]https://laravel.com/docs/6.0/validation#rule-image
[45]https://laravel.com/docs/6.0/validation#rule-in
[46]https://laravel.com/docs/6.0/validation#rule-in-array
[47]https://laravel.com/docs/6.0/validation#rule-integer
[48]https://laravel.com/docs/6.0/validation#rule-ip
[49]https://laravel.com/docs/6.0/validation#rule-json
[50]https://laravel.com/docs/6.0/validation#rule-lt
[51]https://laravel.com/docs/6.0/validation#rule-lte
[52]https://laravel.com/docs/6.0/validation#rule-max
[53]https://laravel.com/docs/6.0/validation#rule-mimetypes
[54]https://laravel.com/docs/6.0/validation#rule-mimes
[55]https://laravel.com/docs/6.0/validation#rule-min
[56]https://laravel.com/docs/6.0/validation#rule-not-in
[57]https://laravel.com/docs/6.0/validation#rule-not-regex
[58]https://laravel.com/docs/6.0/validation#rule-nullable
[59]https://laravel.com/docs/6.0/validation#rule-numeric
[60]https://laravel.com/docs/6.0/validation#rule-present
[61]https://laravel.com/docs/6.0/validation#rule-regex
[62]https://laravel.com/docs/6.0/validation#rule-required

Required If[63]
Required Unless[64]
Required With[65]
Required With All[66]
Required Without[67]
Required Without All[68]
Same[69]
Size[70]
Sometimes[71]
Starts With[72]
String[73]
Timezone[74]
Unique (Database)[75]
URL[76]
UUID[77]

Conditionally Adding Rules

Validating When Present

In some situations, you may wish to run validation checks against a field **only** if that field is present in the input array. 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.

> If you are attempting to validate a field that should always be present but may be empty, check out this note on optional fields

[63]https://laravel.com/docs/6.0/validation#rule-required-if
[64]https://laravel.com/docs/6.0/validation#rule-required-unless
[65]https://laravel.com/docs/6.0/validation#rule-required-with
[66]https://laravel.com/docs/6.0/validation#rule-required-with-all
[67]https://laravel.com/docs/6.0/validation#rule-required-without
[68]https://laravel.com/docs/6.0/validation#rule-required-without-all
[69]https://laravel.com/docs/6.0/validation#rule-same
[70]https://laravel.com/docs/6.0/validation#rule-size
[71]https://laravel.com/docs/6.0/validation#conditionally-adding-rules
[72]https://laravel.com/docs/6.0/validation#rule-starts-with
[73]https://laravel.com/docs/6.0/validation#rule-string
[74]https://laravel.com/docs/6.0/validation#rule-timezone
[75]https://laravel.com/docs/6.0/validation#rule-unique
[76]https://laravel.com/docs/6.0/validation#rule-url
[77]https://laravel.com/docs/6.0/validation#rule-uuid

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:

```
    $v = Validator::make($data, [
        '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. To conditionally add this requirement, we can use the `sometimes` method on the `Validator` instance.

```
    $v->sometimes('reason', 'required|max:500', function ($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 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:

```
    $v->sometimes(['reason', 'cost'], 'required', function ($input) {
        return $input->games >= 100;
    });
```

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.

Blade Templates

Introduction

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

Template Inheritance

Defining A Layout

Two of the primary benefits of using Blade are template inheritance and sections. To get started, let's take a look at a simple example. First, we will examine a "master" 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:

```
<!-- Stored in 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 Blade `@extends` 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`:

```
<!-- Stored in 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.

> 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', View::make('view.name'))
```

Blade views may be returned from routes using the global `view` helper:

```
Route::get('blade', function () {
    return view('child');
});
```

Components & Slots

Components and slots provide similar benefits to sections and layouts; however, some may find the mental model of components and slots easier to understand. First, let's imagine a reusable "alert" component we would like to reuse throughout our application:

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

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

The `{{ $slot }}` variable will contain the content we wish to inject into the component. Now, to construct this component, we can use the `@component` Blade directive:

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

To instruct Laravel to load the first view that exists from a given array of possible views for the component, you may use the `componentFirst` directive:

```
@componentFirst(['custom.alert', 'alert'])
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Sometimes it is helpful to define multiple slots for a component. Let's modify our alert component to allow for the injection of a "title". Named slots may be displayed by "echoing" the variable that matches their name:

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

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

    {{ $slot }}
</div>
```

Now, we can inject content into the named slot using the `@slot` directive. Any content not within a `@slot` directive will be passed to the component in the `$slot` variable:

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

Passing Additional Data To Components

Sometimes you may need to pass additional data to a component. For this reason, you can pass an array of data as the second argument to the `@component` directive. All of the data will be made available to the component template as variables:

```
@component('alert', ['foo' => 'bar'])
    ...
@endcomponent
```

Aliasing Components

If your Blade components are stored in a sub-directory, you may wish to alias them for easier access. For example, imagine a Blade component that is stored at `resources/views/components/alert.blade.php`. You may use the `component` method to alias the component from `components.alert` to `alert`. Typically, this should be done in the `boot` method of your `AppServiceProvider`:

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

Blade::component('components.alert', 'alert');
```

Once the component has been aliased, you may render it using a directive:

```
@alert(['type' => 'danger'])
    You are not allowed to access this resource!
@endalert
```

You may omit the component parameters if it has no additional slots:

```
@alert
    You are not allowed to access this resource!
@endalert
```

Displaying Data

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

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

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

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

Blade `{{ }}` 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() }}.
```

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 !!}.
```

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

Control Structures

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
```

Section Directives

You may check if a section has content using the `@hasSection` directive:

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

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

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:

```
@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 condition with the directive declaration in one line:

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

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

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

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 --}}
```

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
    //
@endphp
```

While Blade provides this feature, using it frequently may be a signal that you have too much logic embedded within your template.

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[78] 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[79] exist for a given attribute. Within an `@error` directive, you may echo the `$message` variable to display the error message:

[78]https://laravel.com/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/csrf
[79]/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/validation#quick-displaying-the-validation-errors

```
<!-- /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
```

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 @ender\
ror">

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

Including Sub-Views

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 extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

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', ['some' => 'data'])
```

If you would like to `@include` a view depending on a given boolean condition, you may use the `@includeWhen` directive:

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

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

```
@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])
```

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.

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
```

Authentication

Introduction

> **Want to get started fast?** Install the `laravel/ui` Composer package and run `php artisan ui vue --auth` in a fresh Laravel application. After migrating your database, navigate your browser to `http://your-app.test/register` or any other URL that is assigned to your application. These commands will take care of scaffolding your entire authentication system!

Laravel makes implementing authentication very simple. In fact, almost everything is configured for you out of the box. The authentication configuration file is located at `config/auth.php`, which contains several well documented options for tweaking the behavior of the authentication services.

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.

Don't worry if this all sounds confusing now! Many applications will never need to modify the default authentication configuration.

Database Considerations

By default, Laravel includes an `App\User` Eloquent model in your `app` 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 driver which uses the Laravel query builder.

When building the database schema for the `App\User` model, make sure the password column is at least 60 characters in length. Maintaining the default string column length of 255 characters would be a good choice.

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.

Authentication Quickstart

Laravel ships with several pre-built authentication controllers, which are located in the `App\Http\Controllers\Auth` namespace. The `RegisterController` handles new user registration, the `LoginController` handles authentication, the `ForgotPasswordController` handles e-mailing links for resetting passwords, and the `ResetPasswordController` contains the logic to reset passwords. Each of these controllers uses a trait to include their necessary methods. For many applications, you will not need to modify these controllers at all.

Routing

Laravel's `laravel/ui` package provides a quick way to scaffold all of the routes and views you need for authentication using a few simple commands:

```
composer require laravel/ui --dev


php artisan ui vue --auth
```

This command should be used on fresh applications and will install a layout view, registration and login views, as well as routes for all authentication end-points. A `HomeController` will also be generated to handle post-login requests to your application's dashboard.

> If your application doesn't need registration, you may disable it by removing the newly created `RegisterController` and modifying your route declaration: `Auth::routes(['register' => false]);`.

Views

As mentioned in the previous section, the `laravel/ui` package's `php artisan ui vue --auth` command will create all of the views you need for authentication and place them in the `resources/views/auth` directory.

The `ui` command will also create a `resources/views/layouts` directory containing a base layout for your application. All of these views use the Bootstrap CSS framework, but you are free to customize them however you wish.

Authenticating

Now that you have routes and views setup for the included authentication controllers, you are ready to register and authenticate new users for your application! You may access your application in a browser since the authentication controllers already contain the logic (via their traits) to authenticate existing users and store new users in the database.

Path Customization

When a user is successfully authenticated, they will be redirected to the `/home` URI. You can customize the post-authentication redirect location by defining a `redirectTo` property on the `LoginController`, `RegisterController`, `ResetPasswordController`, and `VerificationController`:

```
    protected $redirectTo = '/';
```

Next, you should modify the `RedirectIfAuthenticated` middleware's `handle` method to use your new URI when redirecting the user.

If the redirect path needs custom generation logic you may define a `redirectTo` method instead of a `redirectTo` property:

```
    protected function redirectTo()
    {
        return '/path';
    }
```

The `redirectTo` method will take precedence over the `redirectTo` property.

Username Customization

By default, Laravel uses the `email` field for authentication. If you would like to customize this, you may define a `username` method on your `LoginController`:

```
    public function username()
    {
        return 'username';
    }
```

Guard Customization

You may also customize the "guard" that is used to authenticate and register users. To get started, define a `guard` method on your `LoginController`, `RegisterController`, and `ResetPasswordController`. The method should return a guard instance:

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

protected function guard()
{
    return Auth::guard('guard-name');
}
```

Validation / Storage Customization

To modify the form fields that are required when a new user registers with your application, or to customize how new users are stored into your database, you may modify the `RegisterController` class. This class is responsible for validating and creating new users of your application.

The `validator` method of the `RegisterController` contains the validation rules for new users of the application. You are free to modify this method as you wish.

The `create` method of the `RegisterController` is responsible for creating new `App\User` records in your database using the Eloquent ORM. You are free to modify this method according to the needs of your database.

Retrieving The Authenticated User

You may access the authenticated user via the `Auth` facade:

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

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

// Get 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:

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProfileController extends Controller
{
    /**
     * Update the user's profile.
     *
     * @param  Request  $request
     * @return Response
     */
    public function update(Request $request)
    {
        // $request->user() returns an instance of the authenticated user...
    }
}
```

Determining If The Current User Is Authenticated

To determine if the user is already logged into your application, you may use the `check` method on the `Auth` facade, which will return `true` if the user is authenticated:

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

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

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 is defined at `Illuminate\Auth\Middleware\Authenticate`. Since this middleware is already registered in your HTTP kernel, all you need to do is attach the middleware to a route definition:

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

If you are using controllers, you may call the `middleware` method from the controller's constructor instead of attaching it in the route definition directly:

```
public function __construct()
{
    $this->middleware('auth');
}
```

Redirecting Unauthenticated Users

When the `auth` middleware detects an unauthorized user, it will redirect the user to the `login` named route. You may modify this behavior by updating the `redirectTo` function in your `app/Http/Middleware/Authenticate.` 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 Additional Conditions

If you wish, you may also add extra conditions to the authentication query in addition to the user's e-mail and password. For example, we may verify that user is marked as "active":

```
if (Auth::attempt(['email' => $email, 'password' => $password, 'active' => 1])) {
    // The user is active, not suspended, and exists.
}
```

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.

Logging Out

To log users out of your application, you may use the `logout` method on the `Auth` facade. This will clear the authentication information in the user's session:

```
Auth::logout();
```

Remembering Users

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, which 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.

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

If you are using the built-in `LoginController` that is shipped with Laravel, the proper logic to “remember” users is already implemented by the traits used by the controller.

If you are “remembering” users, you may use the `viaRemember` method to determine if the user was authenticated using the “remember me” cookie:

```
if (Auth::viaRemember()) {
    //
}
```

Authorization

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 `Post` model and a corresponding `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 the basic "CRUD" policy methods already included in the class, you may specify a `--model` when executing the command:

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

> 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.

Registering Policies

Once the policy exists, it needs to be registered. The `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 model:

```php
<?php

namespace App\Providers;

use App\Post;
use App\Policies\PostPolicy;
use Illuminate\Support\Facades\Gate;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvid\
er;

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 auto-discover policies as long as the model and policy follow standard Laravel naming conventions. Specifically, the policies must be in a `Policies` directory below the directory that contains the models. So, for example, the models may be placed in the `app` directory while the policies may be placed in the `app/Policies` directory. In addition, the policy name must match the model name and have a `Policy` suffix. So, a `User` model would correspond to a `UserPolicy` class.

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

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

Gate::guessPolicyNamesUsing(function ($modelClass) {
    // return policy class name...
});
```

Any policies that are explicitly mapped in your `AuthServiceProvider` will take precedence over any potential auto-discovered policies.

Writing Policies

Policy Methods

Once the policy 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 `User` can update a given `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, for this example, let's verify that the user's `id` matches the `user_id` on the post:

```php
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\User  $user
     * @param  \App\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` 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 `view`, `create`, `update`, `delete`, `restore`, and `forceDelete` actions.

Policy Responses

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

```
use Illuminate\Auth\Access\Response;

/**
 * Determine if the given post can be updated by the user.
 *
 * @param  \App\User  $user
 * @param  \App\Post  $post
 * @return bool
 */
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 use the `Gate::inspect` method to get the full authorization response returned by the gate:

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

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

Of course, when using the `Gate::authorize` method to throw 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 the currently authenticated user and not an instance of the model they authorize. This situation is most common when authorizing `create` actions. For example, if you are creating a blog, you may wish to check if a user is authorized to create any posts at all.

When defining policy methods that will not receive a model instance, such as a `create` method, it will not receive a model instance. Instead, you should define the method as only expecting the authenticated user:

```
/**
 * Determine if the given user can create posts.
 *
 * @param  \App\User  $user
 * @return bool
 */
public function create(User $user)
{
    //
}
```

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
<?php

namespace App\Policies;

use App\User;
use App\Post;

class PostPolicy
{
    /**
     * Determine if the given post can be updated by the user.
     *
     * @param  \App\User  $user
     * @param  \App\Post  $post
     * @return bool
     */
    public function update(?User $user, Post $post)
    {
        return $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:

```php
public function before($user, $ability)
{
    if ($user->isSuperAdmin()) {
        return true;
    }
}
```

If you would like to deny all authorizations for a user you should return `false` from the `before` method. If `null` is returned, the authorization will fall through to the policy method.

> 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 `User` model that is included with your Laravel application includes two helpful methods for authorizing actions: `can` and `cant`. The `can` method receives the action you wish to authorize and the relevant model. For example, let's determine if a user is authorized to update a given `Post` model:

```
if ($user->can('update', $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 like `create` may 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:

```
use App\Post;

if ($user->can('create', Post::class)) {
    // Executes the "create" method on the relevant policy...
}
```

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 blog post:

```
use App\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 `Post` model will be passed to the policy method. If the user is not authorized to perform the given action, a HTTP response with a `403` status code will be generated by the middleware.

Actions That Don't Require Models

Again, some actions like `create` may 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\Post');
```

Via Controller Helpers

In addition to helpful methods provided to the `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`, which the default Laravel exception handler will convert to an HTTP response with a `403` status code:

```
<?php

namespace App\Http\Controllers;

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

class PostController extends Controller
{
    /**
     * Update the given blog post.
     *
     * @param  Request  $request
     * @param  Post  $post
     * @return 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 actions like `create` may 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:

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

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

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` family of directives:

```
    @can('update', $post)
        <!-- The Current User Can Update The Post -->
    @elsecan('create', App\Post::class)
        <!-- The Current User Can Create New Post -->
    @endcan

    @cannot('update', $post)
        <!-- The Current User Can't Update The Post -->
    @elsecannot('create', App\Post::class)
        <!-- The Current User Can't Create New Post -->
    @endcannot
```

These directives are convenient shortcuts for writing `@if` and `@unless` statements. The `@can` and `@cannot` statements above respectively translate 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 Can't Update The Post -->
    @endunless
```

You may also determine if a user has any authorization ability from a given list of abilities. To accomplish this, use the `@canany` directive:

@canany([‘update’, ‘view’, ‘delete’], $post)
// The current user can update, view, or delete the post
@elsecanany([‘create’], AppPost::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\Post::class)
    <!-- The Current User Can Create Posts -->
@endcan

@cannot('create', App\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\User  $user
 * @param  \App\Post  $post
 * @param  int  $category
 * @return bool
 */
public function update(User $user, Post $post, int $category)
{
    return $user->id === $post->user_id &&
           $category > 3;
}
```

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

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

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

Database: Migrations

Introduction

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema. If you have ever had to tell a teammate to manually add a column to their local database schema, you've faced the problem that database migrations solve.

The Laravel `Schema` facade provides database agnostic support for creating and manipulating tables across all of Laravel's supported database systems.

Generating Migrations

To create a migration, use the `make:migration` Artisan command:

```
php artisan make:migration create_users_table
```

The new migration will be placed in your `database/migrations` directory. Each migration file name contains a timestamp which allows Laravel to determine the order of the migrations.

The `--table` and `--create` options may also be used to indicate the name of the table and whether the migration will be creating a new table. These options pre-fill the generated migration stub file with the specified table:

```
php artisan make:migration create_users_table --create=users
```

```
php artisan make:migration add_votes_to_users_table --table=users
```

If you would like to specify a custom output path for the generated migration, you may use the `--path` option when executing the `make:migration` command. The given path should be relative to your application's base path.

Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should reverse the operations performed by the `up` method.

Within both of these methods you may use the Laravel schema builder to expressively create and modify tables. To learn about all of the methods available on the `Schema` builder, check out its documentation. For example, this migration example creates a `flights` table:

```php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateFlightsTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::drop('flights');
    }
}
```

Running Migrations

To run all of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

> If you are using the Homestead virtual machine, you should run this command from within your virtual machine.

Forcing Migrations To Run In Production

Some migration operations are destructive, which means they may cause you to lose data. In order to protect you from running these commands against your production database, you will be prompted for confirmation before the commands are executed. To force the commands to run without a prompt, use the `--force` flag:

```
php artisan migrate --force
```

Rolling Back Migrations

To rollback the latest migration operation, you may use the `rollback` command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may rollback a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will rollback the last five migrations:

```
php artisan migrate:rollback --step=5
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Rollback & Migrate In Single Command

The `migrate:refresh` command will roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh
```

// Refresh the database and run all database seeds...

```
php artisan migrate:refresh --seed
```

You may rollback & re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will rollback & re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Drop All Tables & Migrate

The `migrate:fresh` command will drop all tables from the database and then execute the `migrate` command:

```
php artisan migrate:fresh
```

```
php artisan migrate:fresh --seed
```

Tables

Creating Tables

To create a new database table, use the `create` method on the `Schema` facade. The `create` method accepts two arguments. The first is the name of the table, while the second is a `Closure` which receives a `Blueprint` object that may be used to define the new table:

```
    Schema::create('users', function (Blueprint $table) {
        $table->bigIncrements('id');
    });
```

When creating the table, you may use any of the schema builder's column methods to define the table's columns.

Checking For Table / Column Existence

You may easily check for the existence of a table or column using the `hasTable` and `hasColumn` methods:

```
if (Schema::hasTable('users')) {
    //
}

if (Schema::hasColumn('users', 'email')) {
    //
}
```

Database Connection & Table Options

If you want to perform a schema operation on a database connection that is not your default connection, use the `connection` method:

Schema::connection(‘foo’)->create(‘users’, function (Blueprint $table) {
$table->bigIncrements(‘id’);
});

You may use the following commands on the schema builder to define the table’s options:

Command	Description
`$table->engine = 'InnoDB';`	Specify the table storage engine (MySQL).
`$table->charset = 'utf8';`	Specify a default character set for the table (MySQL).
`$table->collation = 'utf8_unicode_ci';`	Specify a default collation for the table (MySQL).
`$table->temporary();`	Create a temporary table (except SQL Server).

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Renaming Tables With Foreign Keys

Before renaming a table, you should verify that any foreign key constraints on the table have an explicit name in your migration files instead of letting Laravel assign a convention based name. Otherwise, the foreign key constraint name will refer to the old table name.

Columns

Creating Columns

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a `Closure` that receives a `Blueprint` instance you may use to add columns to the table:

```
    Schema::table('users', function (Blueprint $table) {
        $table->string('email');
    });
```

Available Column Types

The schema builder contains a variety of column types that you may specify when building your tables:

Command	Description
`$table->bigIncrements('id');`	Auto-incrementing UNSIGNED BIGINT (primary key) equivalent column.
`$table->bigInteger('votes');`	BIGINT equivalent column.
`$table->binary('data');`	BLOB equivalent column.
`$table->boolean('confirmed');`	BOOLEAN equivalent column.
`$table->char('name', 100);`	CHAR equivalent column with an optional length.
`$table->date('created_at');`	DATE equivalent column.
`$table->dateTime('created_at');`	DATETIME equivalent column.
`$table->dateTimeTz('created_at');`	DATETIME (with timezone) equivalent column.
`$table->decimal('amount', 8, 2);`	DECIMAL equivalent column with a precision (total digits) and scale (decimal digits).
`$table->double('amount', 8, 2);`	DOUBLE equivalent column with a precision (total digits) and scale (decimal digits).
`$table->enum('level', ['easy', 'hard']);`	ENUM equivalent column.
`$table->float('amount', 8, 2);`	FLOAT equivalent column with a precision (total digits) and scale (decimal digits).
`$table->geometry('positions');`	GEOMETRY equivalent column.
`$table->geometryCollection('positions');`	GEOMETRYCOLLECTION equivalent column.
`$table->increments('id');`	Auto-incrementing UNSIGNED INTEGER (primary key) equivalent column.
`$table->integer('votes');`	INTEGER equivalent column.
`$table->ipAddress('visitor');`	IP address equivalent column.
`$table->json('options');`	JSON equivalent column.

`$table->jsonb('options');` | JSONB equivalent column.
`$table->lineString('positions');` | LINESTRING equivalent column.
`$table->longText('description');` | LONGTEXT equivalent column.
`$table->macAddress('device');` | MAC address equivalent column.
`$table->mediumIncrements('id');` | Auto-incrementing UNSIGNED MEDIUMINT (primary key) equivalent column.
`$table->mediumInteger('votes');` | MEDIUMINT equivalent column.
`$table->mediumText('description');` | MEDIUMTEXT equivalent column.
`$table->morphs('taggable');` | Adds `taggable_id` UNSIGNED BIGINT and `taggable_type` VARCHAR equivalent columns.
`$table->uuidMorphs('taggable');` | Adds `taggable_id` CHAR(36) and `taggable_type` VARCHAR(255) UUID equivalent columns.
`$table->multiLineString('positions');` | MULTILINESTRING equivalent column.
`$table->multiPoint('positions');` | MULTIPOINT equivalent column.
`$table->multiPolygon('positions');` | MULTIPOLYGON equivalent column.
`$table->nullableMorphs('taggable');` | Adds nullable versions of `morphs()` columns.
`$table->nullableUuidMorphs('taggable');` | Adds nullable versions of `uuidMorphs()` columns.
`$table->nullableTimestamps();` | Alias of `timestamps()` method.
`$table->point('position');` | POINT equivalent column.
`$table->polygon('positions');` | POLYGON equivalent column.
`$table->rememberToken();` | Adds a nullable `remember_token` VARCHAR(100) equivalent column.
`$table->set('flavors', ['strawberry', 'vanilla']);` | SET equivalent column.
`$table->smallIncrements('id');` | Auto-incrementing UNSIGNED SMALLINT (primary key) equivalent column.
`$table->smallInteger('votes');` | SMALLINT equivalent column.
`$table->softDeletes();` | Adds a nullable `deleted_at` TIMESTAMP equivalent column for soft deletes.
`$table->softDeletesTz();` | Adds a nullable `deleted_at` TIMESTAMP (with timezone) equivalent column for soft deletes.
`$table->string('name', 100);` | VARCHAR equivalent column with a optional length.
`$table->text('description');` | TEXT equivalent column.
`$table->time('sunrise');` | TIME equivalent column.
`$table->timeTz('sunrise');` | TIME (with timezone) equivalent column.
`$table->timestamp('added_on');` | TIMESTAMP equivalent column.
`$table->timestampTz('added_on');` | TIMESTAMP (with timezone) equivalent column.
`$table->timestamps();` | Adds nullable `created_at` and `updated_at` TIMESTAMP equivalent columns.
`$table->timestampsTz();` | Adds nullable `created_at` and `updated_at` TIMESTAMP (with timezone) equivalent columns.
`$table->tinyIncrements('id');` | Auto-incrementing UNSIGNED TINYINT (primary key) equivalent column.
`$table->tinyInteger('votes');` | TINYINT equivalent column.
`$table->unsignedBigInteger('votes');` | UNSIGNED BIGINT equivalent column.

`$table->unsignedDecimal('amount', 8, 2);`	UNSIGNED DECIMAL equivalent column with a precision (total digits) and scale (decimal digits).
`$table->unsignedInteger('votes');`	UNSIGNED INTEGER equivalent column.
`$table->unsignedMediumInteger('votes');`	UNSIGNED MEDIUMINT equivalent column.
`$table->unsignedSmallInteger('votes');`	UNSIGNED SMALLINT equivalent column.
`$table->unsignedTinyInteger('votes');`	UNSIGNED TINYINT equivalent column.
`$table->uuid('id');`	UUID equivalent column.
`$table->year('birth_year');`	YEAR equivalent column.

Column Modifiers

In addition to the column types listed above, there are several column “modifiers” you may use while adding a column to a database table. For example, to make the column “nullable”, you may use the `nullable` method:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

Below is a list of all the available column modifiers. This list does not include the index modifiers:

Modifier	Description
`->after('column')`	Place the column “after” another column (MySQL)
`->autoIncrement()`	Set INTEGER columns as auto-increment (primary key)
`->charset('utf8')`	Specify a character set for the column (MySQL)
`->collation('utf8_unicode_ci')`	Specify a collation for the column (MySQL/PostgreSQL/SQL Server)
`->comment('my comment')`	Add a comment to a column (MySQL/PostgreSQL)
`->default($value)`	Specify a “default” value for the column
`->first()`	Place the column “first” in the table (MySQL)
`->nullable($value = true)`	Allows (by default) NULL values to be inserted into the column
`->storedAs($expression)`	Create a stored generated column (MySQL)
`->unsigned()`	Set INTEGER columns as UNSIGNED (MySQL)
`->useCurrent()`	Set TIMESTAMP columns to use CURRENT_TIMESTAMP as default value
`->virtualAs($expression)`	Create a virtual generated column (MySQL)
`->generatedAs($expression)`	Create an identity column with specified sequence options (PostgreSQL)
`->always()`	Defines the precedence of sequence values over input for an identity column (PostgreSQL)

Modifying Columns

Prerequisites

Before modifying a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file. The Doctrine DBAL library is used to determine the current state of the column and create the SQL queries needed to make the specified adjustments to the column:

```
composer require doctrine/dbal
```

Updating Column Attributes

The `change` method allows you to modify some existing column types to a new type or modify the column's attributes. For example, you may wish to increase the size of a string column. To see the `change` method in action, let's increase the size of the `name` column from 25 to 50:

```
    Schema::table('users', function (Blueprint $table) {
        $table->string('name', 50)->change();
    });
```

We could also modify a column to be nullable:

```
    Schema::table('users', function (Blueprint $table) {
        $table->string('name', 50)->nullable()->change();
    });
```

Only the following column types can be "changed": bigInteger, binary, boolean, date, dateTime, dateTimeTz, decimal, integer, json, longText, mediumText, smallInteger, string, text, time, unsignedBigInteger, unsignedInteger and unsignedSmallInteger.

Renaming Columns

To rename a column, you may use the `renameColumn` method on the Schema builder. Before renaming a column, be sure to add the `doctrine/dbal` dependency to your `composer.json` file:

```
    Schema::table('users', function (Blueprint $table) {
        $table->renameColumn('from', 'to');
    });
```

Renaming any column in a table that also has a column of type `enum` is not currently supported.

Dropping Columns

To drop a column, use the `dropColumn` method on the Schema builder. Before dropping columns from a SQLite database, you will need to add the `doctrine/dbal` dependency to your `composer.json` file and run the `composer update` command in your terminal to install the library:

```
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('votes');
    });
```

You may drop multiple columns from a table by passing an array of column names to the `dropColumn` method:

```
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn(['votes', 'avatar', 'location']);
    });
```

Dropping or modifying multiple columns within a single migration while using a SQLite database is not supported.

Available Command Aliases

Command	Description
`$table->dropMorphs('morphable');`	Drop the `morphable_id` and `morphable_type` columns.
`$table->dropRememberToken();`	Drop the `remember_token` column.
`$table->dropSoftDeletes();`	Drop the `deleted_at` column.
`$table->dropSoftDeletesTz();`	Alias of `dropSoftDeletes()` method.
`$table->dropTimestamps();`	Drop the `created_at` and `updated_at` columns.
`$table->dropTimestampsTz();`	Alias of `dropTimestamps()` method.

Indexes

Creating Indexes

The schema builder supports several types of indexes. First, let's look at an example that specifies a column's values should be unique. To create the index, we can chain the `unique` method onto the column definition:

```
    $table->string('email')->unique();
```

Alternatively, you may create the index after defining the column. For example:

```
    $table->unique('email');
```

You may even pass an array of columns to an index method to create a compound (or composite) index:

```
    $table->index(['account_id', 'created_at']);
```

Laravel will automatically generate a reasonable index name, but you may pass a second argument to the method to specify the name yourself:

```
    $table->unique('email', 'unique_email');
```

Available Index Types

Each index method accepts an optional second argument to specify the name of the index. If omitted, the name will be derived from the names of the table and column(s).

Command	Description
`$table->primary('id');`	Adds a primary key.
`$table->primary(['id', 'parent_id']);`	Adds composite keys.
`$table->unique('email');`	Adds a unique index.
`$table->index('state');`	Adds a plain index.
`$table->spatialIndex('location');`	Adds a spatial index. (except SQLite)

Index Lengths & MySQL / MariaDB

Laravel uses the `utf8mb4` character set by default, which includes support for storing "emojis" in the database. If you are running a version of MySQL older than the 5.7.7 release or MariaDB older than the 10.2.2 release, you may need to manually configure the default string length generated by migrations in order for MySQL to create indexes for them. You may configure this by calling the `Schema::defaultStringLength` method within your `AppServiceProvider`:

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
    Schema::defaultStringLength(191);
}
```

Alternatively, you may enable the `innodb_large_prefix` option for your database. Refer to your database's documentation for instructions on how to properly enable this option.

Foreign Key Constraints

Laravel also provides support for creating foreign key constraints, which are used to force referential integrity at the database level. For example, let's define a `user_id` column on the `posts` table that references the `id` column on a `users` table:

```
Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

You may also specify the desired action for the "on delete" and "on update" properties of the constraint:

```
$table->foreign('user_id')
      ->references('id')->on('users')
      ->onDelete('cascade');
```

To drop a foreign key, you may use the `dropForeign` method. Foreign key constraints use the same naming convention as indexes. So, we will concatenate the table name and the columns in the constraint then suffix the name with "_foreign":

```
$table->dropForeign('posts_user_id_foreign');
```

Or, you may pass an array value which will automatically use the conventional constraint name when dropping:

```
    $table->dropForeign(['user_id']);
```

You may enable or disable foreign key constraints within your migrations by using the following methods:

```
    Schema::enableForeignKeyConstraints();

    Schema::disableForeignKeyConstraints();
```

SQLite disables foreign key constraints by default. When using SQLite, make sure to enable foreign key support in your database configuration before attempting to create them in your migrations.

Database: Seeding

Introduction

Laravel includes a simple method of seeding your database with test data using seed classes. All seed classes are stored in the `database/seeds` directory. Seed classes may have any name you wish, but probably should follow some sensible convention, such as `UsersTableSeeder`, etc. By default, a `DatabaseSeeder` class is defined for you. From this class, you may use the `call` method to run other seed classes, allowing you to control the seeding order.

Writing Seeders

To generate a seeder, execute the `make:seeder` Artisan command. All seeders generated by the framework will be placed in the `database/seeds` directory:

```
php artisan make:seeder UsersTableSeeder
```

A seeder class only contains one method by default: `run`. This method is called when the `db:seed` Artisan command is executed. Within the `run` method, you may insert data into your database however you wish. You may use the query builder to manually insert data or you may use Eloquent model factories.

> Mass assignment protection is automatically disabled during database seeding.

As an example, let's modify the default `DatabaseSeeder` class and add a database insert statement to the `run` method:

```
<?php

use Illuminate\Support\Str;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
```

```
         *
         * @return void
         */
        public function run()
        {
            DB::table('users')->insert([
                'name' => Str::random(10),
                'email' => Str::random(10).'@gmail.com',
                'password' => bcrypt('password'),
            ]);
        }
    }
```

> You may type-hint any dependencies you need within the `run` method's signature. They will automatically be resolved via the Laravel service container.

Using Model Factories

Of course, manually specifying the attributes for each model seed is cumbersome. Instead, you can use model factories to conveniently generate large amounts of database records. First, review the model factory documentation to learn how to define your factories. Once you have defined your factories, you may use the `factory` helper function to insert records into your database.

For example, let's create 50 users and attach a relationship to each user:

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    factory(App\User::class, 50)->create()->each(function ($user) {
        $user->posts()->save(factory(App\Post::class)->make());
    });
}
```

Calling Additional Seeders

Within the `DatabaseSeeder` class, you may use the `call` method to execute additional seed classes. Using the `call` method allows you to break up your database seeding into multiple files so that no single seeder class becomes overwhelmingly large. Pass the name of the seeder class you wish to run:

```php
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $this->call([
            UsersTableSeeder::class,
            PostsTableSeeder::class,
            CommentsTableSeeder::class,
        ]);
    }
```

Running Seeders

Once you have written your seeder, you may need to regenerate Composer's autoloader using the `dump-autoload` command:

```
composer dump-autoload
```

Now you may use the `db:seed` Artisan command to seed your database. By default, the `db:seed` command runs the `DatabaseSeeder` class, which may be used to call other seed classes. However, you may use the `--class` option to specify a specific seeder class to run individually:

```
php artisan db:seed
```

```
php artisan db:seed --class=UsersTableSeeder
```

You may also seed your database using the `migrate:refresh` command, which will also rollback and re-run all of your migrations. This command is useful for completely re-building your database:

```
php artisan migrate:refresh --seed
```

Forcing Seeders To Run In Production

Some seeding operations may cause you to alter or lose data. In order to protect you from running seeding commands against your production database, you will be prompted for confirmation before the seeders are executed. To force the seeders to run without a prompt, use the `--force` flag:

```
php artisan db:seed --force
```

Eloquent: Getting Started

Introduction

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.

Before getting started, be sure to configure a database connection in `config/database.php`. For more information on configuring your database, check out the documentation.

Defining Models

To get started, let's create an Eloquent model. Models typically live in the `app` directory, but you are free to place them anywhere that can be auto-loaded according to your `composer.json` file. All Eloquent models extend `Illuminate\Database\Eloquent\Model` class.

The easiest way to create a model instance is using the `make:model` Artisan command:

```
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
```

```
php artisan make:model Flight -m
```

Eloquent Model Conventions

Now, let's look at an example `Flight` model, which we will use to retrieve and store information from our `flights` database table:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

Table Names

Note that we did not tell Eloquent which table to use for 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. You may specify a custom table by defining a `table` property on your model:

```php
<?php

namespace App;

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 table has a primary key column named `id`. You may define a protected `$primaryKey` property to override this convention:

```php
<?php

namespace App;

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 by default the primary key will automatically be cast to an `int`. If you wish to use a non-incrementing or a non-numeric primary key you must set the public `$incrementing` property on your model to `false`:

```php
<?php

class Flight extends Model
{
    /**
     * Indicates if the IDs are auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

If your primary key is not an integer, you should set the protected `$keyType` property on your model to `string`:

```php
<?php

class Flight extends Model
{
    /**
     * The "type" of the auto-incrementing ID.
     *
     * @var string
     */
    protected $keyType = 'string';
}
```

Timestamps

By default, Eloquent expects `created_at` and `updated_at` columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the `$timestamps` property on your model to `false`:

```php
<?php

namespace App;

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 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;

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 set the `CREATED_AT` and `UPDATED_AT` constants in your model:

```php
<?php

class Flight extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'last_update';
}
```

Database Connection

By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the `$connection` property:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
```

```
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

Default 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

namespace App;

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. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```php
<?php

$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

Adding Additional Constraints

The Eloquent `all` method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the `get` method to retrieve the results:

```php
$flights = App\Flight::where('active', 1)
               ->orderBy('name', 'desc')
               ->take(10)
               ->get();
```

Since Eloquent models are query builders, you should review all of the methods available on the query builder. You may use any of these methods in your Eloquent queries.

Refreshing Models

You can refresh models 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:

```php
$flight = App\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:

```php
$flight = App\Flight::where('number', 'FR 900')->first();

$flight->number = 'FR 456';

$flight->refresh();

$flight->number; // "FR 900"
```

Collections

For Eloquent methods like `all` and `get` which retrieve multiple results, an instance of `Illuminate\Database\Eloquent\Co` will be returned. The `Collection` class provides a variety of helpful methods for working with your Eloquent results:

```
$flights = $flights->reject(function ($flight) {
    return $flight->cancelled;
});
```

You may also loop over the collection like an array:

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Retrieving Single Models / Aggregates

In addition to retrieving all of the records for a given table, you may also retrieve single records using `find` or `first`. Instead of returning a collection of models, these methods return a single model instance:

```
// Retrieve a model by its primary key...
$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();
```

You may also call the `find` method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

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, a `Illuminate\Database\Eloquent\ModelNotFoundException` will be thrown:

```
$model = App\Flight::findOrFail(1);

$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

If the exception is not caught, a `404` HTTP response is automatically sent back to the user. It is not necessary to write explicit checks to return `404` responses when using these methods:

```
Route::get('/api/flights/{id}', function ($id) {
    return App\Flight::findOrFail($id);
});
```

Retrieving Aggregates

You may also use the `count`, `sum`, `max`, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

Inserting & Updating Models

Inserts

To create a new record in the database, create a new model instance, set attributes on the model, then call the `save` method:

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
    /**
     * Create a new flight instance.
     *
     * @param  Request  $request
```

```
         * @return 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` parameter from the incoming HTTP request to the `name` attribute of the `App\Flight` model instance. When we call the `save` method, a record will be inserted into the database. The `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.

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, set any attributes you wish to update, and then call the `save` method. Again, the `updated_at` timestamp will automatically be updated, so there is no need to manually set its value:

```
$flight = App\Flight::find(1);

$flight->name = 'New Flight Name';

$flight->save();
```

Mass Updates

Updates can also be performed against any number of 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:

```
App\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.

> 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.

Mass Assignment

You may also use the `create` method to save a new model in a single line. The inserted model instance will be returned to you from the method. However, before doing so, you will need to specify either a `fillable` or `guarded` attribute on the model, as all Eloquent models protect against mass-assignment by default.

A mass-assignment vulnerability occurs when a user passes an unexpected HTTP parameter through a request, and that parameter changes a column in your database you did not expect. For example, a malicious user might send an `is_admin` parameter through an HTTP request, which is then passed into 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;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Once we have made the attributes mass assignable, we can use the `create` method to insert a new record in the database. The `create` method returns the saved model instance:

```php
$flight = App\Flight::create(['name' => 'Flight 10']);
```

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' => 'Flight 22']);
```

Guarding Attributes

While `$fillable` serves as a "white list" of attributes that should be mass assignable, you may also choose to use `$guarded`. The `$guarded` property should contain an array of attributes that you do not want to be mass assignable. All other attributes not in the array will be mass assignable. So, `$guarded` functions like a "black list". Importantly, you should use either `$fillable` or `$guarded` - not both. In the example below, all attributes **except for `price`** will be mass assignable:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

If you would like to make all attributes mass assignable, you may define the `$guarded` property as an empty array:

```php
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

Other Creation Methods

`firstOrCreate`/ `firstOrNew`

There are two other methods you may use to create models by mass assigning attributes: `firstOrCreate` and `firstOrNew`. 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 from the first parameter, along with those in the optional second parameter.

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 call `save` manually to persist it:

```
    // Retrieve flight by name, or create it if it doesn't exist...
    $flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);

    // Retrieve flight by name, or create it with the name, delayed, and arrival_tim\
e attributes...
    $flight = App\Flight::firstOrCreate(
        ['name' => 'Flight 10'],
        ['delayed' => 1, 'arrival_time' => '11:30']
    );

    // Retrieve by name, or instantiate...
    $flight = App\Flight::firstOrNew(['name' => 'Flight 10']);

    // Retrieve by name, or instantiate with the name, delayed, and arrival_time att\
ributes...
    $flight = App\Flight::firstOrNew(
        ['name' => 'Flight 10'],
        ['delayed' => 1, 'arrival_time' => '11:30']
    );
```

`updateOrCreate`

You may also come across situations where you want to update an existing model or create a new model if none exists. Laravel provides an `updateOrCreate` method to do this in one step. Like the `firstOrCreate` method, `updateOrCreate` persists the model, so there's no need to call `save()`:

```
    // If there's a flight from Oakland to San Diego, set the price to $99.
    // If no matching model exists, create one.
    $flight = App\Flight::updateOrCreate(
        ['departure' => 'Oakland', 'destination' => 'San Diego'],
        ['price' => 99, 'discounted' => 1]
    );
```

Deleting Models

To delete a model, call the `delete` method on a model instance:

```php
$flight = App\Flight::find(1);

$flight->delete();
```

Deleting An Existing Model By 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 retrieving it by calling the `destroy` method. In addition to a single primary key as its argument, the `destroy` method will accept multiple primary keys, an array of primary keys, or a collection of primary keys:

```php
App\Flight::destroy(1);

App\Flight::destroy(1, 2, 3);

App\Flight::destroy([1, 2, 3]);

App\Flight::destroy(collect([1, 2, 3]));
```

Deleting Models By Query

You can also run a delete statement on a set of models. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not fire any model events for the models that are deleted:

```php
$deletedRows = App\Flight::where('active', 0)->delete();
```

> When executing a mass delete statement via Eloquent, the `deleting` and `deleted` model events will not be fired 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 and inserted into the database. If a model has a non-null `deleted_at` value, the model has been soft deleted. To enable soft deletes for a model, use the `Illuminate\Database\Eloquent\SoftDeletes` trait on the model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;

class Flight extends Model
{
    use SoftDeletes;
}
```

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[80] contains a helper method to create this column:

```php
Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});
```

Now, when you call the `delete` method on the model, the `deleted_at` column will be set to the current date and time. And, 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, use the `trashed` method:

```php
if ($flight->trashed()) {
    //
}
```

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 appear in a result set using the `withTrashed` method on the query:

[80]/docs/\protect\char"007B\relax\protect\char"007B\relaxversion\protect\char"007D\relax\protect\char"007D\relax/migrations

```
$flights = App\Flight::withTrashed()
                ->where('account_id', 1)
                ->get();
```

The `withTrashed` method may also be used on a relationship query:

```
$flight->history()->withTrashed()->get();
```

Retrieving Only Soft Deleted Models

The `onlyTrashed` method will retrieve **only** soft deleted models:

```
$flights = App\Flight::onlyTrashed()
                ->where('airline_id', 1)
                ->get();
```

Restoring Soft Deleted Models

Sometimes you may wish to "un-delete" a soft deleted model. To restore a soft deleted model into an active state, use the `restore` method on a model instance:

```
$flight->restore();
```

You may also use the `restore` method in a query to quickly restore multiple models. Again, like other "mass" operations, this will not fire any model events for the models that are restored:

```
App\Flight::withTrashed()
        ->where('airline_id', 1)
        ->restore();
```

Like the `withTrashed` method, the `restore` method may also be used on relationships:

```
$flight->history()->restore();
```

Permanently Deleting Models

Sometimes you may need to truly remove a model from your database. To permanently remove a soft deleted model from the database, use the `forceDelete` method:

```php
// Force deleting a single model instance...
$flight->forceDelete();

// Force deleting all related models...
$flight->history()->forceDelete();
```

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 pull "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. Define a class that implements the `Illuminate\Database\Eloquent\Scope` interface. This interface requires you to implement one method: `apply`. The `apply` method may add `where` constraints to the query as needed:

```php
<?php

namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope 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('age', '>', 200);
    }
}
```

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 a given model's `boot` method and use the `addGlobalScope` method:

```php
<?php

namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope);
    }
}
```

After adding the scope, a query to `User::all()` will produce the following SQL:

```
select * from `users` where `age` > 200
```

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:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class User extends Model
{
    /**
     * The "booting" method of the model.
     *
     * @return void
     */
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope('age', function (Builder $builder) {
            $builder->where('age', '>', 200);
        });
    }
}
```

Removing Global Scopes

If you would like to remove a global scope for a given query, you may use the `withoutGlobalScope` method. The method accepts the class name of the global scope as its only argument:

```php
User::withoutGlobalScope(AgeScope::class)->get();
```

Or, if you defined the global scope using a Closure:

```php
User::withoutGlobalScope('age')->get();
```

If you would like to remove several or even all of the 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 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 a query builder instance:

```
<?php

namespace App;

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  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopeActive($query)
    {
            return $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, for example:

```
    $users = App\User::popular()->active()->orderBy('created_at')->get();
```

Combining multiple Eloquent model scopes via an `or` query operator may require the use of Closure callbacks:

```
    $users = App\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 these scopes together without the use of Closures:

```
    $users = App\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. Scope parameters should be defined after the `$query` parameter:

```
    <?php

    namespace App;

    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);
        }
    }
```

Now, you may pass the parameters when calling the scope:

```
$users = App\User::ofType('admin')->get();
```

Comparing Models

Sometimes you may need to determine if two models are the "same". The `is` method may be used to quickly verify two models have same primary key, table, and database connection:

```
if ($post->is($anotherPost)) {
    //
}
```

Eloquent: Relationships

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 several different types of relationships:

- One To One
- One To Many
- Many To Many

Defining Relationships

Eloquent relationships are defined as methods on your Eloquent model classes. Since, like Eloquent models themselves, relationships also serve as powerful query builders, defining relationships as methods provides powerful method chaining and querying capabilities. For example, we may chain additional 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.

One To One

A one-to-one relationship is a very basic relation. For example, a `User` model might be associated with one `Phone`. To define this relationship, we place a `phone` method on the `User` model. The `phone` method should call the `hasOne` method and return its result:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

The first argument passed to the `hasOne` method is the name of the related model. 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:

```php
$phone = User::find(1)->phone;
```

Eloquent determines the foreign key of the relationship based on the 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:

```php
return $this->hasOne('App\Phone', 'foreign_key');
```

Additionally, Eloquent assumes that the foreign key should have a value matching the `id` (or the custom `$primaryKey`) 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 value other than `id`, you may pass a third argument to the `hasOne` method specifying your custom key:

```php
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Defining The Inverse Of The Relationship

So, we can access the `Phone` model from our `User`. Now, 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;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

In the example above, Eloquent will try to match the `user_id` from the `Phone` model to an `id` on the `User` model. Eloquent determines the default foreign key name by examining the name of the relationship method and suffixing the method name with `_id`. 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('App\User', 'foreign_key');
}
```

If your parent model does not use `id` as its primary key, or you wish to join the child model to a different column, you may pass a third argument to the `belongsTo` method specifying your parent table's custom key:

```php
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User', 'foreign_key', 'other_key');
    }
```

One To Many

A one-to-many relationship is used to define relationships where a single model owns any amount of other 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 placing a function on your Eloquent model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Remember, Eloquent will automatically determine the proper foreign key column on the `Comment` model. By convention, Eloquent will take the "snake case" name of the owning model and suffix it with `_id`. So, for this example, Eloquent will assume the foreign key on the `Comment` model is `post_id`.

Once the relationship has been defined, we can access the collection of comments by accessing the `comments` property. Remember, since Eloquent provides "dynamic properties", we can access relationship methods as if they were defined as properties on the model:

```
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Since all relationships also serve as query builders, you can add further constraints to which comments are retrieved by calling the `comments` method and continuing to chain conditions onto the query:

```
$comment = App\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('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key', 'local_key');
```

One To Many (Inverse)

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 function on the child model which calls the `belongsTo` method:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Once the relationship has been defined, we can retrieve the `Post` model for a `Comment` by accessing the `post` "dynamic property":

```
    $comment = App\Comment::find(1);

    echo $comment->post->title;
```

In the example above, Eloquent will try to match the `post_id` from the `Comment` model to an `id` on the `Post` 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 primary key column. However, if the foreign key on the `Comment` model is not `post_id`, you may pass a custom key name as the second argument to the `belongsTo` method:

```
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post', 'foreign_key');
    }
```

If your parent model does not use `id` as its primary key, or you wish to join the child model to 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('App\Post', 'foreign_key', 'other_key');
    }
```

Many To Many

Many-to-many relations are slightly more complicated than `hasOne` and `hasMany` relationships. An example of such a relationship is a user with many roles, where the roles are also shared by other users. For example, many users may have the role of "Admin". 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 the `user_id` and `role_id` columns.

Many-to-many relationships are defined by writing a method that returns the result of the `belongsToMany` method. For example, let's define the `roles` method on our `User` model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

Once the relationship is defined, you may access the user's roles using the `roles` dynamic property:

```php
$user = App\User::find(1);

foreach ($user->roles as $role) {
    //
}
```

Like all other relationship types, you may call the `roles` method to continue chaining query constraints onto the relationship:

```php
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

As mentioned previously, to determine the table name of the relationship's joining 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:

```php
return $this->belongsToMany('App\Role', 'role_user');
```

In addition to customizing the name of the joining 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('App\Role', 'role_user', 'user_id', 'role_id');
```

Defining The Inverse Of The Relationship

To define the inverse of a many-to-many relationship, you place another call to `belongsToMany` on your related model. To continue our user roles example, let's define the `users` method on the `Role` model:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User');
    }
}
```

As you can see, the relationship is defined exactly the same as its `User` counterpart, with the exception of referencing the `App\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` object has many `Role` objects that it is related to. After accessing this relationship, we may access the intermediate table using the `pivot` attribute on the models:

```php
$user = App\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, and may be used like any other Eloquent model.

By default, only the model keys will be present on the `pivot` object. If your pivot table contains extra attributes, you must specify them when defining the relationship:

```php
return $this->belongsToMany('App\Role')->withPivot('column1', 'column2');
```

If you want your pivot table to have automatically maintained `created_at` and `updated_at` timestamps, use the `withTimestamps` method on the relationship definition:

```php
return $this->belongsToMany('App\Role')->withTimestamps();
```

Customizing The `pivot` Attribute Name

As noted earlier, attributes from the intermediate table may be accessed on models using 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 probably have a many-to-many relationship between users and podcasts. If this is the case, you may wish to rename your intermediate table accessor to `subscription` instead of `pivot`. This can be done using the `as` method when defining the relationship:

```php
return $this->belongsToMany('App\Podcast')
                ->as('subscription')
                ->withTimestamps();
```

Once this is done, you may access the intermediate table data using the customized name:

```php
$users = User::with('podcasts')->get();

foreach ($users->flatMap->podcasts as $podcast) {
    echo $podcast->subscription->created_at;
}
```

Filtering Relationships Via Intermediate Table Columns

You can also filter the results returned by `belongsToMany` using the `wherePivot` and `wherePivotIn` methods when defining the relationship:

```
    return $this->belongsToMany('App\Role')->wherePivot('approved', 1);

    return $this->belongsToMany('App\Role')->wherePivotIn('priority', [1, 2]);
```

Defining Custom Intermediate Table Models

If you would like to define a custom model to represent the intermediate table of your relationship, you may call the `using` method when defining the relationship. 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` which uses a custom `RoleUser` pivot model:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User')->using('App\RoleUser');
    }
}
```

When defining the `RoleUser` model, we will extend the `Pivot` class:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Relations\Pivot;

class RoleUser extends Pivot
{
    //
}
```

You can combine `using` and `withPivot` in order to retrieve columns from the intermediate table. For example, you may retrieve the `created_by` and `updated_by` columns from the `RoleUser` pivot table by passing the column names to the `withPivot` method:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany('App\User')
                        ->using('App\RoleUser')
                        ->withPivot([
                            'created_by',
                            'updated_by',
                        ]);
    }
}
```

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`.

```php
/**
 * Indicates if the IDs are auto-incrementing.
 *
 * @var bool
 */
public $incrementing = true;
```

Querying Relations

Since all types of Eloquent relationships are defined via methods, you may call those methods to obtain an instance of the relationship without actually executing the relationship queries. 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 against your database.

For example, imagine a blog system in which a `User` model has many associated `Post` models:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * Get all of the posts for the user.
     */
    public function posts()
    {
        return $this->hasMany('App\Post');
    }
}
```

You may query the `posts` relationship and add additional constraints to the relationship like so:

```php
$user = App\User::find(1);

$user->posts()->where('active', 1)->get();
```

You are able to use any of the query builder 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();

// select * from posts
// where user_id = ? and active = 1 or votes >= 100
```

In most situations, you likely intend to use constraint groups to logically 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();

// 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:

```
$user = App\User::find(1);

foreach ($user->posts as $post) {
    //
}
```

Dynamic properties are "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 accessing the records for a model, 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:

```
// Retrieve all posts that have at least one comment...
$posts = App\Post::has('comments')->get();
```

You may also specify an operator and count to further customize the query:

```
// Retrieve all posts that have three or more comments...
$posts = App\Post::has('comments', '>=', 3)->get();
```

Nested `has` statements may also be constructed using "dot" notation. For example, you may retrieve all posts that have at least one comment and vote:

```
// Retrieve posts that have at least one comment with votes...
$posts = App\Post::has('comments.votes')->get();
```

If you need even more power, you may use the `whereHas` and `orWhereHas` methods to put "where" conditions on your `has` queries. These methods allow you to add customized constraints to a relationship constraint, such as checking the content of a comment:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like foo%...
$posts = App\Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'foo%');
})->get();

// Retrieve posts with at least ten comments containing words like foo%...
$posts = App\Post::whereHas('comments', function (Builder $query) {
    $query->where('content', 'like', 'foo%');
}, '>=', 10)->get();
```

Querying Relationship Absence

When accessing the records for a model, 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:

```
    $posts = App\Post::doesntHave('comments')->get();
```

If you need even more power, you may use the `whereDoesntHave` and `orWhereDoesntHave` methods to put "where" conditions on your `doesntHave` queries. These methods allows you to add customized constraints to a relationship constraint, such as checking the content of a comment:

```
    use Illuminate\Database\Eloquent\Builder;

    $posts = App\Post::whereDoesntHave('comments', function (Builder $query) {
        $query->where('content', 'like', 'foo%');
    })->get();
```

You may use "dot" notation to execute a query against a nested relationship. For example, the following query will retrieve all posts with comments from authors that are not banned:

```
    use Illuminate\Database\Eloquent\Builder;

    $posts = App\Post::whereDoesntHave('comments.author', function (Builder $query) {
        $query->where('banned', 1);
    })->get();
```

Querying Polymorphic Relationships

To query the existence of `MorphTo` relationships, you may use the `whereHasMorph` method and its corresponding methods:

```
    use Illuminate\Database\Eloquent\Builder;

    // Retrieve comments associated to posts or videos with a title like foo%...
    $comments = App\Comment::whereHasMorph(
        'commentable',
        ['App\Post', 'App\Video'],
        function (Builder $query) {
            $query->where('title', 'like', 'foo%');
        }
    )->get();

    // Retrieve comments associated to posts with a title not like foo%...
    $comments = App\Comment::whereDoesntHaveMorph(
        'commentable',
        'App\Post',
        function (Builder $query) {
            $query->where('title', 'like', 'foo%');
        }
    )->get();
```

You may use the `$type` parameter to add different constraints depending on the related model:

```
    use Illuminate\Database\Eloquent\Builder;

    $comments = App\Comment::whereHasMorph(
        'commentable',
        ['App\Post', 'App\Video'],
        function (Builder $query, $type) {
            $query->where('title', 'like', 'foo%');

            if ($type === 'App\Post') {
                $query->orWhere('content', 'like', 'foo%');
            }
        }
    )->get();
```

Instead of passing an array of possible polymorphic models, you may provide `*` as a wildcard and let Laravel retrieve all 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 = App\Comment::whereHasMorph('commentable', '*', function (Builder $qu\
ery) {
        $query->where('title', 'like', 'foo%');
    })->get();
```

Counting Related Models

If you want to count the number of results from a relationship without actually loading them you may use the `withCount` method, which will place a `{relation}_count` column on your resulting models. For example:

```php
$posts = App\Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

You may add the "counts" for multiple relations as well as add constraints to the queries:

```php
use Illuminate\Database\Eloquent\Builder;

$posts = App\Post::withCount(['votes', 'comments' => function (Builder $query) {
    $query->where('content', 'like', 'foo%');
}])->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:

```php
use Illuminate\Database\Eloquent\Builder;

$posts = App\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;
```

If you're combining `withCount` with a `select` statement, ensure that you call `withCount` after the `select` method:

```php
$posts = App\Post::select(['title', 'body'])->withCount('comments')->get();

echo $posts[0]->title;
echo $posts[0]->body;
echo $posts[0]->comments_count;
```

Eager Loading

When accessing Eloquent relationships as properties, the relationship data is “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 is related to `Author`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Book extends Model
{
    /**
     * Get the author that wrote the book.
     */
    public function author()
    {
        return $this->belongsTo('App\Author');
    }
}
```

Now, let’s retrieve all books and their authors:

```php
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

This loop will execute 1 query to retrieve all of the books on the table, then another query for each book to retrieve the author. So, if we have 25 books, this loop would run 26 queries: 1 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 2 queries. When querying, you may specify which relationships should be eager loaded using the `with` method:

```
    $books = App\Book::with('author')->get();

    foreach ($books as $book) {
        echo $book->author->name;
    }
```

For this operation, only two queries will be executed:

```
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 in a single operation. To do so, just pass additional arguments to the `with` method:

```
    $books = App\Book::with(['author', 'publisher'])->get();
```

Nested Eager Loading

To eager load nested 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 in one Eloquent statement:

```
    $books = App\Book::with('author.contacts')->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 = App\Book::with('author:id,name')->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;

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('App\Author');
    }
}
```

If you would like to remove an item from the `$with` property for a single query, you may use the `without` method:

```php
$books = App\Book::without('author')->get();
```

Constraining Eager Loads

Sometimes you may wish to eager load a relationship, but also specify additional query conditions for the eager loading query. Here's an example:

```php
$users = App\User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%first%');
}])->get();
```

In this example, Eloquent will only eager load posts where the post's `title` column contains the word `first`. You may call other query builder methods to further customize the eager loading operation:

```
$users = App\User::with(['posts' => function ($query) {
    $query->orderBy('created_at', 'desc');
}])->get();
```

Note: The `limit` and `take` query builder methods may not be used when constraining eager loads.

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:

```
$books = App\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:

```
$books->load(['author' => function ($query) {
    $query->orderBy('published_date', 'asc');
}]);
```

To load a relationship only when it has not already been loaded, use the `loadMissing` method:

```
public function format(Book $book)
{
    $book->loadMissing('author');

    return [
        'name' => $book->name,
        'author' => $book->author->name,
    ];
}
```

Inserting & Updating Related Models

The Save Method

Eloquent provides convenient methods for adding new models to relationships. For example, perhaps you need to insert a new `Comment` for a `Post` model. Instead of manually setting the `post_id` attribute on the `Comment`, you may insert the `Comment` directly from the relationship's `save` method:

```
$comment = new App\Comment(['message' => 'A new comment.']);

$post = App\Post::find(1);

$post->comments()->save($comment);
```

Notice 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 = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new comment.']),
    new App\Comment(['message' => 'Another comment.']),
]);
```

Recursively Saving Models & Relationships

If you would like to `save` your model and all of its associated relationships, you may use the `push` method:

```
$post = App\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. Again, the difference between `save` and `create` is that `save` accepts a full Eloquent model instance while `create` accepts a plain PHP `array`:

```
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

Tip: Before using the `create` method, be sure to review the documentation on attribute mass assignment.

You may use the `createMany` method to create multiple related models:

```
$post = App\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.

Belongs To Relationships

When updating a `belongsTo` relationship, you may use the `associate` method. This method will set the foreign key on the child model:

```
    $account = App\Account::find(10);

    $user->account()->associate($account);

    $user->save();
```

When removing a `belongsTo` relationship, 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 a few additional helper methods to make working with related models more convenient. For example, let's imagine a user can have many roles and a role can have many users. To attach a role to a user by inserting a record in the intermediate table that joins the models, use the `attach` method:

```
    $user = App\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 = App\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 do not want to detach existing IDs, 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 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]);
```

Saving Additional Data On A Pivot Table

When working with a many-to-many relationship, the `save` method accepts an array of additional intermediate table attributes as its second argument:

```
App\User::find(1)->roles()->save($role, ['expires' => $expires]);
```

Updating A Record On A Pivot Table

If you need to update an existing row in your pivot table, you may use `updateExistingPivot` method. This method accepts the pivot record foreign key and an array of attributes to update:

```
$user = App\User::find(1);

$user->roles()->updateExistingPivot($roleId, $attributes);
```

Touching Parent Timestamps

When a model `belongsTo` or `belongsToMany` 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`. Eloquent makes it easy. Just add a `touches` property containing the names of the relationships to the child model:

```
<?php

namespace App;

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('App\Post');
    }
}
```

Now, when you update a `Comment`, the owning `Post` will have its `updated_at` column updated as well, making it more convenient to know when to invalidate a cache of the `Post` model:

```php
$comment = App\Comment::find(1);

$comment->text = 'Edit to this comment!';

$comment->save();
```