Chapter 3. Routing and Controllers

The essential function of any web application framework is to take requests from a user and deliver responses, usually via HTTP(S). This means defining an application's routes is the first and most important project to tackle when learning a web framework; without routes, you have little to no ability to interact with the end user.

In this chapter we will examine routes in Laravel; you'll see how to define them, how to point them to the code they should execute, and how to use Laravel's routing tools to handle a diverse array of routing needs.

A Quick Introduction to MVC, the HTTP Verbs, and REST

Most of what we'll talk about in this chapter references how Model–View–Controller (MVC) applications are structured, and many of the examples we'll be looking at use REST-ish route names and verbs, so let's take a quick look at both.

What Is MVC?

In MVC, you have three primary concepts:

model

> Represents an individual database table (or a record from that table)—think "Company" or "Dog."

view

Represents the template that outputs your data to the end user—think "the login page template with this given set of HTML and CSS and JavaScript."

controller

Like a traffic cop, takes HTTP requests from the browser, gets the right data out of the database and other storage mechanisms, validates user input, and eventually sends a response back to the user.

In Figure 3-1, you can see that the end user will first interact with the controller by sending an HTTP request using their browser. The controller, in response to that request, may write data to and/or pull data from the model (database). The controller will then likely send data to a view, and then the view will be returned to the end user to display in their browser.

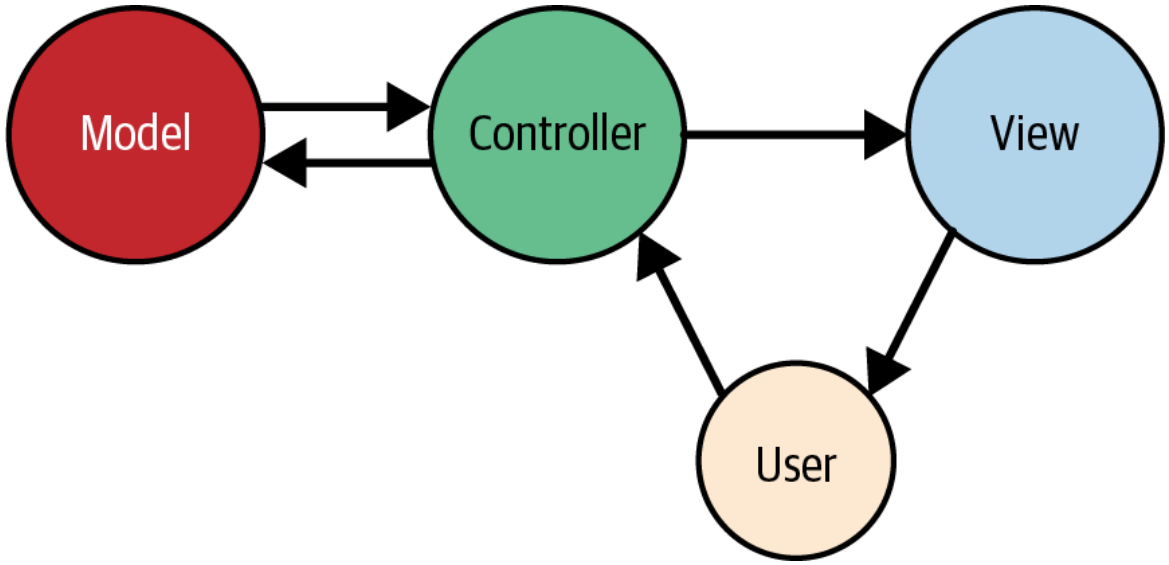

Figure 3-1. A basic illustration of MVC

We'll cover some use cases for Laravel that don't fit this relatively simplistic way of looking at application architecture, so don't get hung up on MVC, but this will at least get you ready to approach the rest of this chapter as we talk about views and controllers.

The HTTP Verbs

the most common HTTP verbs are `GET` and `POST`, followed by `PUT` and `DELETE`. There are also `HEAD`, `OPTIONS`, and `PATCH`, and two others that are pretty much never used in normal web development, `TRACE` and `CONNECT`.

Here's a quick rundown:

`GET`

: Request a resource (or a list of resources).

`HEAD`

: Ask for a headers-only version of the `GET` response.

`POST`

: Create a resource.

`PUT`

: Overwrite a resource.

`PATCH`

: Modify a resource.

`DELETE`

: Delete a resource.

`OPTIONS`

: Ask the server which verbs are allowed at this URL.

Table 3-1 shows the actions available on a resource controller (more on these in "Resource Controllers"). Each action expects you to call a specific URL pattern using a specific verb, so you can get a sense of what each verb is used for.

Table 3-1. The methods of Laravel's resource controllers

Verb	URL	Controller method	Name	Desc
GET	tasks	index()	tasks.index	Show
GET	tasks/create	create()	tasks.create	Show creat form
POST	tasks	store()	tasks.store	Acce subm from creat form
GET	tasks/{task}	show()	tasks.show	Show
GET	tasks/{task}/edit	edit()	tasks.edit	Edit
PUT/PATCH	tasks/{task}	update()	tasks.update	Acce subm from task
DELETE	tasks/{task}	destroy()	tasks.destroy	Delet task

What Is REST?

We'll cover REST in greater detail in "The Basics of REST-Like JSON APIs", but as a brief introduction, it's an architectural style for building APIs. When we talk about REST in this book, we'll mainly be referencing a few characteristics, such as:

- Being structured around one primary resource at a time (e.g., `tasks`)
- Consisting of interactions with predictable URL structures using HTTP verbs (as seen in Table 3-1)
- Returning JSON and often being requested with JSON

There's more to it, but usually "RESTful" as it'll be used in this book will mean "patterned after these URL-based structures so we can make predictable calls like `GET /tasks/14/edit` for the edit page." This is relevant (even when not building APIs) because Laravel's routing structures are based around a REST-like structure, as you can see in Table 3-1.

REST-based APIs follow mainly this same structure, except they don't have a *create* route or an *edit* route, since APIs just represent actions, not pages that prep for the actions.

Route Definitions

In a Laravel application, you will define your web routes in *routes/web.php* and your API routes in *routes/api.php*. *Web routes* are those that will be visited by your end users; *API routes* are those for your API, if you have one. For now, we'll primarily focus on the routes in *routes/web.php*.

The simplest way to define a route is to match a path (e.g., `/`) with a closure, as seen in Example 3-1.

Example 3-1. Basic route definition

```
// routes/web.php
Route::get('/', function () {
    return 'Hello, World!';
});
```

WHAT'S A CLOSURE?

Closures are PHP's version of anonymous functions. A closure is a function that you can pass around as an object, assign to a variable, pass as a parameter to other functions and methods, or even serialize.

You've now defined that if anyone visits `/` (the root of your domain), Laravel's router should run the closure defined there and return the result. Note that we `return` our content and don't `echo` or `print` it.

A QUICK INTRODUCTION TO MIDDLEWARE

You might be wondering, "Why am I returning 'Hello, World!' instead of echoing it?"

There are quite a few answers, but the simplest is that there are a lot of wrappers around Laravel's request and response cycle, including something called *middleware*. When your route closure or controller method is done, it's not time to send the output to the browser yet; returning the content allows it to continue flowing through the response stack and the middleware before it is returned to the user.

Many simple websites could be defined entirely within the web routes file. With a few simple `GET` routes combined with some templates, as illustrated in Example 3-2, you can serve a classic website easily.

Example 3-2. Sample website

```
Route::get('/', function () {
    return view('welcome');
});

Route::get('about', function () {
    return view('about');
});

Route::get('products', function () {
    return view('products');
});

Route::get('services', function () {
```

```
    return view('services');
});
```

STATIC CALLS

If you have much experience developing with PHP, you might be surprised to see static calls on the `Route` class. This is not actually a static method per se, but rather a service location using Laravel's facades, which we'll cover in Chapter 11.

If you prefer to avoid facades, you can accomplish these same definitions like this:

```
$router->get('/', function () {
    return 'Hello, World!';
});
```

Route Verbs

You might've noticed that we've been using `Route::get()` in our route definitions. This means we're telling Laravel to only match for these routes when the HTTP request uses the `GET` action. But what if it's a form `POST`, or maybe some JavaScript sending `PUT` or `DELETE` requests? There are a few other options for methods to call on a route definition, as illustrated in Example 3-3.

Example 3-3. Route verbs

```
Route::get('/', function () {
    return 'Hello, World!';
});

Route::post('/', function () {
    // Handle someone sending a POST request to this route
});

Route::put('/', function () {
    // Handle someone sending a PUT request to this route
});

Route::delete('/', function () {
    // Handle someone sending a DELETE request to this route
});
```

```
Route::any('/', function () {
    // Handle any verb request to this route
});

Route::match(['get', 'post'], '/', function () {
    // Handle GET or POST requests to this route
});
```

Route Handling

As you've probably guessed, passing a closure to the route definition is not the only way to teach it how to resolve a route. Closures are quick and simple, but the larger your application gets, the clumsier it becomes to put all of your routing logic in one file. Additionally, applications using route closures can't take advantage of Laravel's route caching (more on that later), which can shave up to hundreds of milliseconds off of each request.

The other common option is to pass a controller name and method as a string in place of the closure, as in Example 3-4.

Example 3-4. Routes calling controller methods

```
use App\Http\Controllers\WelcomeController;

Route::get('/', [WelcomeController::class, 'index']);
```

This is telling Laravel to pass requests to that path to the `index()` method of the `App\Http\Controllers\WelcomeController` controller. This method will be passed the same parameters and treated the same way as a closure you might've alternatively put in its place.

LARAVEL'S CONTROLLER/METHOD REFERENCE SYNTAX

Laravel has a convention for how to refer to a particular method in a given controller: *`[ControllerName::class, methodName]`*, known as *tuple syntax* or *callable array syntax*. Sometimes this is just a casual communication convention, but it's also used in real bindings, like in Example 3-4. The first item on the array identifies the controller and the second the method.

Laravel also supports an older "string" syntax (`Route::get('/', 'WelcomeController@index')`), and this is still a common way to describe a method in written communication.

Route Parameters

If the route you're defining has parameters—segments in the URL structure that are variable—it's simple to define them in your route and pass them to your closure (see Example 3-5).

Example 3-5. Route parameters

```
Route::get('users/{id}/friends', function ($id) {
    //
});
```

You can also make your route parameters optional by including a question mark (`?`) after the parameter name, as illustrated in Example 3-6. In this case, you should also provide a default value for the route's corresponding variable.

Example 3-6. Optional route parameters

```
Route::get('users/{id?}', function ($id = 'fallbackId') {
    //
});
```

And you can use regular expressions (regexes) to define that a route should only match if a parameter meets particular requirements, as in Example 3-7.

Example 3-7. Regular expression route constraints

```
Route::get('users/{id}', function ($id) {
    //
})->where('id', '[0-9]+');

Route::get('users/{username}', function ($username) {
    //
})->where('username', '[A-Za-z]+');

Route::get('posts/{id}/{slug}', function ($id, $slug) {
    //
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

As you've probably guessed, if you visit a path that matches a route string but the regex doesn't match the parameter, it won't be matched. Since routes are matched top to bottom, `users/abc` would skip the first closure in Example 3-7, but it would be matched by the second closure, so it would get routed there. On the other hand, `posts/abc/123` wouldn't match any of the closures, so it would return a 404 (Not Found) error.

Laravel also offers convenience methods for common regular expression matching patterns, as you can see in Example 3-8.

Example 3-8. Regular expression route constraint helpers

```
Route::get('users/{id}/friends/{friendname}', function ($id, $friendname) {
    //
})->whereNumber('id')->whereAlpha('friendname');

Route::get('users/{name}', function ($name) {
    //
})->whereAlphaNumeric('name');

Route::get('users/{id}', function ($id) {
    //
})->whereUuid('id');

Route::get('users/{id}', function ($id) {
    //
})->whereUlid('id');

Route::get('friends/types/{type}', function ($type) {
    //
})->whereIn('type', ['acquaintance', 'bestie', 'frenemy']);
```

THE NAMING RELATIONSHIP BETWEEN ROUTE PARAMETERS AND CLOSURE/CONTROLLER METHOD PARAMETERS

As you can see in Example 3-5, it's most common to use the same names for your route parameters (`{id}`) and the method parameters they inject into your route definition (`function ($id)`). But is this necessary?

Unless you're using route model binding, discussed later in this chapter, no. The only thing that defines which route parameter matches with which method parameter is their order (left to right), as you can see here:

```
Route::get('users/{userId}/comments/{commentId}', function (
    $thisIsActuallyTheUserId,
    $thisIsReallyTheCommentId
    ) {
    //
});
```

That having been said, just because you *can* make them different doesn't mean you *should*. I recommend keeping them the same for the sake of future developers, who could get tripped up by inconsistent naming.

Route Names

The simplest way to refer to these routes elsewhere in your application is just by their path. There's a `url()` global helper to simplify that linking in your views, if you need it; see Example 3-9 for an example. The helper will prefix your route with the full domain of your site.

Example 3-9. The `url()` helper

```
<a href="<?php echo url('/'); ?>">
// Outputs <a href="http://myapp.com/">
```

However, Laravel also allows you to name each route, which enables you to refer to it without explicitly referencing the URL. This is helpful because it means you can give simple nicknames to complex routes, and also because linking them by name means you don't have to rewrite your frontend links if the paths change (see Example 3-10).

Example 3-10. Defining route names

```
// Defining a route with name() in routes/web.php:
Route::get('members/{id}', [\App\Http\Controller\MemberController::class,
'show'])
    ->name('members.show');

// Linking the route in a view using the route() helper:
<a href="<?php echo route('members.show', ['id' => 14]); ?>">
```

This example illustrates a few new concepts. First, we're using fluent route definition to add the name, by chaining the `name()` method after the `get()` method. This method allows us to name the route, giving it a short alias to make it easier to reference elsewhere.

In our example, we've named this route `members.show`; *`resourcePlural.action`* is a common convention within Laravel for route and view names.

ROUTE NAMING CONVENTIONS

You can name your route anything you'd like, but the common convention is to use the plural of the resource name, then a period, then the action. So, here are the routes most common for a resource named `photo`:

```
photos.index
photos.create
photos.store
photos.show
photos.edit
photos.update
photos.destroy
```

To learn more about these conventions, see "Resource Controllers".

This example also introduced the `route()` helper. Just like `url()`, it's intended to be used in views to simplify linking to a named route. If the route has no parameters, you can simply pass the route name (`route('members.index')`) and receive a route string (`"http://myapp.com/members"`). If it has parameters, pass them as an array in the second parameter like we did in Example 3-10.

In general, I recommend using route names instead of paths to refer to your routes, and therefore using the `route()` helper instead of the `url()` helper. Sometimes it can get a bit clumsy—for example, if you're working with multiple subdomains—but it provides an incredible level of flexibility to later change the application's routing structure without major penalty.

PASSING ROUTE PARAMETERS TO THE ROUTE() HELPER

When your route has parameters (e.g., `users/id`), you need to define those parameters when you're using the `route()` helper to generate a link to the route.

There are a few different ways to pass these parameters. Let's imagine a route defined as `users/userId/comments/commentId`. If the user ID is `1` and the comment ID is `2`, let's look at a few options we have available to us:

Option 1:

```
route('users.comments.show', [1, 2])
// http://myapp.com/users/1/comments/2
```

Option 2:

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2])
// http://myapp.com/users/1/comments/2
```

Option 3:

```
route('users.comments.show', ['commentId' => 2, 'userId' => 1])
// http://myapp.com/users/1/comments/2
```

Option 4:

```
route('users.comments.show', ['userId' => 1, 'commentId' => 2, 'opt' =>
'a'])
// http://myapp.com/users/1/comments/2?opt=a
```

As you can see, nonkeyed array values are assigned in order; keyed array values are matched with the route parameters matching their keys, and anything left over is added as a query parameter.

Route Groups

Often a group of routes shares a particular characteristic—a certain authentication requirement, a path prefix, or perhaps a controller namespace. Defining these shared characteristics again and again on each route not only seems tedious but also can muddy up the shape of your routes file and obscure some of the structures of your application.

Route groups allow you to reduce this duplication by grouping several routes together and applying any shared configuration settings once to the entire group. Additionally, route groups are visual cues to future developers (and to your own brain) that these routes are grouped together.

To group two or more routes together, you "surround" the route definitions with a route group, as shown in Example 3-11. In reality, you're actually passing a closure to the group definition and defining the grouped routes within that closure.

Example 3-11. Defining a route group

```
Route::group(function () {
    Route::get('hello', function () {
        return 'Hello';
    });
    Route::get('world', function () {
        return 'World';
    });
});
```

By default, a route group doesn't actually do anything. There's no difference between using the group in Example 3-11 and separating a segment of your routes with code comments.

Middleware

Probably the most common use for route groups is to apply middleware to a group of routes. You'll learn more about middleware in Chapter 10, but, among other things, they're what Laravel uses for authenticating users and restricting guest users from using certain parts of a site.

In Example 3-12, we're creating a route group around the `dashboard` and `account` views and applying the `auth` middleware to both. In this example, this means users have to be logged in to the application to view the dashboard or the account page.

Example 3-12. Restricting a group of routes to logged-in users only

```
Route::middleware('auth')->group(function() {
    Route::get('dashboard', function () {
        return view('dashboard');
    });
    Route::get('account', function () {
        return view('account');
    });
});
```

Often, it's clearer and more direct to attach middleware to your routes in the controller instead of at the route definition. You can do this by calling the `middleware()` method in the constructor of your controller. The string you pass to the `middleware()` method is the name of the middleware, and you can optionally chain modifier methods (`only()` and `except()`) to define which methods will receive that middleware:

```
class DashboardController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');

        $this->middleware('admin-auth')
            ->only('editUsers');

        $this->middleware('team-member')
            ->except('editUsers');
    }
}
```

Note that if you're doing a lot of "only" and "except" customizations, that's often a sign that you should break out a new controller for the exceptional routes.

A BRIEF INTRODUCTION TO ELOQUENT

We'll be covering Eloquent, database access, and Laravel's query builder in depth in Chapter 5, but there will be a few references between now and then that will make a basic understanding useful.

Eloquent is Laravel's ActiveRecord database object-relational mapper (ORM), which makes it easy to relate a `Post` class (model) to the `posts` database table and get all records with a call like `Post::all()`.

The query builder is the tool that makes it possible to make calls like `Post::where('active', true)->get()` or even `DB::table('users')->all()`. You're *building* a query by chaining methods one after another.

Path Prefixes

If you have a group of routes that share a segment of their path—for example, if your site's dashboard is prefixed with `/dashboard`—you can use route groups to simplify this structure (see Example 3-13).

Example 3-13. Prefixing a group of routes

```
Route::prefix('dashboard')->group(function () {
    Route::get('/', function () {
        // Handles the path /dashboard
    });
    Route::get('users', function () {
        // Handles the path /dashboard/users
    });
});
```

Note that each prefixed group also has a `/` route that represents the root of the prefix—in Example 3-13 that's `/dashboard`.

Subdomain Routing

Subdomain routing is the same as route prefixing, but it's scoped by subdomain instead of route prefix. There are two primary uses for this.

First, you may want to present different sections of the application (or entirely different applications) to different subdomains. Example 3-14 shows how you can achieve this.

Example 3-14. Subdomain routing

```
Route::domain('api.myapp.com')->group(function () {
    Route::get('/', function () {
        //
    });
});
```

Second, you might want to set part of the subdomain as a parameter, as illustrated in Example 3-15. This is most often done in cases of multitenancy (think Slack or Harvest, where each company gets its own subdomain, like *tighten.slack.co*).

Example 3-15. Parameterized subdomain routing

```
Route::domain('{account}.myapp.com')->group(function () {
    Route::get('/', function ($account) {
        //
    });
    Route::get('users/{id}', function ($account, $id) {
        //
    });
});
```

Note that any parameters for the group get passed into the grouped routes' methods as the first parameter(s).

Name Prefixes

It's common that route names will reflect the inheritance chain of path elements, so `users/comments/5` will be served by a route named `users.comments.show`. In this case, it's common to use a route group around all of the routes that are beneath the `users.comments` resource.

Just like we can prefix URL segments, we can also prefix strings to the route name. With route group name prefixes, we can define that every route within this group should have a given string prefixed to its name. In this

context, we're prefixing "`users.`" to each route name, then "`comments.`" (see Example 3-16).

Example 3-16. Route group name prefixes

```
Route::name('users.')->prefix('users')->group(function () {
    Route::name('comments.')->prefix('comments')->group(function () {
        Route::get('{id}', function () {
            // ...
        })->name('show'); // Route named 'users.comments.show'

        Route::destroy('{id}', function () {})->name('destroy');
    });
});
```

Route Group Controllers

When you're grouping routes that are served by the same controller, such as when we're showing, editing, and deleting users, for example, we can use the route group `controller()` method, as shown in Example 3-17, to avoid having to define the full tuple for every route.

Example 3-17. Route group controllers

```
use App\Http\Controllers\UserController;

Route::controller(UserController::class)->group(function () {
    Route::get('/', 'index');
    Route::get('{id}', 'show');
});
```

Fallback Routes

In Laravel you can define a "fallback route" (which you need to define at the end of your routes file) to catch all unmatched requests:

```
Route::fallback(function () {
    //
});
```

Signed Routes

Many applications regularly send notifications about one-off actions (resetting a password, accepting an invitation, etc.) and provide simple links to take those actions. Let's imagine sending an email confirming the recipient was willing to be added to a mailing list.

There are three ways to send that link:

- Make that URL public and hope no one else discovers the approval URL or modifies their own approval URL to approve someone else.
- Put the action behind authentication, link to the action, and require the user to log in if they're not logged in yet (which, in this case, may be impossible, as many mailing list recipients likely won't be users with accounts).
- "Sign" the link so that it uniquely proves that the user received the link from your email, without them having to log in—something like *http://myapp.com/invitations/5816/yes?signature=030ab0ef6a8237bd86a8b8*.

One simple way to accomplish the last option is to use a feature called *signed URLs*, which makes it easy to build a signature authentication system for sending out authenticated links. These links are composed of the normal route link with a "signature" appended that proves that the URL has not been changed since it was sent (and therefore that no one has modified the URL to access someone else's information).

Signing a Route

To build a signed URL to access a given route, the route must have a name:

```
Route::get('invitations/{invitation}/{answer}', InvitationController::class)
    ->name('invitations');
```

To generate a normal link to this route you would use the `route()` helper, as we've already covered, but you could also use the `URL` facade to do the same thing: `URL::route('invitations', ['invitation' => 12345,`

`'answer' => 'yes']`). To generate a *signed* link to this route, simply use the `signedRoute()` method instead. And if you want to generate a signed route with an expiration, use `temporarySignedRoute()`:

```
// Generate a normal link
URL::route('invitations', ['invitation' => 12345, 'answer' => 'yes']);

// Generate a signed link
URL::signedRoute('invitations', ['invitation' => 12345, 'answer' => 'yes']);

// Generate an expiring (temporary) signed link
URL::temporarySignedRoute(
    'invitations',
    now()->addHours(4),
    ['invitation' => 12345, 'answer' => 'yes']
);
```

USING THE NOW() HELPER

Laravel offers a `now()` helper that's the equivalent of `Carbon::now()`; it returns a Carbon object representative of today, right at this second.

Carbon is a datetime library that's included with Laravel.

Modifying Routes to Allow Signed Links

Now that you've generated a link to your signed route, you need to protect against any unsigned access. The easiest option is to apply the `signed` middleware:

```
Route::get('invitations/{invitation}/{answer}', InvitationController::class)
    ->name('invitations')
    ->middleware('signed');
```

If you'd prefer, you can manually validate using the `hasValidSignature()` method on the `Request` object instead of using the `signed` middleware:

```
class InvitationController
{
```

```
    public function __invoke(Invitation $invitation, $answer, Request
$request)
    {
        if (! $request->hasValidSignature()) {
            abort(403);
        }

        //
    }
}
```

Views

In a few of the route closures we've looked at so far, we've seen something along the lines of `return view('account')`. What's going on here?

In the MVC pattern (Figure 3-1), *views* (or templates) are files that describe what some particular output should look like. You might have views that output JSON or XML or email, but the most common views in a web framework output HTML.

In Laravel, there are two view formats you can use out of the box: plain PHP and Blade templates (see Chapter 4). The difference is in the filename: *about.php* will be rendered with the PHP engine, and *about.blade.php* will be rendered with the Blade engine.

THREE WAYS TO LOAD A VIEW

There are three ways to return a view. For now, just concern yourself with `view()`, but if you ever see `View::make()`, it's the same thing, or you could inject `Illuminate\View\ViewFactory` if you prefer.

Once you've "loaded" a view with the `view()` helper, you have the option to simply return it (as in Example 3-18), which will work fine if the view doesn't rely on any variables from the controller.

Example 3-18. Simple `view()` usage

```
Route::get('/', function () {
   return view('home');
});
```

This code looks for a view in *resources/views/home.blade.php* or *resources/views/home.php* and loads its contents and parses any inline PHP or control structures until you have just the view's output. Once you return it, it's passed on to the rest of the response stack and eventually returned to the user.

But what if you need to pass in variables? Take a look at Example 3-19.

Example 3-19. Passing variables to views

```
Route::get('tasks', function () {
    return view('tasks.index')
        ->with('tasks', Task::all());
});
```

This closure loads the *resources/views/tasks/index.blade.php* or *resources/views/tasks/index.php* view and passes it a single variable named `tasks`, which contains the result of the `Task::all()` method. `Task::all()` is an Eloquent database query you'll learn about in Chapter 5.

Returning Simple Routes Directly with Route::view()

Because it's so common for a route to just return a view with no custom data, Laravel allows you to define a route as a "view" route without even passing the route definition a closure or a controller/method reference, as you can see in Example 3-20.

Example 3-20. `Route::view()`

```
// Returns resources/views/welcome.blade.php
Route::view('/', 'welcome');

// Passing simple data to Route::view()
Route::view('/', 'welcome', ['User' => 'Michael']);
```

Using View Composers to Share Variables with Every View

Sometimes it can become a hassle to pass the same variables over and over. There may be a variable that you want accessible to every view in the site or to a certain class of views or a certain included subview—for example, all views related to tasks or the header partial.

It's possible to share certain variables with every template or just certain templates, like in the following code:

```
view()->share('variableName', 'variableValue');
```

To learn more, check out "View Composers and Service Injection".

Controllers

I've mentioned controllers a few times, but until now, most of the examples have shown route closures. In the MVC pattern, controllers are essentially classes that organize the logic of one or more routes together in one place. Controllers tend to group similar routes together, especially if your application is structured in a traditionally CRUD-like format; in this case, a controller might handle all the actions that can be performed on a particular resource.

> **WHAT IS CRUD?**
>
> CRUD stands for *create*, *read*, *update*, *delete*, which are the four primary operations that web applications most commonly provide on a resource. For example, you can create a new blog post, you can read that post, you can update it, or you can delete it.

It may be tempting to cram all of the application's logic into the controllers, but it's better to think of controllers as the traffic cops that route HTTP requests around your application. Since there are other ways requests can come into your application—cron jobs, Artisan command-line calls, queue jobs, etc.—it's wise to not rely on controllers for much behavior. This

means a controller's primary job is to capture the intent of an HTTP request and pass it on to the rest of the application.

So, let's create a controller. One easy way to do this is with an Artisan command, so from the command line, run the following:

```
php artisan make:controller TaskController
```

> **ARTISAN AND ARTISAN GENERATORS**
>
> Laravel comes bundled with a command-line tool called Artisan. Artisan can be used to run migrations, create users and other database records manually, and perform many other manual, one-time tasks.
>
> Under the `make` namespace, Artisan provides tools for generating skeleton files for a variety of system files. That's what allows us to run `php artisan make:controller`.
>
> To learn more about this and other Artisan features, see Chapter 8.

This will create a new file named *TaskController.php* in *app/Http/Controllers*, with the contents shown in Example 3-21.

Example 3-21. Default generated controller

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TaskController extends Controller
{
    //
}
```

Modify this file as shown in Example 3-22, creating a new public method called `index()`. We'll just return some text there.

Example 3-22. Simple controller example

```
<?php

namespace App\Http\Controllers;
```

```
class TaskController extends Controller
{
    public function index()
    {
        return 'Hello, World!';
    }
}
```

Then, like we learned before, we'll hook up a route to it, as shown in Example 3-23.

Example 3-23. Route for the simple controller

```
// routes/web.php
<?php

use Illuminate\Support\Facades\Route;
use App\Http\Controllers\TaskController;

Route::get('/', [TaskController::class, 'index']);
```

That's it. Visit the `/` route and you'll see the words "Hello, World!"

The most common use of a controller method, then, will be something like Example 3-24, which provides the same functionality as our route closure in Example 3-19.

Example 3-24. Common controller method example

```
// TaskController.php
...
public function index()
{
    return view('tasks.index')
        ->with('tasks', Task::all());
}
```

This controller method loads the *resources/views/tasks/index.blade.php* or *resources/views/tasks/index.php* view and passes it a single variable named `tasks`, which contains the result of the `Task::all()` Eloquent method.

GENERATING RESOURCE CONTROLLERS

If you want to create a resource controller with autogenerated methods for all the basic resource routes like `create()` and `update()`, you can pass the `--resource` flag when using `php artisan make:controller`:

```
php artisan make:controller TaskController --resource
```

Getting User Input

The second most common action to perform in a controller method is to take input from the user and act on it. That introduces a few new concepts, so let's take a look at a bit of sample code and walk through the new pieces.

First, let's bind our route; see Example 3-25.

Example 3-25. Binding basic form actions

```
// routes/web.php
Route::get('tasks/create', [TaskController::class, 'create']);
Route::post('tasks', [TaskController::class, 'store']);
```

Notice that we're binding the `GET` action of `tasks/create` (which shows a form for creating a new task) and the `POST` action of `tasks` (which is where our form will `POST` to when we're creating a new task). We can assume the `create()` method in our controller just shows a form, so let's look at the `store()` method in Example 3-26.

Example 3-26. Common form input controller method

```
// TaskController.php
...
public function store()
{
    Task::create(request()->only(['title', 'description']));

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

This example makes use of Eloquent models and the `redirect()` functionality, and we'll talk about them more later, but for now let's talk quickly about how we're getting our data here.

We're using the `request()` helper to represent the HTTP request (more on that later) and using its `only()` method to pull just the `title` and `description` fields the user submitted.

We're then passing that data into the `create()` method of our `Task` model, which creates a new instance of the `Task` with `title` set to the passed-in title and `description` set to the passed-in description. Finally, we redirect back to the page that shows all tasks.

There are a few layers of abstraction at work here, which we'll cover in a second, but know that the data coming from the `only()` method comes from the same pool of data all common methods used on the `Request` object draw from, including `all()` and `get()`. The set of data each of these methods is pulling from represents all user-provided data, whether from query parameters or `POST` values. So, our user filled out two fields on the "add task" page: "title" and "description."

To break down the abstraction a bit, `request()->only()` takes an associative array of input names and returns them:

```
request()->only(['title', 'description']);
// returns:
[
    'title' => 'Whatever title the user typed on the previous page',
    'description' => 'Whatever description the user typed on the previous
page',
]
```

And `Task::create()` takes an associative array and creates a new task from it:

```
Task::create([
    'title' => 'Buy milk',
    'description' => 'Remember to check the expiration date this time,
```

```
    Norbert!',
    ]);
```

Combining them together creates a task with just the user-provided "title" and "description" fields.

Injecting Dependencies into Controllers

Laravel's facades and global helpers present a simple interface to the most useful classes in Laravel's codebase. You can get information about the current request and user input, the session, caches, and much more.

But if you prefer to inject your dependencies, or if you want to use a service that doesn't have a facade or a helper, you'll need to find some way to bring instances of these classes into your controller.

This is our first exposure to Laravel's service container. For now, if this is unfamiliar, you can think about it as a little bit of Laravel magic; or, if you want to know more about how it's actually functioning, you can skip ahead to Chapter 11.

All controller methods (including the constructors) are resolved out of Laravel's container, which means anything you typehint that the container knows how to resolve will be automatically injected.

TYPEHINTS IN PHP

Typehinting in PHP means putting the name of a class or interface in front of a variable in a method signature:

```
public function __construct(Logger $logger) {}
```

This typehint is telling PHP that whatever is passed into the method *must* be of type `Logger`, which could be either an interface or a class.

As a nice example, what if you'd prefer having an instance of the `Request` object instead of using the global helper? Just typehint

`Illuminate\Http\Request` in your method parameters, like in Example 3-27.

Example 3-27. Controller method injection via typehinting

```
// TaskController.php
...
public function store(\Illuminate\Http\Request $request)
{
    Task::create($request->only(['title', 'description']));

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

So, you've defined a parameter that must be passed into the `store()` method. And since you typehinted it, and since Laravel knows how to resolve that class name, you're going to have the `Request` object ready for you to use in your method with no work on your part. No explicit binding, no anything else—it's just there as the `$request` variable.

And, as you can tell from comparing Examples 3-26 and 3-27, the `request()` helper and the `Request` object behave exactly the same.

Resource Controllers

Sometimes naming the methods in your controllers can be the hardest part of writing a controller. Thankfully, Laravel has some conventions for all of the routes of a traditional REST/CRUD controller (called a *resource controller* in Laravel); additionally, it comes with a generator out of the box and a convenience route definition that allows you to bind an entire resource controller at once.

To see the methods that Laravel expects for a resource controller, let's generate a new controller from the command line:

```
php artisan make:controller MySampleResourceController --resource
```

Now open *app/Http/Controllers/MySampleResourceController.php*. You'll see it comes prefilled with quite a few methods. Let's walk through what

each represents. We'll use a `Task` as an example.

The methods of Laravel's resource controllers

Remember the table from earlier? Table 3-1 shows the HTTP verb, the URL, the controller method name, and the name for each of these default methods that are generated in Laravel's resource controllers.

Binding a resource controller

So, we've seen that these are the conventional route names to use in Laravel, and also that it's easy to generate a resource controller with methods for each of these default routes. Thankfully, you don't have to generate routes for each of these controller methods by hand, if you don't want to. There's a trick for that, called *resource controller binding*. Take a look at Example 3-28.

Example 3-28. Resource controller binding

```
// routes/web.php
Route::resource('tasks', TaskController::class);
```

This will automatically bind all of the routes listed in Table 3-1 for this resource to the appropriate method names on the specified controller. It'll also name these routes appropriately; for example, the `index()` method on the `tasks` resource controller will be named `tasks.index`.

ARTISAN ROUTE:LIST

If you ever find yourself in a situation where you're wondering what routes your current application has available, there's a tool for that: from the command line, run `php artisan route:list` and you'll get a listing of all of the available routes. I prefer `php artisan route:list --exclude-vendor` so I don't see all the weird routes my dependencies register in order for them to operate (see Figure 3-2).

```
mattstauffer at LaunchpdMcQuack in ~/RealSites/book-up-and-running
○ php artisan route:list --except-vendor

  GET|HEAD  / ..................................................
  GET|HEAD  api/user ...........................................
  GET|HEAD  dogs ............................. DogsController@index
  GET|HEAD  dogs/create ...................... DogsController@create
  POST      dogs/store ....................... DogsController@store
  GET|HEAD  dogs/{dog} ........................ DogsController@show
  PUT       dogs/{dog} ....................... DogsController@update
  DELETE    dogs/{dog} ..................... DogsController@destroy
  GET|HEAD  dogs/{dog}/edit ................... DogsController@edit

                                                Showing [9] routes
```

Figure 3-2. artisan route:list

API Resource Controllers

When you're working with RESTful APIs, the list of potential actions on a resource is not the same as it is with an HTML resource controller. For example, you can send a `POST` request to an API to create a resource, but you can't really "show a create form" in an API.

To generate an *API resource controller*, which is a controller that has the same structure as a resource controller except it excludes the *create* and *edit* actions, pass the `--api` flag when creating a controller:

```
php artisan make:controller MySampleResourceController --api
```

To bind an API resource controller, use the `apiResource()` method instead of the `resource()` method, as shown in Example 3-29.

Example 3-29. API resource controller binding

```
// routes/web.php
Route::apiResource('tasks', TaskController::class);
```

Single Action Controllers

There will be times in your applications when a controller should only service a single route. You may find yourself wondering how to name the controller method for that route. Thankfully, you can point a single route at

a single controller without concerning yourself with naming the one method.

As you may already know, the `__invoke()` method is a PHP magic method that allows you to "invoke" an instance of a class, treating it like a function and calling it. This is the tool Laravel's *single action controllers* use to allow you to point a route to a single controller, as you can see in Example 3-30.

Example 3-30. Using the `__invoke()` method

```
// \App\Http\Controllers\UpdateUserAvatar.php
public function __invoke(User $user)
{
    // Update the user's avatar image
}

// routes/web.php
Route::post('users/{user}/update-avatar', UpdateUserAvatar::class);
```

Route Model Binding

One of the most common routing patterns is that the first line of any controller method tries to find the resource with the given ID, like in Example 3-31.

Example 3-31. Getting a resource for each route

```
Route::get('conferences/{id}', function ($id) {
    $conference = Conference::findOrFail($id);
});
```

Laravel provides a feature that simplifies this pattern called *route model binding*. This allows you to define that a particular parameter name (e.g., `{conference}`) will indicate to the route resolver that it should look up an Eloquent database record with that ID and then pass it in as the parameter *instead* of just passing the ID.

There are two kinds of route model binding: implicit and custom (or explicit).

Implicit Route Model Binding

The simplest way to use route model binding is to name your route parameter something unique to that model (e.g., name it `$conference` instead of `$id`), then typehint that parameter in the closure/controller method and use the same variable name there. It's easier to show than to describe, so take a look at Example 3-32.

Example 3-32. Using an implicit route model binding

```
Route::get('conferences/{conference}', function (Conference $conference) {
    return view('conferences.show')->with('conference', $conference);
});
```

Because the route parameter (`{conference}`) is the same as the method parameter (`$conference`), and the method parameter is typehinted with a `Conference` model (`Conference $conference`), Laravel sees this as a route model binding. Every time this route is visited, the application will assume that whatever is passed into the URL in place of `{conference}` is an ID that should be used to look up a `Conference`, and then that resulting model instance will be passed in to your closure or controller method.

CUSTOMIZING THE ROUTE KEY FOR AN ELOQUENT MODEL

Any time an Eloquent model is looked up via a URL segment (usually because of route model binding), the default column Eloquent will look it up by is its primary key (ID).

To change the column your Eloquent model uses for URL lookups in all your routes, add a method to your model named `getRouteKeyName()`:

```
public function getRouteKeyName()
{
    return 'slug';
}
```

Now, a URL like `conferences/{conference}` will expect to get an entry from the slug column instead of the ID and will perform its lookups accordingly.

CUSTOMIZING THE ROUTE KEY IN A SPECIFIC ROUTE

In Laravel, you can also change the route key on a specific route instead of globally by appending a colon and the column name in the route definition:

```
Route::get(
    'conferences/{conference:slug}',
    function (Conference $conference) {
        return view('conferences.show')
            ->with('conference', $conference);
    });
```

If you have two dynamic segments in your URL (for example: `organizers/{organizer}/conferences/{conference:slug}`), Laravel will automatically attempt to scope the second model's queries to only those related to the first. So it'll check the `Organizer` model for a `conferences` relationship, and if it exists, only return Conferences that are associated with the `Organizer` found by the first segment.

```
use App\Models\Conference;
use App\Models\Organizer;

Route::get(
    'organizers/{organizer}/conferences/{conference:slug}',
    function (Organizer $organizer, Conference $conference) {
        return $conference;
    });
```

Custom Route Model Binding

To manually configure route model bindings, add a line like the one in Example 3-33 to the `boot()` method in `App\Providers\RouteServiceProvider`.

Example 3-33. Adding a route model binding

```
    public function boot()
    {
        // Perform the binding
        Route::model('event', Conference::class);
    }
```

You've now specified that whenever a route has a parameter in its definition named `{event}`, as demonstrated in Example 3-34, the route resolver will return an instance of the `Conference` class with the ID of that URL parameter.

Example 3-34. Using an explicit route model binding

```
Route::get('events/{event}', function (Conference $event) {
    return view('events.show')->with('event', $event);
});
```

Route Caching

If you're looking to squeeze every millisecond out of your load time, you may want to take a look at *route caching*. One of the pieces of Laravel's bootstrap that can take anywhere from a few dozen to a few hundred milliseconds is parsing the *routes/** files, and route caching speeds up this process dramatically.

To cache your routes file, you need to be using all controller, redirect, view, and resource routes (no route closures). If your app isn't using any route closures, you can run `php artisan route:cache` and Laravel will serialize the results of your *routes/** files. If you want to delete the cache, run `php artisan route:clear`.

Here's the drawback: Laravel will now match routes against that cached file instead of your actual *routes/** files. You can make endless changes to your routes files, and they won't take effect until you run `route:cache` again. This means you'll have to recache every time you make a change, which introduces a lot of potential for confusion.

Here's what I would recommend instead: since Git ignores the route cache file by default anyway, consider only using route caching on your

production server, and run the `php artisan route:cache` command every time you deploy new code (whether via a Git post-deploy hook, a Forge deploy command, or as a part of whatever other deployment system you use). This way, you won't have confusing local development issues, but your remote environment will still benefit from route caching.

Form Method Spoofing

Sometimes you need to manually define which HTTP verb a form should send as. HTML forms only allow for `GET` or `POST`, so if you want any other sort of verb, you'll need to specify that yourself.

HTTP Verbs in Laravel

As we've seen already, you can define which verbs a route will match in the route definition using `Route::get()`, `Route::post()`, `Route::any()`, or `Route::match()`. You can also match with `Route::patch()`, `Route::put()`, and `Route::delete()`.

But how does one send a request other than `GET` with a web browser? First, the `method` attribute in an HTML form determines its HTTP verb: if your form has a `method` of `"GET"`, it will submit via query parameters and a `GET` method; if the form has a `method` of `"POST"`, it will submit via the post body and a `POST` method.

JavaScript frameworks make it easy to send other requests, like `DELETE` and `PATCH`. But if you find yourself needing to submit HTML forms in Laravel with verbs other than `GET` or `POST`, you'll need to use *form method spoofing*, which means spoofing the HTTP method in an HTML form.

HTTP Method Spoofing in HTML Forms

To inform Laravel that the form you're currently submitting should be treated as something other than a `POST`, add a hidden variable named

`_method` with the value of `"PUT"`, `"PATCH"`, or `"DELETE"`, and Laravel will match and route that form submission as if it were actually a request with that verb.

The form in Example 3-35, since it's passing Laravel the method of `"DELETE"`, will match routes defined with `Route::delete()`, but not those with `Route::post()`.

Example 3-35. Form method spoofing

```
<form action="/tasks/5" method="POST">
    <input type="hidden" name="_method" value="DELETE">
    <!-- or: -->
    @method('DELETE')
</form>
```

CSRF Protection

If you've tried to submit a form in a Laravel application already, including the one in Example 3-35, you've likely run into the dreaded `TokenMismatchException`.

By default, all routes in Laravel except "read-only" routes (those using `GET`, `HEAD`, or `OPTIONS`) are protected against cross-site request forgery (CSRF) attacks by requiring a token, in the form of an input named `_token`, to be passed along with each request. This token is generated at the start of every session, and every non–read-only route compares the submitted `_token` against the session token.

WHAT IS CSRF?

A *cross-site request forgery* is when one website pretends to be another. The goal is for someone to hijack your users' access to your website by submitting forms from *their* website to *your* website via the logged-in user's browser.

The best way around CSRF attacks is to protect all inbound routes—`POST`, `DELETE`, etc.—with a token, which Laravel does out of the box.

You have two options for getting around this CSRF error. The first, and preferred, method is to add the `_token` input to each of your submissions. In HTML forms, there's a simple way to do it, as you can see in Example 3-36.

Example 3-36. CSRF tokens

```
<form action="/tasks/5" method="POST">
    @csrf
</form>
```

In JavaScript applications, it takes a bit more work, but not much. The most common solution for sites using JavaScript frameworks is to store the token on every page in a `<meta>` tag like this one:

```
<meta name="csrf-token" content="<?php echo csrf_token(); ?>">
```

Storing the token in a `<meta>` tag makes it easy to bind it to the correct HTTP header, which you can do once globally for all requests from your JavaScript framework, like in Example 3-37.

Example 3-37. Globally binding a header for CSRF

```
// In jQuery:
$.ajaxSetup({
    headers: {
        'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
    }
});

// With Axios: it automatically retrieves it from a cookie. Nothing to do!
```

Laravel will check the `X-CSRF-TOKEN` (and `X-XSRF-TOKEN`, which Axios and other JavaScript frameworks like Angular use) on every request, and valid tokens passed there will mark the CSRF protection as satisfied.

BINDING CSRF TOKENS WITH VUE RESOURCE

Bootstrapping the CSRF token into Vue Resource looks a bit different than it does for Laravel; see the Vue Resource docs for examples.

Redirects

So far, the only things we've explicitly talked about returning from a controller method or route definition have been views. But there are a few other structures we can return to give the browser instructions on how to behave.

First, let's cover the *redirect*. You've already seen a few of these in other examples. There are two common ways to generate a redirect; we'll use the `redirect()` global helper here, but you may prefer the facade. Both create an instance of `Illuminate\Http\RedirectResponse`, perform some convenience methods on it, and then return it. You can also do this manually, but you'll have to do a little more work yourself. Take a look at Example 3-38 to see a few ways you can return a redirect.

Example 3-38. Different ways to return a redirect

```
// Using the global helper to generate a redirect response
Route::get('redirect-with-helper', function () {
    return redirect()->to('login');
});

// Using the global helper shortcut
Route::get('redirect-with-helper-shortcut', function () {
    return redirect('login');
});

// Using the facade to generate a redirect response
Route::get('redirect-with-facade', function () {
    return Redirect::to('login');
});

// Using the Route::redirect shortcut
Route::redirect('redirect-by-route', 'login');
```

Note that the `redirect()` helper exposes the same methods as the `Redirect` facade, but it also has a shortcut; if you pass parameters directly to the helper instead of chaining methods after it, it's a shortcut to the `to()` redirect method.

Also note that the (optional) third parameter for the `Route::redirect()` route helper can be the status code (e.g., 302) for your redirect.

redirect()->to()

The method signature for the `to()` method for redirects looks like this:

```
function to($to = null, $status = 302, $headers = [], $secure = null)
```

`$to` is a valid internal path, `$status` is the HTTP status (defaulting to `302`), `$headers` allows you to define which HTTP headers to send along with your redirect, and `$secure` allows you to override the default choice of `http` versus `https` (which is normally set based on your current request URL). Example 3-39 shows an example of its use.

Example 3-39. `redirect()->to()`

```
Route::get('redirect', function () {
    return redirect()->to('home');

    // Or same, using the shortcut:

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

redirect()->route()

The `route()` method is the same as the `to()` method, but rather than pointing to a particular path, it points to a particular route name (see Example 3-40).

Example 3-40. `redirect()->route()`

```
Route::get('redirect', function () {
    return redirect()->route('conferences.index');
});
```

Note that since some route names require parameters, its parameter order is a little different. `route()` has an optional second parameter for the route parameters:

```
function route($to = null, $parameters = [], $status = 302, $headers = [])
```

So, using it might look a little like Example 3-41.

Example 3-41. `redirect()->route()` with parameters

```
Route::get('redirect', function () {
    return to_route('conferences.show', [
        'conference' => 99,
    ];
});
```

> **REDIRECT TO_ROUTE() HELPER**
>
> You can use the `to_route()` helper as an alias for the `redirect()->route()` method. The signature for both is the same:
>
> ```
> Route::get('redirect', function () {
>     return to_route('conferences.show', ['conference' => 99]);
> });
> ```

redirect()->back()

Because of some of the built-in conveniences of Laravel's session implementation, your application will always have knowledge of what the user's previously visited page was. That opens up the opportunity for a `redirect()->back()` redirect, which simply redirects the user to whatever page they came from. There's also a global shortcut for this: `back()`.

Other Redirect Methods

The redirect service provides other methods that are less commonly used, but still available:

`refresh()`

: Redirects to the same page the user is currently on.

`away()`

: Allows for redirecting to an external URL without the default URL validation.

`secure()`

Like `to()` with the `secure` parameter set to `"true"`.

`action()`

Allows you to link to a controller and method in one of two ways: as a string (`redirect()->action('MyController@myMethod')`) or as a tuple (`redirect()\->action([MyController::class, 'myMethod'])`).

`guest()`

Used internally by the authentication system (discussed in Chapter 9); when a user visits a route they're not authenticated for, this captures the "intended" route and then redirects the user (usually to a login page).

`intended()`

Also used internally by the authentication system; after a successful authentication, this grabs the "intended" URL stored by the `guest()` method and redirects the user there.

redirect()->with()

While it is structured similarly to the other methods you can call on `redirect()`, `with()` is different in that it doesn't define where you're redirecting to, but what data you're passing along with the redirect. When you're redirecting users to different pages, you often want to pass certain data along with them. You could manually flash the data to the session, but Laravel has some convenience methods to help you with that.

Most commonly, you can pass along either an array of keys and values or a single key and value using `with()`, like in Example 3-42. This saves your `with()` data to the session just for the next page load.

Example 3-42. Redirect with data

```
Route::get('redirect-with-key-value', function () {
    return redirect('dashboard')
        ->with('error', true);
});

Route::get('redirect-with-array', function () {
    return redirect('dashboard')
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```

> **CHAINING METHODS ON REDIRECTS**
>
> As with many other facades, most calls to the `Redirect` facade can accept fluent method chains, like the `with()` calls in Example 3-42. You'll learn more about fluency in "What Is a Fluent Interface?".

You can also use `withInput()`, as in Example 3-43, to redirect with the user's form input flashed; this is most common in the case of a validation error, where you want to send the user back to the form they just came from.

Example 3-43. Redirect with form input

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

Route::post('form', function () {
    return redirect('form')
        ->withInput()
        ->with(['error' => true, 'message' => 'Whoops!']);
});
```

The easiest way to get the flashed input that was passed with `withInput()` is using the `old()` helper, which can be used to get all old input (`old()`) or just the value for a particular key, as shown in the following example, with the second parameter as the default if there is no old value. You'll commonly see this in views, which allows this HTML to be used both on the "create" and the "edit" view for this form:

```
<input name="username" value="<?=
    old('username', 'Default username instructions here');
?>">
```

Speaking of validation, there is also a useful method for passing errors along with a redirect response: `withErrors()`. You can pass it any "provider" of errors, which may be an error string, an array of errors, or, most commonly, an instance of the Illuminate `Validator`, which we'll cover in Chapter 10. Example 3-44 shows an example of its use.

Example 3-44. Redirect with errors

```
Route::post('form', function (Illuminate\Http\Request $request) {
    $validator = Validator::make($request->all(), $this->validationRules);

    if ($validator->fails()) {
        return back()
            ->withErrors($validator)
            ->withInput();
    }
});
```

`withErrors()` automatically shares an `$errors` variable with the views of the page it's redirecting to, for you to handle however you'd like.

THE VALIDATE() METHOD ON REQUESTS

Don't like how Example 3-44 looks? There's a simple and powerful tool that will make it easy for you to clean up that code. Read more in "validate() on the Request Object".

Aborting the Request

Aside from returning views and redirects, the most common way to exit a route is to abort. There are a few globally available methods (`abort()`, `abort_if()`, and `abort_unless()`), which optionally take HTTP status codes, a message, and a headers array as parameters.

As Example 3-45 shows, `abort_if()` and `abort_unless()` take a first parameter that is evaluated for its truthiness and perform the abort depending on the result.

Example 3-45. 403 Forbidden aborts

```
Route::post('something-you-cant-do', function (Illuminate\Http\Request $request)
{
    abort(403, 'You cannot do that!');
    abort_unless($request->has('magicToken'), 403);
    abort_if($request->user()->isBanned, 403);
});
```

Custom Responses

There are a few other options available for us to return, so let's go over the most common responses after views, redirects, and aborts. Just like with redirects, you can run these methods on either the `response()` helper or the `Response` facade.

response()->make()

If you want to create an HTTP response manually, just pass your data into the first parameter of `response()->make()`: for example, `return response()->make(Hello, World!)`. Once again, the second parameter is the HTTP status code and the third is your headers.

response()->json() and ->jsonp()

To create a JSON-encoded HTTP response manually, pass your JSON-able content (arrays, collections, or whatever else) to the `json()` method: for example `return response()->json(User::all())`. It's just like `make()`, except it `json_encodes` your content and sets the appropriate headers.

response()->download(), ->streamDownload(), and ->file()

To send a file for the end user to download, pass either an `SplFileInfo` instance or a string filename to `download()`, with an optional second parameter of the download filename: for example, `return response()->download('file501751.pdf', 'myFile.pdf')`, which would send a file that's at *file501751.pdf* and rename it, as it's sent, to *myFile.pdf*.

To display the same file in the browser (if it's a PDF or an image or something else the browser can handle), use `response()->file()` instead, which takes the same parameters as `response->download()`.

If you want to make some content from an external service available as a download without having to write it directly to your server's disk, you can stream the download using `response()->streamDownload()`. This method expects as parameters a closure that echoes a string, a filename, and optionally an array of headers; see Example 3-46.

Example 3-46. Streaming downloads from external servers

```
return response()->streamDownload(function () {
    echo DocumentService::file('myFile')->getContent();
}, 'myFile.pdf');
```

Testing

In some other communities the idea of unit-testing controller methods is common, but within Laravel (and most of the PHP community) it's typical to rely on *application testing* to test the functionality of routes.

For example, to verify that a `POST` route works correctly, we can write a test like Example 3-47.

Example 3-47. Writing a simple `POST` route test

```
// tests/Feature/AssignmentTest.php
public function test_post_creates_new_assignment()
{
    $this->post('/assignments', [
        'title' => 'My great assignment',
    ]);

    $this->assertDatabaseHas('assignments', [
```

```
        'title' => 'My great assignment',
    ]);
}
```

Did we directly call the controller methods? No. But we ensured that the goal of this route—to receive a `POST` and save its important information to the database—was met.

You can also use similar syntax to visit a route and verify that certain text shows up on the page, or that clicking certain buttons does certain things (see Example 3-48).

Example 3-48. Writing a simple `GET` route test

```
// AssignmentTest.php
public function test_list_page_shows_all_assignments()
{
    $assignment = Assignment::create([
        'title' => 'My great assignment',
    ]);

    $this->get('/assignments')
        ->assertSee('My great assignment');
}
```

TL;DR

Laravel's routes are defined in *routes/web.php* and *routes/api.php*. You can define the expected path for each route, which segments are static and which are parameters, which HTTP verbs can access the route, and how to resolve it. You can also attach middleware to routes, group them, and give them names.

What is returned from the route closure or controller method dictates how Laravel responds to the user. If it's a string or a view, it's presented to the user; if it's other sorts of data, it's converted to JSON and presented to the user; and if it's a redirect, it forces a redirect.

Laravel provides a series of tools and conveniences to simplify common routing-related tasks and structures. These include resource controllers, route model binding, and form method spoofing.