VERSION

6.x

Search Docs

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);
```

Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the `match` method. Or, you may even register a route that responds to all HTTP verbs using the `any` method:

```
Route::match(['get', 'post'], '/', function () {
    //
});

Route::any('/', function () {
    //
});
```

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:

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

Redirect Routes

If you are defining a route that redirects to another URI, you may use the `Route::redirect` method. This method provides a convenient shortcut so that you do not have to define a full route or controller for performing a simple redirect:

```
Route::redirect('/here', '/there');
```

By default, `Route::redirect` returns a `302` status code. You may customize the status code using the optional third parameter:

```
Route::redirect('/here', '/there', 301);
```

You may use the `Route::permanentRedirect` method to return a `301` status code:

```
Route::permanentRedirect('/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:

```
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;
});

Route::get('user/{name?}', function ($name = 'John') {
    return $name;
});
```

Regular Expression Constraints

You may constrain the format of your route parameters using the `where` method on a route instance. The `where` method accepts the name of the parameter and a regular expression defining how the parameter should be constrained:

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

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

Route::get('user/{id}/{name}', function ($id, $name) {
    //
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

Global Constraints

If you would like a route parameter to always be constrained by a given regular expression, you may use the `pattern` method. You should define these patterns in the `boot` method of your `RouteServiceProvider`:

```
/**
 * Define your route model bindings, pattern filters, etc.
 *
 * @return void
 */
public function boot()
{
    Route::pattern('id', '[0-9]+');

    parent::boot();
}
```

Once the pattern has been defined, it is automatically applied to all routes using that parameter name:

```
Route::get('user/{id}', function ($id) {
    // Only executed if {id} is numeric...
});
```

Encoded Forward Slashes

The Laravel routing component allows all characters except `/`. You must explicitly allow `/` to be part of your placeholder using a `where` condition regular expression:

```
Route::get('search/{search}', function ($search) {
    return $search;
})->where('search', '.*');
```

> Encoded forward slashes are only supported within the last route segment.

Named Routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the `name` method onto the route definition:

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

You may also specify route names for controller actions:

```
Route::get('user/profile', 'UserProfileController@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:

```
// 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:

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

$url = route('profile', ['id' => 1]);
```

If you pass additional parameters in the array, those key / value pairs will automatically be added to the generated URL's query string:

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

$url = route('profile', ['id' => 1, 'photos' => 'yes']);

// /user/1/profile?photos=yes
```

Inspecting The Current Route

If you would like to determine if the current request was routed to a given named route, you may use the `named` method on a Route instance. For example, you may check the current route name from a route middleware:

```php
/**
 * Handle an incoming request.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  \Closure  $next
 * @return mixed
 */
public function handle($request, Closure $next)
{
    if ($request->route()->named('profile')) {
        //
    }

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

Route Groups

Route groups allow you to share route attributes, such as middleware 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:

```
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:

```
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.

Subdomain Routing

Route groups may also be used to handle subdomain routing. Subdomains may be assigned route parameters just like route URIs, allowing you to capture a portion of the subdomain for usage in your route or controller. The subdomain may be specified by calling the `domain` method before defining the group:

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

In order to ensure your subdomain routes are reachable, you should register subdomain routes before registering root domain routes. This will prevent root domain routes from overwriting subdomain routes which have the same URI path.

Route Prefixes

The `prefix` method may be used to prefix each route in the group with a given URI. For example, you may want to prefix all route URIs within the group with `admin`:

```
Route::prefix('admin')->group(function () {
    Route::get('users', function () {
        // Matches The "/admin/users" URL
    });
});
```

Route Name Prefixes

The `name` method may be used to prefix each route name in the group with a given string. For example, you may want to prefix all of the grouped route's names with `admin`. The given string is prefixed to the route name exactly as it is specified, so we will be sure to provide the trailing `.` character in the prefix:

```
Route::name('admin.')->group(function () {
    Route::get('users', function () {
        // Route assigned name "admin.users"...
    })->name('users');
});
```

Route Model Binding

When injecting a model ID to a route or controller action, you will often query 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.

Customizing The Resolution Logic

If you wish to use your own resolution logic, you may use the `Route::bind` method. The `Closure` you pass to the `bind` method will receive the value of the URI segment and should return the instance of the class that should be injected into the route:

```
/**
 * Bootstrap any application services.
 *
 * @return void
```

```
     */
    public function boot()
    {
        parent::boot();

        Route::bind('user', function ($value) {
            return App\User::where('name', $value)->firstOrFail();
        });
    }
```

Alternatively, you may override the `resolveRouteBinding` method on your Eloquent model. This method will receive the value of the URI segment and should return the instance of the class that should be injected into the route:

```
/**
 * Retrieve the model for a bound value.
 *
 * @param  mixed  $value
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveRouteBinding($value)
{
    return $this->where('name', $value)->firstOrFail();
}
```

Fallback Routes

Using the `Route::fallback` method, you may define a route that will be executed when no other route matches the incoming request. Typically, unhandled requests will automatically render a "404" page via your application's exception handler. However, since you may define the `fallback` route within your `routes/web.php` file, all middleware in the `web` middleware group will apply to the route. You are free to add additional middleware to this route as needed:

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

The fallback route should always be the last route registered by your application.

Rate Limiting

Laravel includes a middleware to rate limit access to routes within your application. To get started, assign the `throttle` middleware to a route or a group of routes. The `throttle` middleware accepts two parameters that determine the maximum number of requests that can be made in a given number of minutes. For example, let's specify that an authenticated user may access the following group of routes 60 times per minute:

```
Route::middleware('auth:api', 'throttle:60,1')->group(function () {
    Route::get('/user', function () {
        //
    });
});
```

Dynamic Rate Limiting

You may specify a dynamic request maximum based on an attribute of the authenticated `User` model. For example, if your `User` model contains a

`rate_limit` attribute, you may pass the name of the attribute to the `throttle` middleware so that it is used to calculate the maximum request count:

```
Route::middleware('auth:api', 'throttle:rate_limit,1')->group(function () {
    Route::get('/user', function () {
        //
    });
});
```

Distinct Guest & Authenticated User Rate Limits

You may specify different rate limits for guest and authenticated users. For example, you may specify a maximum of `10` requests per minute for guests `60` for authenticated users:

```
Route::middleware('throttle:10|60,1')->group(function () {
    //
});
```

You may also combine this functionality with dynamic rate limits. For example, if your `User` model contains a `rate_limit` attribute, you may pass the name of the attribute to the `throttle` middleware so that it is used to calculate the maximum request count for authenticated users:

```
Route::middleware('auth:api', 'throttle:10|rate_limit,1')->group(function ()
    Route::get('/user', function () {
        //
    });
});
```

Rate Limit Segments

Typically, you will probably specify one rate limit for your entire API. However, your application may require different rate limits for different segments of your API. If this is the case, you will need to pass a segment name as the third argument to the `throttle` middleware:

```
Route::middleware('auth:api')->group(function () {
    Route::middleware('throttle:60,1,default')->group(function () {
        Route::get('/servers', function () {
            //
        });
    });

    Route::middleware('throttle:60,1,deletes')->group(function () {
        Route::delete('/servers/{id}', function () {
            //
        });
    });
});
```

Form Method Spoofing

HTML forms do not support `PUT`, `PATCH` or `DELETE` actions. So, when defining `PUT`, `PATCH` or `DELETE` routes that are called from an HTML form, you will need to add a hidden `_method` field to the form. The value sent with the `_method` field will be used as the HTTP request method:

```
<form action="/foo/bar" method="POST">
    <input type="hidden" name="_method" value="PUT">
    <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

You may use the `@method` Blade directive to generate the `_method` input:

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

Accessing The Current Route

You may use the `current`, `currentRouteName`, and `currentRouteAction` methods on the `Route` facade to access information about the route handling the incoming request:

```
$route = Route::current();

$name = Route::currentRouteName();

$action = Route::currentRouteAction();
```

Refer to the API documentation for both the underlying class of the Route facade and Route instance to review all accessible methods.

Laravel

Highlights +

Resources +

Partners +

Ecosystem +

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.