Routing

Basic Routing

The most basic Laravel routes accept a URI and a closure, providing a very simple and expressive method of defining routes and behavior without complicated routing configuration files:

```
use Illuminate\Support\Facades\Route;

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

The Default Route Files

All Laravel routes are defined in your route files, which are located in the `routes` directory. These files are automatically loaded by Laravel using the configuration specified in your application's `bootstrap/app.php` file. 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.

For most applications, you will begin by defining routes in your `routes/web.php` file. The routes defined in `routes/web.php` may be accessed by entering the defined route's URL in your browser. For example, you may access the following route by navigating to `http://example.com/user` in your browser:

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

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

API Routes

If your application will also offer a stateless API, you may enable API routing using the `install:api` Artisan command:

```
php artisan install:api
```

The `install:api` command installs Laravel Sanctum, which provides a robust, yet simple API token authentication guard which can be used to authenticate third-party API consumers, SPAs, or mobile applications. In addition, the `install:api` command creates the `routes/api.php` file:

```
Route::get('/user', function (Request $request) {
    return $request->user();
})->middleware('auth:sanctum');
```

The routes in `routes/api.php` are stateless and are assigned to the `api` middleware group. Additionally, the `/api` URI prefix is automatically applied to these routes, so you do not need to manually apply it to every route in the file. You may change the prefix by modifying your application's `bootstrap/app.php` file:

```
->withRouting(
    api: __DIR__.'/../routes/api.php',
    apiPrefix: 'api/admin',
    // ...
)
```

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 () {
    // ...
});
```

> [!NOTE] When defining multiple routes that share the same URI, routes using the `get`, `post`, `put`, `patch`, `delete`, and `options` methods should be defined before routes using the `any`, `match`, and `redirect` methods. This ensures the incoming request is matched with the correct route.

Dependency Injection

You may type-hint any dependencies required by your route in your route's callback signature. The declared dependencies will automatically be resolved and injected into the callback by the Laravel service container. For example, you may type-hint the `Illuminate\Http\Request` class to have the current HTTP request automatically injected into your route callback:

```
use Illuminate\Http\Request;

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

CSRF Protection

Remember, any HTML forms pointing to `POST`, `PUT`, `PATCH`, or `DELETE` routes that are defined in the `web` routes file should include a CSRF token field. Otherwise, the request will be rejected. You can read more about CSRF protection in the CSRF documentation:

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

Redirect Routes

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

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

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

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

Or, you may use the `Route::permanentRedirect` method to return a `301` status code:

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

> [!WARNING] When using route parameters in redirect routes, the following parameters are reserved by Laravel and cannot be used: `destination` and `status`.

View Routes

If your route only needs to return a view, you may use the `Route::view` method. Like the `redirect` method, this method provides a simple shortcut so that you do not have to define a full route or controller. The `view` method accepts a URI as its first argument and a view name as its second argument. In addition, you may provide an array of data to pass to the view as an optional third argument:

```
Route::view('/welcome', 'welcome');

Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

> [!WARNING] When using route parameters in view routes, the following parameters are reserved by Laravel and cannot be used: `view`, `data`, `status`, and `headers`.

Listing Your Routes

The `route:list` Artisan command can easily provide an overview of all of the routes that are defined by your application:

```
php artisan route:list
```

By default, the route middleware that are assigned to each route will not be displayed in the `route:list` output; however, you can instruct Laravel to display the route middleware and middleware group names by adding the `-v` option to the command:

```
php artisan route:list -v

# Expand middleware groups...
php artisan route:list -vv
```

You may also instruct Laravel to only show routes that begin with a given URI:

```
php artisan route:list --path=api
```

In addition, you may instruct Laravel to hide any routes that are defined by third-party packages by providing the `--except-vendor` option when executing the `route:list` command:

```
php artisan route:list --except-vendor
```

Likewise, you may also instruct Laravel to only show routes that are defined by third-party packages by providing the `--only-vendor` option when executing the `route:list` command:

```
php artisan route:list --only-vendor
```

Routing Customization

By default, your application's routes are configured and loaded by the `bootstrap/app.php` file:

```
<?php

use Illuminate\Foundation\Application;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )->create();
```

However, sometimes you may want to define an entirely new file to contain a subset of your application's routes. To accomplish this, you may provide a `then` closure to the `withRouting` method. Within this closure, you may register any additional routes that are necessary for your application:

```
use Illuminate\Support\Facades\Route;

->withRouting(
    web: __DIR__.'/../routes/web.php',
    commands: __DIR__.'/../routes/console.php',
    health: '/up',
    then: function () {
        Route::middleware('api')
            ->prefix('webhooks')
            ->name('webhooks.')
            ->group(base_path('routes/webhooks.php'));
    },
)
```

Or, you may even take complete control over route registration by providing a `using` closure to the `withRouting` method. When this argument is passed, no HTTP routes will be registered by the framework and you are responsible for manually registering all routes:

```
use Illuminate\Support\Facades\Route;

->withRouting(
    commands: __DIR__.'/../routes/console.php',
    using: function () {
        Route::middleware('api')
            ->prefix('api')
            ->group(base_path('routes/api.php'));

        Route::middleware('web')
            ->group(base_path('routes/web.php'));
    },
)
```

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 (string $id) {
    return 'User '.$id;
});
```

You may define as many route parameters as required by your route:

```
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string
$commentId) {
    // ...
});
```

Route parameters are always encased within `{}` braces and should consist of alphabetic characters. Underscores (_) are also acceptable within route parameter names. Route parameters are injected into route callbacks / controllers based on their order - the names of the route callback / controller arguments do not matter.

Parameters and Dependency Injection

If your route has dependencies that you would like the Laravel service container to automatically inject into your route's callback, you should list your route parameters after your dependencies:

```
use Illuminate\Http\Request;

Route::get('/user/{id}', function (Request $request, string $id) {
    return 'User '.$id;
});
```

Optional Parameters

Occasionally you may need to specify a route parameter that may not always be present in the URI. You may do so by placing a `?` mark after the parameter name. Make sure to give the route's corresponding variable a default value:

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

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

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

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

For convenience, some commonly used regular expression patterns have helper methods that allow you to quickly add pattern constraints to your routes:

```
Route::get('/user/{id}/{name}', function (string $id, string $name) {
    // ...
})->whereNumber('id')->whereAlpha('name');

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

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

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

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', ['movie', 'song', 'painting']);

Route::get('/category/{category}', function (string $category) {
    // ...
})->whereIn('category', CategoryEnum::cases());
```

If the incoming request does not match the route pattern constraints, a 404 HTTP response will be returned.

Global Constraints

If you would like a route parameter to always be constrained by a given regular expression, you may use the `pattern` method. You should define these patterns in the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use Illuminate\Support\Facades\Route;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Route::pattern('id', '[0-9]+');
}
```

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

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

Encoded Forward Slashes

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

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

> [!WARNING] Encoded forward slashes are only supported within the last route segment.

Named Routes

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

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

You may also specify route names for controller actions:

```
Route::get(
    '/user/profile',
    [UserProfileController::class, 'show']
)->name('profile');
```

> [!WARNING] Route names should always be unique.

Generating URLs to Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via Laravel's `route` and `redirect` helper functions:

```
// Generating URLs...
$url = route('profile');

// Generating Redirects...
return redirect()->route('profile');

return to_route('profile');
```

If the named route defines parameters, you may pass the parameters as the second argument to the `route` function. The given parameters will automatically be inserted into the generated URL in their correct positions:

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

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

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

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

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

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

> [!NOTE] Sometimes, you may wish to specify request-wide default values for URL parameters, such as the current locale. To accomplish this, you may use the `URL::defaults` method.

Inspecting the Current Route

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

```
use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

/**
 * Handle an incoming request.
 *
 * @param  \Closure(\Illuminate\Http\Request):
(\Symfony\Component\HttpFoundation\Response)  $next
 */
public function handle(Request $request, Closure $next): Response
{
    if ($request->route()->named('profile')) {
        // ...
    }

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

Route Groups

Route groups allow you to share route attributes, such as middleware, across a large number of routes without needing to define those attributes on each individual route.

Nested groups attempt to intelligently "merge" attributes with their parent group. Middleware and `where` conditions are merged while names and prefixes are appended. Namespace delimiters and slashes in URI prefixes are automatically added where appropriate.

Middleware

To assign middleware to all routes within a group, you may use the `middleware` method before defining the group. Middleware are executed in the order they are listed in the array:

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

    Route::get('/user/profile', function () {
        // Uses first & second middleware...
    });
});
```

Controllers

If a group of routes all utilize the same controller, you may use the `controller` method to define the common controller for all of the routes within the group. Then, when defining the routes, you only need to provide the controller method that they invoke:

```
use App\Http\Controllers\OrderController;

Route::controller(OrderController::class)->group(function () {
    Route::get('/orders/{id}', 'show');
    Route::post('/orders', 'store');
});
```

Subdomain Routing

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

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

> [!WARNING] 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 the names of all of the routes in the group with `admin`. The given string is prefixed to the route name exactly as it is specified, so we will be sure to provide the trailing `.` character in the prefix:

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

Route Model Binding

When injecting a model ID to a route or controller action, you will often query the database to retrieve the model that corresponds to that ID. Laravel route model binding provides a convenient way to automatically inject the model instances directly into your routes. For example, instead of injecting a user's ID, you can inject the entire `User` model instance that matches the given ID.

Implicit Binding

Laravel automatically resolves Eloquent models defined in routes or controller actions whose type-hinted variable names match a route segment name. For example:

```
use App\Models\User;

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

Since the `$user` variable is type-hinted as the `App\Models\User` Eloquent model and the variable name matches the `{user}` URI segment, Laravel will automatically inject the model instance that has an ID matching the corresponding value from the request URI. If a matching model instance is not found in the database, a 404 HTTP response will automatically be generated.

Of course, implicit binding is also possible when using controller methods. Again, note the `{user}` URI segment matches the `$user` variable in the controller which contains an `App\Models\User` type-hint:

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

// Route definition...
Route::get('/users/{user}', [UserController::class, 'show']);

// Controller method definition...
public function show(User $user)
{
    return view('user.profile', ['user' => $user]);
}
```

Soft Deleted Models

Typically, implicit model binding will not retrieve models that have been soft deleted. However, you may instruct the implicit binding to retrieve these models by chaining the `withTrashed` method onto your route's definition:

```
use App\Models\User;

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

Customizing the Key

Sometimes you may wish to resolve Eloquent models using a column other than `id`. To do so, you may specify the column in the route parameter definition:

```
use App\Models\Post;

Route::get('/posts/{post:slug}', function (Post $post) {
    return $post;
});
```

If you would like model binding to always use a database column other than `id` when retrieving a given model class, you may override the `getRouteKeyName` method on the Eloquent model:

```
/**
 * Get the route key for the model.
 */
public function getRouteKeyName(): string
{
    return 'slug';
}
```

Custom Keys and Scoping

When implicitly binding multiple Eloquent models in a single route definition, you may wish to scope the second Eloquent model such that it must be a child of the previous Eloquent model. For example, consider this route definition that retrieves a blog post by slug for a specific user:

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

Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
});
```

When using a custom keyed implicit binding as a nested route parameter, Laravel will automatically scope the query to retrieve the nested model by its parent using conventions to guess the relationship name on the parent. In this case, it will be assumed that the `User` model has a relationship named `posts` (the plural form of the route parameter name) which can be used to retrieve the `Post` model.

If you wish, you may instruct Laravel to scope "child" bindings even when a custom key is not provided. To do so, you may invoke the `scopeBindings` method when defining your route:

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

Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
    return $post;
})->scopeBindings();
```

Or, you may instruct an entire group of route definitions to use scoped bindings:

```
Route::scopeBindings()->group(function () {
    Route::get('/users/{user}/posts/{post}', function (User $user, Post $post) {
        return $post;
    });
});
```

Similarly, you may explicitly instruct Laravel to not scope bindings by invoking the `withoutScopedBindings` method:

```
Route::get('/users/{user}/posts/{post:slug}', function (User $user, Post $post) {
    return $post;
})->withoutScopedBindings();
```

Customizing Missing Model Behavior

Typically, a 404 HTTP response will be generated if an implicitly bound model is not found. However, you may customize this behavior by calling the `missing` method when defining your route. The `missing` method accepts a closure that will be invoked if an implicitly bound model cannot be found:

```
use App\Http\Controllers\LocationsController;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Redirect;

Route::get('/locations/{location:slug}', [LocationsController::class, 'show'])
    ->name('locations.view')
    ->missing(function (Request $request) {
        return Redirect::route('locations.index');
    });
```

Implicit Enum Binding

PHP 8.1 introduced support for Enums. To complement this feature, Laravel allows you to type-hint a string-backed Enum on your route definition and Laravel will only invoke the route if that route segment corresponds to a valid Enum value. Otherwise, a 404 HTTP response will be returned automatically. For example, given the following Enum:

```
<?php

namespace App\Enums;

enum Category: string
{
    case Fruits = 'fruits';
    case People = 'people';
}
```

You may define a route that will only be invoked if the `{category}` route segment is `fruits` or `people`. Otherwise, Laravel will return a 404 HTTP response:

```
use App\Enums\Category;
use Illuminate\Support\Facades\Route;

Route::get('/categories/{category}', function (Category $category) {
    return $category->value;
});
```

Explicit Binding

You are not required to use Laravel's implicit, convention based model resolution in order to use model binding. You can also explicitly define how route parameters correspond to models. To register an explicit binding, use the router's `model` method to specify the class for a given parameter. You should define your explicit model bindings at the beginning of the `boot` method of your `AppServiceProvider` class:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Route::model('user', User::class);
}
```

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

```
use App\Models\User;

Route::get('/users/{user}', function (User $user) {
    // ...
});
```

Since we have bound all `{user}` parameters to the `App\Models\User` model, an instance of that class will be injected into the route. So, for example, a request to `users/1` will inject the `User` instance from the database which has an ID of `1`.

If a matching model instance is not found in the database, a 404 HTTP response will be automatically generated.

Customizing the Resolution Logic

If you wish to define your own model binding resolution logic, you may use the `Route::bind` method. The closure you pass to the `bind` method will receive the value of the URI segment and should return the instance of the class that should be injected into the route. Again, this customization should take place in the `boot` method of your application's `AppServiceProvider`:

```
use App\Models\User;
use Illuminate\Support\Facades\Route;

/**
 * Bootstrap any application services.
 */
public function boot(): void
{
    Route::bind('user', function (string $value) {
        return User::where('name', $value)->firstOrFail();
    });
}
```

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

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

If a route is utilizing implicit binding scoping, the `resolveChildRouteBinding` method will be used to resolve the child binding of the parent model:

```
/**
 * Retrieve the child model for a bound value.
 *
 * @param  string  $childType
 * @param  mixed  $value
 * @param  string|null  $field
 * @return \Illuminate\Database\Eloquent\Model|null
 */
public function resolveChildRouteBinding($childType, $value, $field)
{
    return parent::resolveChildRouteBinding($childType, $value, $field);
}
```

Fallback Routes

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

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

Rate Limiting

Defining Rate Limiters

Laravel includes powerful and customizable rate limiting services that you may utilize to restrict the amount of traffic for a given route or group of routes. To get started, you should define rate limiter configurations that meet your application's needs.

Rate limiters may be defined within the `boot` method of your application's `App\Providers\AppServiceProvider` class:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Bootstrap any application services.
 */
protected function boot(): void
{
    RateLimiter::for('api', function (Request $request) {
        return Limit::perMinute(60)->by($request->user()?->id ?: $request->ip());
    });
}
```

Rate limiters are defined using the `RateLimiter` facade's `for` method. The `for` method accepts a rate limiter name and a closure that returns the limit configuration that should apply to routes that are assigned to the rate limiter. Limit configuration are instances of the `Illuminate\Cache\RateLimiting\Limit` class. This class contains helpful "builder" methods so that you can quickly define your limit. The rate limiter name may be any string you wish:

```
use Illuminate\Cache\RateLimiting\Limit;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\RateLimiter;

/**
 * Bootstrap any application services.
 */
protected function boot(): void
{
    RateLimiter::for('global', function (Request $request) {
        return Limit::perMinute(1000);
    });
}
```

If the incoming request exceeds the specified rate limit, a response with a 429 HTTP status code will automatically be returned by Laravel. If you would like to define your own response that should be returned by a rate limit, you may use the `response` method:

```
RateLimiter::for('global', function (Request $request) {
    return Limit::perMinute(1000)->response(function (Request $request, array $headers) {
        return response('Custom response...', 429, $headers);
    });
});
```

Since rate limiter callbacks receive the incoming HTTP request instance, you may build the appropriate rate limit dynamically based on the incoming request or authenticated user:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
        ? Limit::none()
        : Limit::perMinute(100);
});
```

Segmenting Rate Limits

Sometimes you may wish to segment rate limits by some arbitrary value. For example, you may wish to allow users to access a given route 100 times per minute per IP address. To accomplish this, you may use the `by` method when building your rate limit:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()->vipCustomer()
        ? Limit::none()
        : Limit::perMinute(100)->by($request->ip());
});
```

To illustrate this feature using another example, we can limit access to the route to 100 times per minute per authenticated user ID or 10 times per minute per IP address for guests:

```
RateLimiter::for('uploads', function (Request $request) {
    return $request->user()
        ? Limit::perMinute(100)->by($request->user()->id)
        : Limit::perMinute(10)->by($request->ip());
});
```

Multiple Rate Limits

If needed, you may return an array of rate limits for a given rate limiter configuration. Each rate limit will be evaluated for the route based on the order they are placed within the array:

```
RateLimiter::for('login', function (Request $request) {
    return [
        Limit::perMinute(500),
        Limit::perMinute(3)->by($request->input('email')),
    ];
});
```

If you're assigning multiple rate limits segmented by identical `by` values, you should ensure that each `by` value is unique. The easiest way to achieve this is to prefix the values given to the `by` method:

```
RateLimiter::for('uploads', function (Request $request) {
    return [
        Limit::perMinute(10)->by('minute:'.$request->user()->id),
        Limit::perDay(1000)->by('day:'.$request->user()->id),
    ];
});
```

Attaching Rate Limiters to Routes

Rate limiters may be attached to routes or route groups using the `throttle` middleware. The throttle middleware accepts the name of the rate limiter you wish to assign to the route:

```
Route::middleware(['throttle:uploads'])->group(function () {
    Route::post('/audio', function () {
        // ...
    });

    Route::post('/video', function () {
        // ...
    });
});
```

Throttling With Redis

By default, the `throttle` middleware is mapped to the `Illuminate\Routing\Middleware\ThrottleRequests` class. However, if you are using Redis as your application's cache driver, you may wish to instruct Laravel to use Redis to manage rate limiting. To do so, you should use the `throttleWithRedis` method in your application's `bootstrap/app.php` file. This method maps the `throttle` middleware to the `Illuminate\Routing\Middleware\ThrottleRequestsWithRedis` middleware class:

```
->withMiddleware(function (Middleware $middleware) {
    $middleware->throttleWithRedis();
    // ...
})
```

Form Method Spoofing

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

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

For convenience, you may use the `@method` Blade directive to generate the `_method` input field:

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

Accessing the Current Route

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

```
use Illuminate\Support\Facades\Route;

$route = Route::current(); // Illuminate\Routing\Route
$name = Route::currentRouteName(); // string
$action = Route::currentRouteAction(); // string
```

You may refer to the API documentation for both the underlying class of the Route facade and Route instance to review all of the methods that are available on the router and route classes.

Cross-Origin Resource Sharing (CORS)

Laravel can automatically respond to CORS `OPTIONS` HTTP requests with values that you configure. The `OPTIONS` requests will automatically be handled by the `HandleCors` middleware that is automatically included in your application's global middleware stack.

Sometimes, you may need to customize the CORS configuration values for your application. You may do so by publishing the `cors` configuration file using the `config:publish` Artisan command:

```
php artisan config:publish cors
```

This command will place a `cors.php` configuration file within your application's `config` directory.

> [!NOTE] For more information on CORS and CORS headers, please consult the MDN web documentation on CORS.

Route Caching

When deploying your application to production, you should take advantage of Laravel's route cache. Using the route cache will drastically decrease the amount of time it takes to register all of your application's routes. To generate a route cache, execute the `route:cache` Artisan command:

```
php artisan route:cache
```

After running this command, your cached routes file will be loaded on every request. Remember, if you add any new routes you will need to generate a fresh route cache. Because of this, you should only run the `route:cache` command during your project's deployment.

You may use the `route:clear` command to clear the route cache:

```
php artisan route:clear
```