Routing

Routing is one of the most important code concepts of Laravel (and most likely almost any other web framework). Unlike normal 'static' websites (like your portfolio website), where each HTTP request is processed by the webserver trying to find a file that matches the specified URI, all the requests are processed by the framework itself.

The essentials

In Laravel the webserver is configured to direct all incoming requests to the single entry point of the application: the `public/index.php` file. The `index.php` file doesn't contain much code. Rather, it is a starting point for loading the rest of the framework.

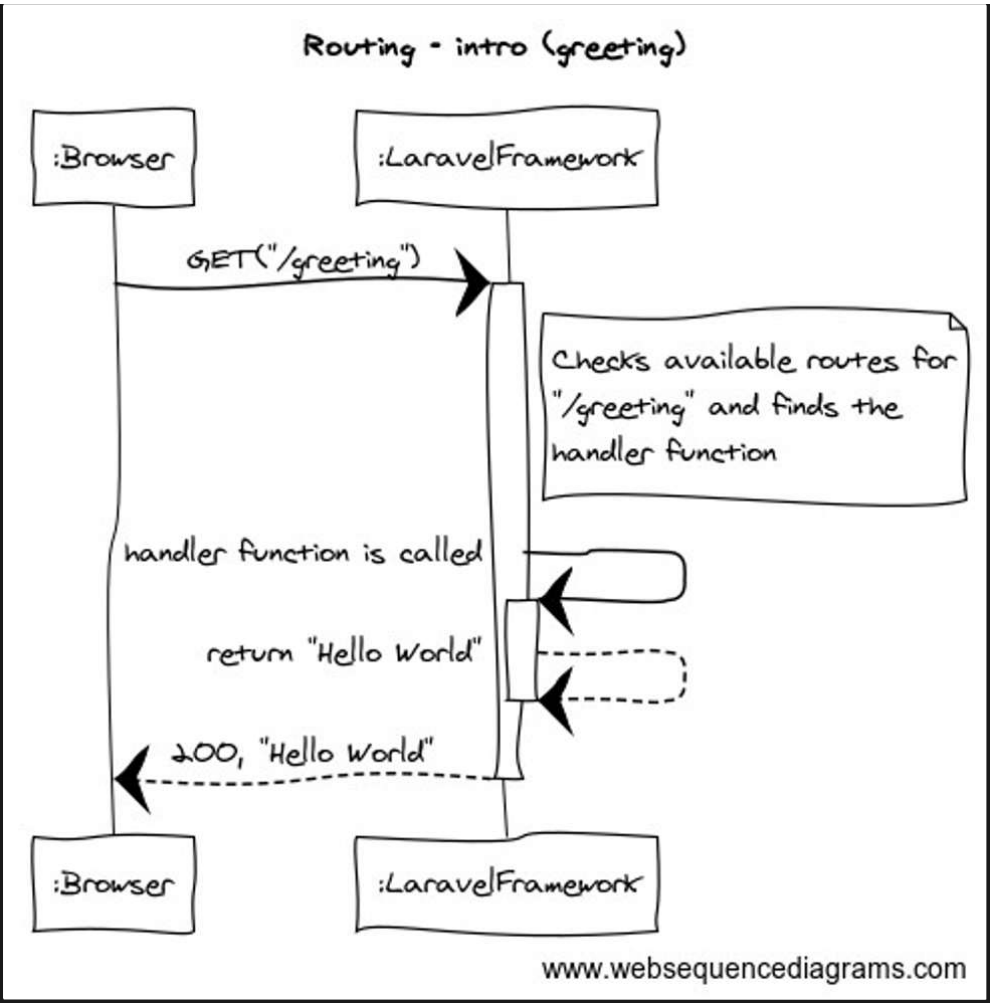
> ! You should NOT change the content of the `public/index.php` file in normal conditions (99.99999% of all the cases). Consult with a teacher first if you ever think you need to change this file!

Defining web routes

All Laravel routes are defined in route files, which are located in the `routes` directory. For most applications, you will begin by defining routes in your `routes/web.php` file. The other files in this folder are only needed for more advanced use. You should leave them as they are for now. The routes defined in `routes/web.php` may be accessed by entering the defined route's URL in your browser. For example, the content of (a part of) `routes/web.php` might look like:

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

This line of code defines a route for a GET request for the `/greeting` URI and declares an anonymous callback function that handles that request. This means that the application will now behave as shown in the following sequence diagram:



Explanation:

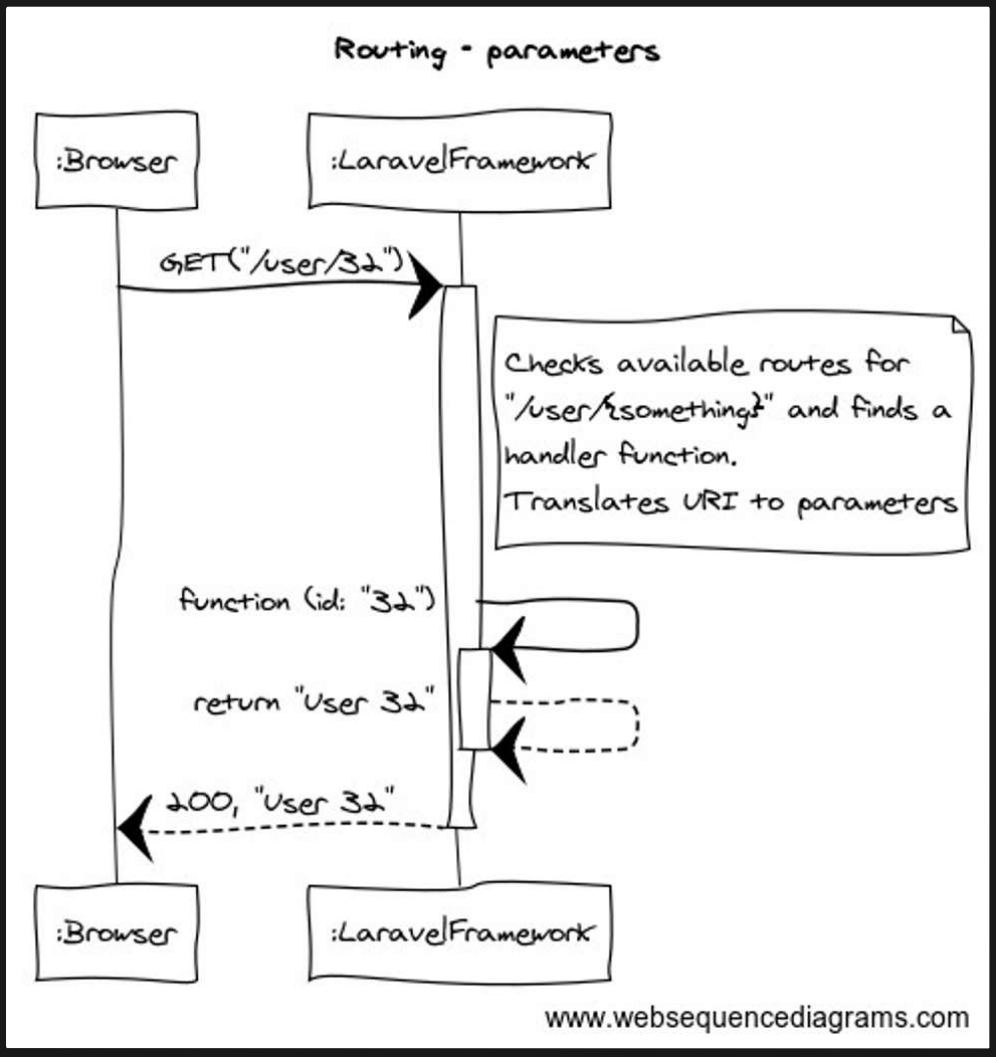
1. Some user uses his browser to request the content of the `/greeting` URI so the browser sends an HTTP GET request (to e.g. `http://example.com/greeting`)
2. The request is received by the Laravel application. It checks all the routes configured in `web.php` and finds the declared route handler function
3. The declared function is called. This function returns the string “Hello World”.
4. This string is wrapped in a 200 response by the Laravel application and sent back to the sender.

Route 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; });
```

Route parameters are always encased within `{}` braces and should consist of alphabetic characters. Notice that Laravel automatically implies that the callback function has a `$id` parameter and will try to pass the content of the URI that corresponds to tha parameters' location as a value of that parameter. This means that the application will now behave as follows:



Explanation:

1. Some user uses his browser to request the content of a certain user with the `/user/32` URI so the browser sends a HTTP GET request (to e.g. `http://example.com/user/32`)
2. The request is received by the Laravel application and finds the declared route callback function
3. The declared function is called, with string value `'32'` passed as the value of the `$id` parameter resulting the function returning the string "User 32".

4. This string is wrapped in a 200 response by the Laravel application and sent back to the sender.

Other router methods

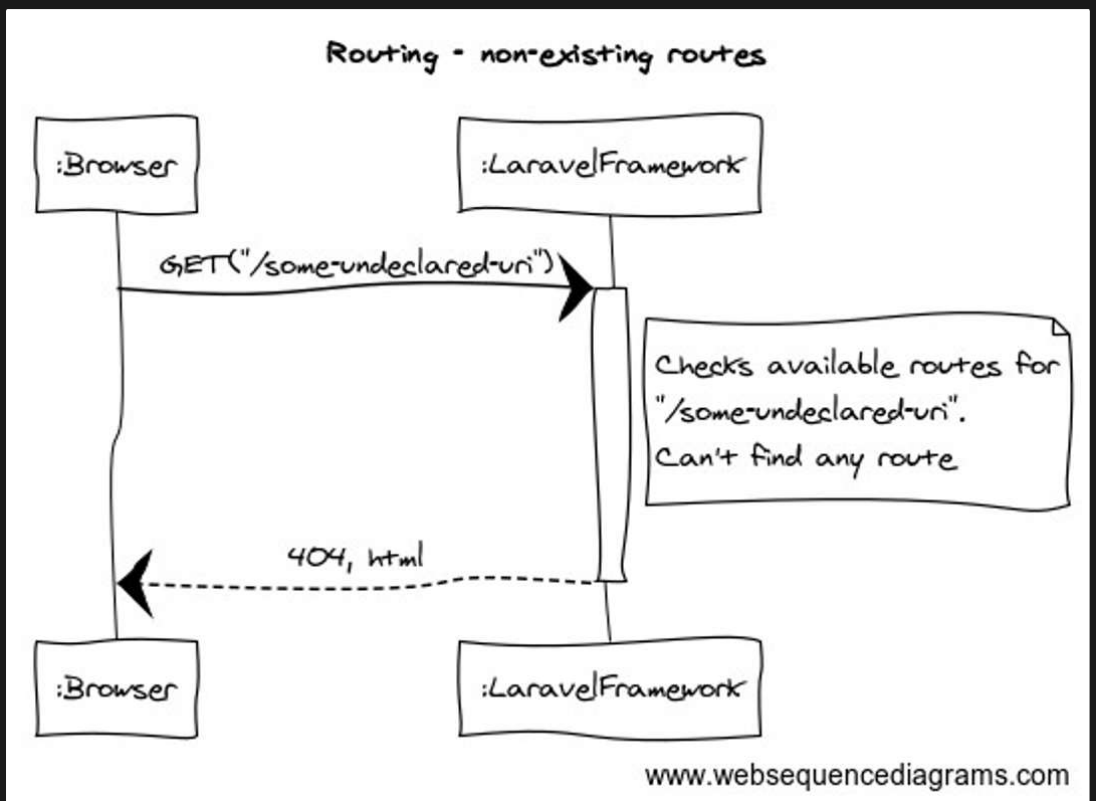
As you may already know, the HTTP protocol has, besides GET, other verbs. These verbs are also made available for routes:

```
Route::get($uri, $callback); Route::post($uri, $callback);
Route::put($uri, $callback); Route::patch($uri, $callback);
Route::delete($uri, $callback); Route::options($uri, $callback);
```

Handling non-existing routes

Users have total freedom. They can send any HTTP request to your application, either by accident or on purpose. They can accidentally type the wrong URL in their browser. On the other hand, you as the developer can provide incorrect navigation links, for instance in your menu. Even hackers can try different links to see if they can find some vulnerability they can exploit.

You application needs to be able to handle these non-existing routes properly. HTTP conventions say that it must respond with a 404 response. Typically, Laravel handles this as follows:



This works fine for URI's with mistakes and routes without parameters. When routes do have parameters, Laravel can't always decide for you whether or not this route exists for you. Consider for example the `/user/{id}` route. Let's say that our application context has 1500 users and therefore users with an id > 1500 simply doesn't exist. This means that when a user calls this route with `/user/532` it must return a 200 and with `/user/65536` it must return a 400.

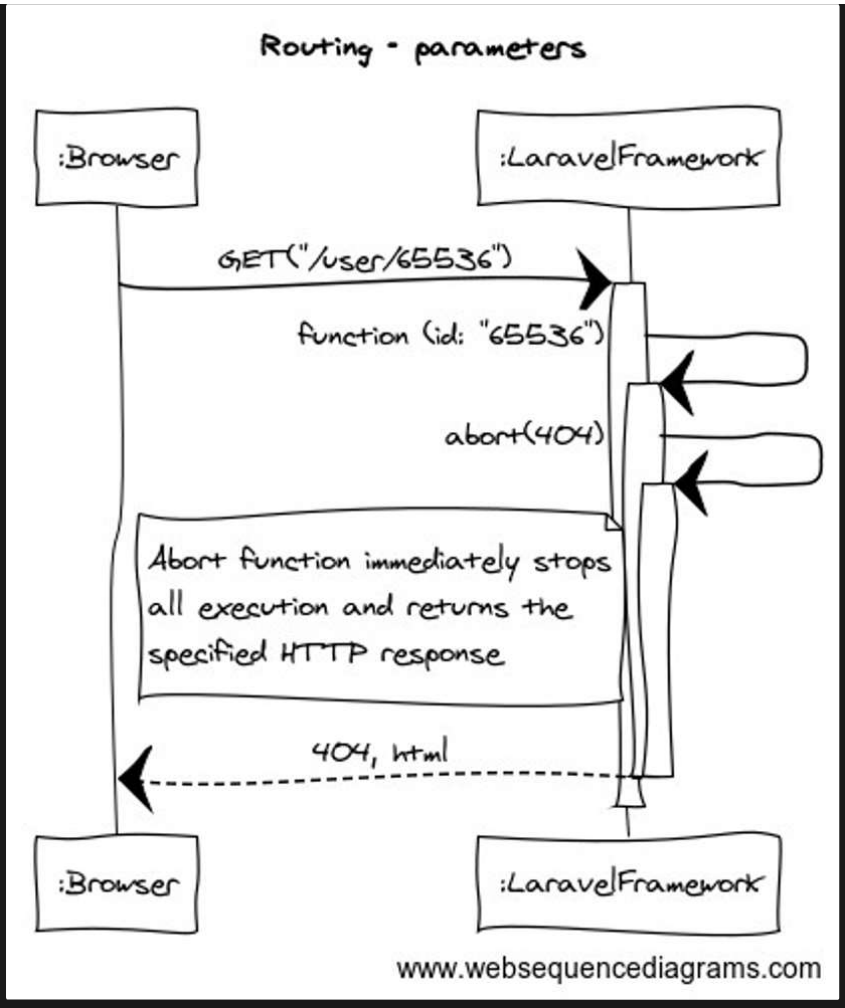
This is something Laravel can't solve for you. You must check this in your route handler yourself. Luckily, Laravel can make it easier for you with the `abort()` helper function, for instance:

```
Route::get('/user/{id}', function (string $id) { if ($id > 1500) { // some arbitrary test to demonstrate a check if a user exists abort(404); } return 'User '.$id; });
```

Explanation

1. The user sends an HTTP request `http://example.com/user/65536`
2. Laravel finds an appropriate handler function and calls it with `$id = "65536"`
3. The handler function determines that the `$id` is larger than 1500 and calls `abort()` with 404
4. The `abort()` makes Laravel to stop all other execution and returns a HTTP response with status code 404 and html that represents a page that explains the error

In a sequence diagram:



Controllers

When applications evolve, the amount of routes may grow and grow, and therefore the content of the `routes/web.php` file. There might be a moment when the content becomes too large to be maintainable. Laravel uses Controller classes to solve this. You can group related request handling logic into a single Controller class. For example, a `UserController` class might handle all incoming requests related to users, including showing, creating, updating, and deleting users. By default, controllers are stored in the `app/Http/Controllers` directory.

To quickly generate a new controller, you may run the `make:controller` Artisan command:

```
php artisan make:controller UserController
```

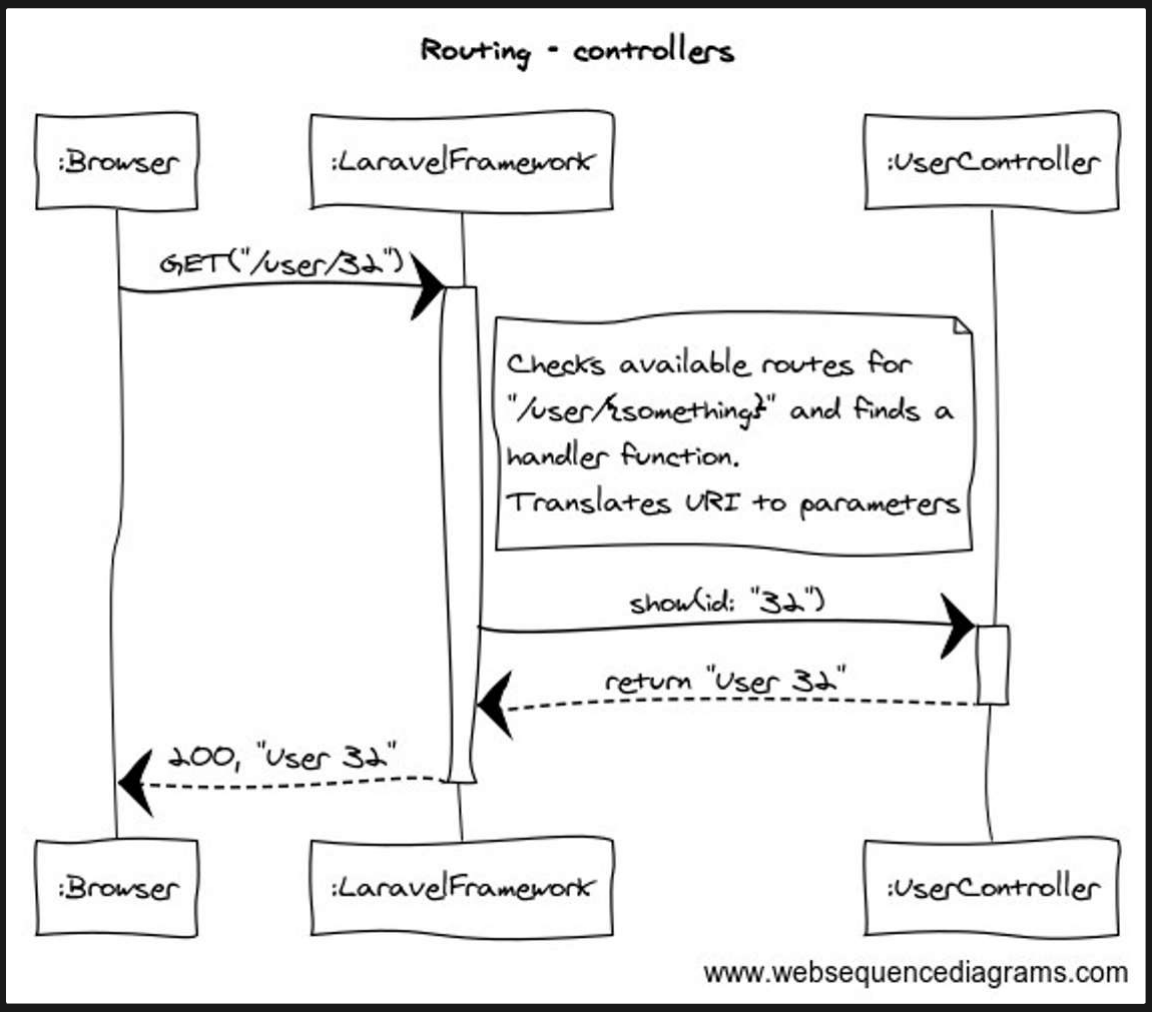
The generated controller can then be edited by, for example, add a method:

```
<?php namespace App\Http\Controllers; use App\Models\User; use Illuminate\View\View; class UserController extends Controller { /** * Show the profile for a given user. */ public function show(string $id): View { return 'User '.$id; } }
```

Once you have written a controller class and method, you may define a route to the controller method in the `routes/web.php` file like so:

```
Route::get('/user/{id}', [UserController::class, 'show']);
```

This means that the application will now behave as follows:



1. Some user uses his browser to request the content of a certain user with the `/user/32` URI so the browser sends a HTTP GET request (to e.g. `http://example.com/user/32`)
2. The request is recieved by the Laravel application and finds the declared route

3. It instantiates a `UserController` object and invokes the `show` method with string value `'32'` passed as the value of the `$id` parameter
4. The declared function is executed and returns the string “User 32”.
5. This string is wrapped in a 200 response by the Laravel application and sent back to the sender.

> 💡 Because applications almost always grow, it is considered a best practice, from now on, to NEVER use these anonymous callback functions again. Using controllers should be the normal way to go. They have even more advantages as you might learn later on in this course.

Route List Tables

When developing an application based on Laravel you need to think about how to organize your route handling. You can do this by creating a **route list table** for this, like:

URI	Handler	View name	Route name (optional)
`'/foo'`	`BarController@gaz`	`'vag'`	`prizo`
...			

The **URI** is the section of the URL that must lead to this specific route

The **handler** is either a:

- Closure (but know that this is considered a bad practice when developing real applications)
- Controller method (showed as `ControllerClassName@methodName`),
- Single Action Controller name (showed as `ControllerClassName`)
- View route (showed by the exact words ‘View Route’)

The **view name** is the name of the blade file, or more specifically, the parameter you must add to the `return view()` statement

The optional **route name** is the name you will use if you want to use Named Routes

Digging deeper

The route list

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
```

Multiple route parameters

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

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

Route groups

Route groups allow you to share route attributes across a large number of routes without needing to define those attributes on each individual route. One specific attribute that you might be able to use is a *prefix*.

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::get('/foobar', function () { // Matches The "/admin/foobar" URL }); });
```

Named routes

Named routes allow the convenient generation of URLs or redirects for specific routes. This makes it possible to declare the actual URL of a route only once and refer to it from anywhere else in your project by its name. You may specify a name for a route by chaining the `name` method onto the route definition:

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

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');
```

Dependency injection

When you write a route in Laravel, you can mention what other things your route needs right in the route's code. Laravel's magic box (called the service container) then gives - *injects* - those things to your route automatically. This is called *dependency injection*.

For instance, if your route needs information about the current web request, you can just mention that in your route's code, and Laravel will make sure your route gets that information without you having to do extra work.

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

Here the `Illuminate\Http\Request` class is type-hinted to have the current HTTP request automatically injected into your route callback.

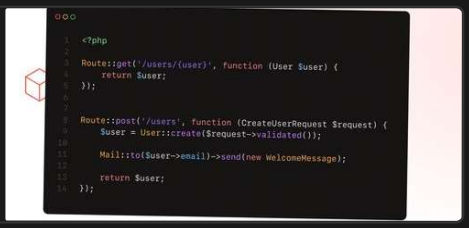
Route Model Binding

When (Eloquent) models are involved in a route handler, it is very common that the model's ID is passed as a route parameter and that the database is queried to fetch the corresponding model. Laravel's *route model binding* offers a convenient method to automatically inject these model instances directly into your routes. For instance, rather than injecting just a user's ID, you can inject the complete User model instance that matches the provided ID. Laravel automatically resolves Eloquent models defined in routes or controller actions whose type-hinted variable names match a route segment name. For example:

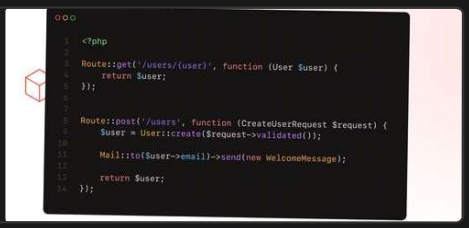
```
use App\Models\User; Route::get('/users/{user}', function (User $user) {
return $user->email; });
```

Further reading

You can read all about routing in the Laravel documentation here:



Controllers are explained here:



The following topics in these pages are particularly noteworthy:

Controller route groups

If a group of routes all utilize the same controller, you may use the `controller`