

AN OPENSOURCE EBOOK

Laravel



Tips and Tricks

Bobby Iliev

Table of Contents

About the book	15
About the author	16
Sponsors	17
Ebook PDF Generation Tool	19
Book Cover	20
License	21
How to Install Laravel on DigitalOcean with 1-Click?	22
Installing Laravel on DigitalOcean	23
Using the Laravel DigitalOcean 1-Click	24
Completing the Laravel configuration	26
Video Demo	28
Conclusion	29
How to get a free domain name for your Laravel project	30
Choosing a free domain name	31
Adding your Domain to DigitalOcean	34
Updating your Nameservers	35
Conclusion	37
8 Awesome VS Code Extensions for Laravel Developers	38
1. Laravel Blade Snippets	39
2. Laravel Snippets	41
3. Laravel Blade Spacer	43
4. Laravel Artisan	44
5. Laravel Extra Intellisense	46

6. Laravel goto Controller	48
7. Laravel goto View	49
8. DotENV syntax highlighting	50
Book recommendation	51
Conclusion	52
What is Laravel Jetstream and how to get started	53
What is Laravel Jetstream	54
Installing Laravel Jetstream	55
Authentication	57
Profile management	59
Jetstream Security	61
Jetstream API	62
Jetstream Teams	63
Conclusion	64
How to check Laravel Blade View Syntax using artisan	65
Installation	66
Testing	67
Conclusion	68
How to speed up your Laravel application with PHP OPcache ..	69
Enable PHP OPcache	70
Configure PHP OPcache	72
Configure Laravel OPCache	74
Conclusion	76
What is Laravel Sail and how to get started	77
Installing And Setting Up Laravel Sail	79

Laravel Sail Commands	80
Video Introduction	82
Conclusion	83
How to add simple search to your Laravel blog/website	84
Controller changes	85
Route changes	86
Blade view changes	87
Conclusion	88
How to Create Custom Laravel Maintenance Page	89
Enable Default Maintenance Mode	90
Create Custom Maintenance Page	92
Disable Maintenance Mode	94
Conclusion	95
What is Laravel Blade UI Kit and how to get started	96
Installation	97
Configuration	98
Example	101
Conclusion	103
How to Add a Simple Infinite Scroll Pagination to Laravel	104
Configuring your Controller	106
Configuring your Blade View	107
Adding and configuring jScroll	110
Conclusion	112
How to add a simple RSS feed to Laravel without using a package	113

Configuring RSS Controller	115
Configuring your Blade view	117
Configuring your Route	119
Conclusion	120
What is Laravel Zero and how to get started	121
Installation	122
Commands	125
Configuration	127
Addons	129
Building the application	132
Conclusion	133
How to build a blog with Laravel and Wink	134
Installation	135
Creating a Controller	137
Creating Views	139
Adding routes	143
Conclusion	144
How to copy or move records from one table to another in Laravel	145
Copy all records from one table to another	146
Move all records from one table to another	148
Conclusion	149
How to generate title slugs in Laravel	150
Creating a slug in Laravel	151
Conclusion	152

What is Laravel Enlighthn and how to use it	153
Installation	155
Configuration	156
Usage	157
Analyzing the output	159
Conclusion	161
How to consume an external API with Laravel and Guzzle ...	162
Creating a new table	163
Create the Model	165
Create the Controller	166
QuizAPI overview	167
Building the method	169
Add the route	172
Conclusion	173
How to send Discord notifications with Laravel	174
Creating a Discord Channel and Webhook	176
Installing the http-client	177
Adding the Controller	178
Adding the Routes	181
Conclusion	182
How to encrypt and decrypt a string in Laravel	183
App Key Configuration	185
Creating a Route	186
Creating a Controller	187
Conclusion	190

How to remove a package from Laravel using composer	191
Adding a package to Laravel	192
Removing a package from Laravel using composer	193
Conclusion	194
 What is Laravel Breeze and how to get started	 195
Overview	197
Installation	198
File structure	200
Video Overview	202
Conclusion	203
 What are signed routes in Laravel and how to use them	 204
Defining a signed route	206
Generating the signature	207
Using the signed route	208
Conclusion	210
 How to Quickly Change the Password for a User in Laravel ..	
211	
Reset password with tinker	212
Reset password via a route	215
Conclusion	216
 How to convert markdown to HTML in Laravel and Voyager ..	
217	
Install the Laravel Markdown Package	219
Configure your Controller	220
Change the Input type in Voyager	222

Conclusion	223
How to Create Response Macros in Laravel	224
Creating a service provider	225
Creating the Response Macro	226
Using the Response Macro	227
Conclusion	228
How to Get the Base URL in Laravel	229
Access the Laravel Base URL	231
Access the current URL in Laravel	232
Named routes	233
Access Assets URLs	234
Conclusion	235
How to limit the length of a string in Laravel	236
Limit string length in Blade	237
Limit string length in Model	238
Limit string length in Controller	239
Conclusion	240
How to check 'if not null' with Laravel Eloquent	241
Check if null: whereNull	242
Check if not null: whereNotNull	243
Conclusion	244
How to get the current date and time in Laravel	245
Using Carbon to get the current date and time	246
Using the now() helper function	249
Conclusion	250

How to Get Current Route Name in Laravel	251
Get route name in Controller or Middleware	252
Get route name in Blade View	254
Getting additional information about your route	255
Conclusion	256
 How to Count and Detect Empty Laravel Eloquent Collections ..	
257	
Check if a collection is empty	259
Check if a collection is not empty	260
Counting the records of a collection	261
Conclusion	262
 How to use Forelse loop in Laravel Blade	263
Check if not empty then	264
Forelse example	265
Conclusion	266
 How to Delete All Entries in a Table Using Laravel Eloquent ..	
267	
Using truncate to delete all entries from a table	269
Delete all entries using the delete() method	270
Conclusion	271
 How to check if a record exists with Laravel Eloquent	272
Check if a record exists	273
Create record if it does not exist already	274
Conclusion	276

How to Add Multiple Where Clauses Using Laravel Eloquent ..

277

Chaining multiple where clauses together	278
Passning an array to the where method	279
Using orWhere Clauses	280
Conclusion	281

How to Add a New Column to an Existing Table in a Laravel

Migration 282

Creating a new table with a migration 283

Add a New Column to an Existing Table 286

Conclusion 289

How to Rollback Database Migrations in Laravel 290

Rollback the Last Database Migration	291
Rollback Specific Migration	292
Rollback All Migrations	293
Conclusion	294

How to Remove a Migration in Laravel 295

Creating a Laravel migration	296
Remove a Migration in Laravel	297
Conclusion	299

How to create a contact form with Laravel Livewire 300

Installing Livewire	301
---------------------------	-----

Adding new Livewire component	302
Creating your Blade view	303
Prepare your Livewire view	305
Prepare your Livewire Logic	308
Add routes	311
Adding SMTP details	312
Conclusion	313
How To Display HTML Tags In Blade With Laravel	314
Display HTML In Laravel Blade Views	315
Conclusion	317
How to Set a Variable in Laravel Blade Template	318
Returning a variable from a controller	319
Setting variables in Laravel Blade Template	320
Conclusion	321
How to limit the result with Laravel Eloquent	322
Limiting the result with pure SQL	324
The limit() method	325
The take() method	326
The paginate() method	327
Conclusion	328
How to Select Specific Columns in Laravel Eloquent	329
Select specific columns with SQL only	330
Select specific columns with Laravel Eloquent	331
Conclusion	332
How to get the Laravel Query Builder to Output the Raw SQL	

Query	333
The toSql() method	334
Laravel Debugbar	336
Conclusion	337
 How to Get the Last Inserted Id Using Laravel Eloquent	 338
Last Inserted Id Using Laravel Eloquent	339
Conclusion	341
 How to Order the Results of all() in Laravel Eloquent	 342
Ordering the results on a query level	344
Ordering the results on a collection level with all()	346
Conclusion	348
 How to fix Laravel Unknown Column 'updated_at'	 349
Disable the timestamp columns in your Model	351
Change the name of the timestamp tables	352
Conclusion	353
 How to Define Custom ENV Variables in Laravel	 354
Defining a variable in .env	355
Accessing the env variable with env()	356
Accessing the env variable with config()	357
Conclusion	359
 How to fix 'Please provide a valid cache path' error in Laravel	 360
Creating the cache folders	361
Clearing the cache	362
Conclusion	363

How to check your exact Laravel version	364
Check your Laravel version with artisan	365
Check your Laravel version via your text editor	366
Conclusion	367
 Contact Form with Voyager and Laravel	 368
Steps	369
Configuration	370
Conclusion	377
 How to filter routes in Laravel	 378
Steps	379
Filter routes by term	380
Conclusion	382
 How to add a gravatar profile picture in Laravel	 383
Steps	384
Create a trait to handle the gravatar	385
Render an image in a blade view:	387
Use gravatars with Jetstream	388
Conclusion	389
 How to append a mutator to a collection	 390
Append a mutator on a single model	391
Usage	392
Append a mutator to a collection	393
Conclusion	394
 How to use env variables in javascript	 395

An example	396
Conclusion	397
How to customize forgot password notification	398
Create a custom notification	399
Override the notification	400
Conclusion	401
How to fire an event when user is logged in	402
Create a custom service provider	403
Conclusion	404
Scaling Laravel App with Multiple Databases	405
Prerequisites	406
Steps	407
1. Deploy 3 servers on Digital Ocean	408
2. Install and Configure our Web server	409
Install and configure our database servers along with our MySQL replication	413
Configure Laravel to work with the Master-Slave MySQL setup	416
Install the Voyager admin panel on the new setup	418
Conclusion	420
Conclusion	422
Other eBooks	423

About the book

- **This version was published on July 25 2021**

This is an open-source **Laravel Tips and Tricks eBook** that is a collection of my own notes that I've put together for myself throughout the years. You would more likely than not need many of those tips at some point in your career as a Laravel Developer.

The guide is suitable for anyone working as a Laravel developer and would love to learn some random Laravel tips and tricks. You can read the chapters in this book in a random order as they are completely separate tips or tricks.

About the author

My name is Bobby Iliev, and I have been working as a Linux DevOps Engineer since 2014. I am an avid Linux lover and supporter of the open-source movement philosophy. I am always doing that which I cannot do in order that I may learn how to do it, and I believe in sharing knowledge.

I think it's essential always to keep professional and surround yourself with good people, work hard, and be nice to everyone. You have to perform at a consistently higher level than others. That's the mark of a true professional.

For more information, please visit my blog at <https://bobbyiliev.com>, follow me on Twitter [@bobbyiliev_](#) and [YouTube](#).

Sponsors

This book is made possible thanks to these fantastic companies!

DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale.

It provides highly available, secure, and scalable compute, storage, and networking solutions that help developers build great software faster.

Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available.

For more information, please visit <https://www.digitalocean.com> or follow [@digitalocean](#) on Twitter.

If you are new to DigitalOcean, you can get a free \$100 credit and spin up your own servers via this referral link here:

[Free \\$100 Credit For DigitalOcean](#)

DevDojo

The DevDojo is a resource to learn all things web development and web design. Learn on your lunch break or wake up and enjoy a cup of coffee with us to learn something new.

Join this developer community, and we can all learn together, build together, and grow together.

Join DevDojo

For more information, please visit <https://www.devdojo.com> or follow [@thedeveloper](#) on Twitter.

Ebook PDF Generation Tool

This ebook was generated by [Ibis](#) developed by [Mohamed Said](#).

Ibis is a PHP tool that helps you write eBooks in markdown.

Book Cover

The cover for this ebook was created with [Canva.com](https://www.canva.com).

If you ever need to create a graphic, poster, invitation, logo, presentation – or anything that looks good — give Canva a go.

License

MIT License

Copyright (c) 2020 Bobby Iliev

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

How to Install Laravel on DigitalOcean with 1-Click?

[Laravel](#) is an amazing PHP framework that makes working with PHP great.

As of the time of writing this tutorial, Laravel has more than 105 million installs according to [Packagist](#), so as you can imagine there are multiple ways of **installing Laravel on DigitalOcean** or any other cloud provider.

In this tutorial, we will go through a few ways of installing Laravel and also show you how to do this with just 1-Click on DigitalOcean!

All you need in order to follow along is a DigitalOcean account. To make things even better you can use the following referral link to get **free \$100 credit** that you could use to deploy your servers and test the guide yourself:

[DigitalOcean \\$100 Free Credit](#)

Installing Laravel on DigitalOcean

If you wanted to configure your LEMP server from scratch you could, for example, do it manually by following this amazing step by step guide provided by DigitalOcean:

- [How to Install and Configure Laravel with LEMP on Ubuntu 18.04](#)

Another option of automating the above steps is to use the [LaraSail](#) open-source script to set up your server and install Laravel:

- <https://github.com/thedevdojo/larasail>

Using the Laravel DigitalOcean 1-Click

DigitalOcean has a Marketplace, where you could get a lot of various 1-Click Applications which you can deploy on your servers and Kubernetes clusters.

Recently we built a Laravel image and submitted a listing to DigitalOcean. The image is now available on the DigitalOcean Marketplace:

<https://marketplace.digitalocean.com/apps/laravel>

The image comes with a preconfigured LEMP stack and certbot installed. The software stack that comes with the image is:

- [Laravel](#) 7.20.0
- [Nginx](#) 1.18.0
- [MySQL server](#) 8.0.20
- [PHP](#) 7.4.3
- [Certbot](#) 0.40.0
- [Composer](#) 1.10.1

In order to use the image just go to the [1-Click Laravel application page](#) and hit the **Create Laravel Droplet** button:



After that just follow the standard flow and choose the details that match your needs like:

- The size of the Droplet
- Choose a datacenter region
- Choose your SSH keys
- Choose hostname
- I would recommend enabling backups as well

Finally hit the **Create Droplet** button. Then in around 30 seconds or so your Laravel server will be up and running!

Completing the Laravel configuration

To complete the installation, copy your IP address and [SSH to the Droplet](#).

You would get to an interactive menu that would ask you for the following details:

```
Enter the domain name for your new Laravel site.  
(ex. example.org or test.example.org) do not include www or  
http/s  
-----  
Domain/Subdomain name:
```

There just type your domain name or subdomain name, I would go for `laravel.bobbyiliev.com` for my example.

After that you would see the following output:

```
Configuring Laravel database details  
Generating new Laravel App Key  
Application key set successfully.
```

Then you will be asked if you want to secure your Laravel installation with an SSL certificate:

Next, you have the option of configuring LetsEncrypt to secure your **new** site.

Before doing this, be sure that you have pointed your domain **or** subdomain to this server's **IP address**.

You can also run LetsEncrypt certbot later with the command **'certbot --nginx'**

**Would you like to use LetsEncrypt (certbot)
to configure SSL(https) for your new site? (y/n):**

Note that before hitting **y** make sure to have your domain name or subdomain DNS configured so that your A record points to the Droplet's IP address, otherwise Let's Encrypt will not be able to validate your domain and would not issue a certificate for you.

If you don't want a certificate, just type **n** and hit enter.

That is pretty much it! Now if you visit **yourdomain.com** in your browser you would see a fresh new installation!

Video Demo

Here's also a quick video demo on how to do the above:

[How to Install Laravel on DigitalOcean with 1-Click](#)

Conclusion

Thanks to the Laravel 1-Click installation from the DigitalOcean's Marketplace, we can have a fully running LEMP server with Laravel installed in less than 30 seconds!

[Originally posted here.](#)

How to get a free domain name for your Laravel project

There could be various reasons why you would need a free domain for your project.

For example, having multiple side projects could be quite costly in case that your projects are not generating any income. So saving costs could be crucial.

Another reason why you might need a free domain name is that you might want to do some just testing and not really need an actual domain which could cost you anywhere from \$5 to +\$50 dollars depending on the provider and the domain extension.

In this tutorial, I will show you **how to get a free domain name** from [Freenom](#) and use it with your Laravel Project.

Choosing a free domain name

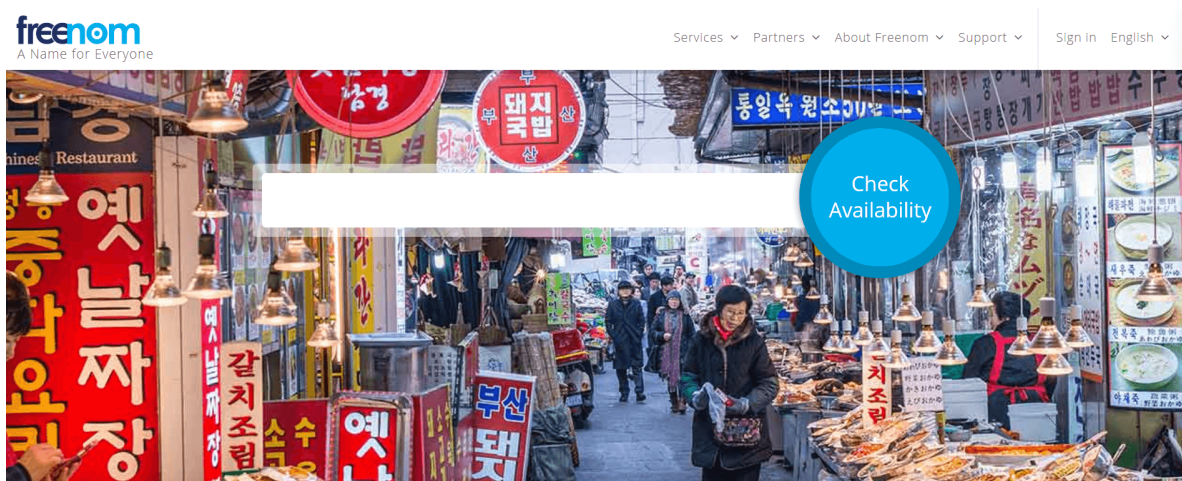
In this tutorial, we will use [Freenom](#) to register a free domain name! As far as I am aware, Freenom is the world's only free domain name provider.

They provide free domain names with the following domain extensions:

- .TK
- .ML
- .GA
- .CF
- .GQ

Unfortunately, Freenom does not offer free **.com** domains but for testing purposes, only the above domain extensions are a great alternative!

To check if a specific domain is available, just visit the [Freenom](#) website, you would get to a page where you can search and check if a specific domain is available:



In my case I will look for **bobbyiliev** and then choose the available extension:

bobbyiliev .tk	• FREE	EUR 0. ⁰⁰	<input data-bbox="1209 465 1305 488" type="button" value="Get it now!"/>
bobbyiliev .ml	• FREE	EUR 0. ⁰⁰	<input data-bbox="1209 562 1305 584" type="button" value="Get it now!"/>
bobbyiliev .ga	• FREE	EUR 0. ⁰⁰	<input data-bbox="1209 658 1305 680" type="button" value="Get it now!"/>

In my case, **bobbyiliev.ml** is available so I will go for that one by clicking on the **Get it now!** button, and then click on **Checkout**.

You would get to a page where you could choose for how long would you like to keep the domain name for:

Domain
 or

Period

3 Months @ FREE
1 Month @ FREE
2 Months @ FREE
3 Months @ FREE
4 Months @ FREE
5 Months @ FREE
6 Months @ FREE
7 Months @ FREE
8 Months @ FREE
9 Months @ FREE
10 Months @ FREE
11 Month @ FREE
12 Months @ FREE
1 Year @ EUR 8.22
2 Years @ EUR 16.44
3 Years @ EUR 24.66
4 Years @ EUR 32.88
5 Years @ EUR 41.10
6 Years @ EUR 49.32
7 Years @ EUR 57.54
8 Years @ EUR 65.76

Choose the period from the dropdown menu and then hit next. After that follow the steps and sign up for a free account.

Once you are ready with the registration process, then go ahead and

proceed to the next step where you will need to add your new free domain name to your DigitalOcean account.

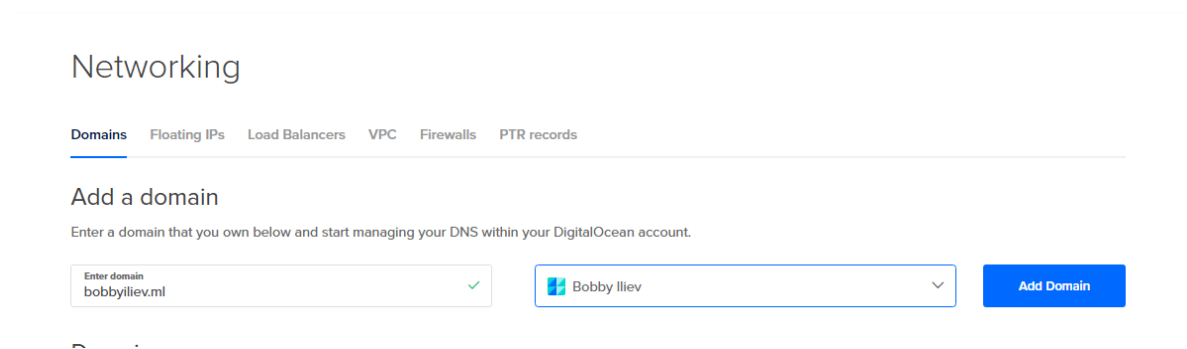
Adding your Domain to DigitalOcean

If you don't have a DigitalOcean account already make sure to create one via this link here:

[Sign up for DigitalOcean](#)

Once you've created a DigitalOcean account follow these steps here:

- First go to your [DigitalOcean Control Panel](#)
- Then click on Networking on the left
- After that click on Domains
- And there add your new domain name button and then add the domain name that you just registered:



The screenshot shows the DigitalOcean 'Networking' section with the 'Domains' tab selected. Below the navigation bar, there's a heading 'Add a domain' and a subtext 'Enter a domain that you own below and start managing your DNS within your DigitalOcean account.' The form consists of a text input field containing 'bobbyiliev.ml' with a green checkmark, a dropdown menu showing 'Bobby Iliev' with a downward arrow, and a blue 'Add Domain' button.

Then you would see your new DNS zone where you would need to add an A record for your domain name and point it your Laravel server! For more information on [how to manage your DNS make sure to read this guide!](#)

Updating your Nameservers

Once you have your Domain name all configured and ready to go in DigitalOcean, then you need to update your Nameservers, so that your domain would start using your new DigitalOcean DNS zone.

To do so just copy the following 3 nameservers:

- ns1.digitalocean.com
- ns2.digitalocean.com
- ns3.digitalocean.com

And then go back to your Freenom account, there follow these steps:

- From the menu click on Services
- Then click on My Domains
- After that click on Manage Domain for your new domain
- Then from the "Management Tools" dropdown click on "Nameservers"
- There choose "Use custom nameservers (enter below)" and enter the 3 DigitalOcean nameservers from above and click "Save"

The screenshot shows the Freenom domain management interface for the domain **bobbyiliev.ml**. At the top, there is a navigation bar with links: Information, Upgrade, Management Tools (with a dropdown arrow), and Manage Freenom DNS. Below this is a blue banner that says "Changes Saved Successfully!". The main content area is titled "Nameservers" and includes a sub-header "Managing bobbyiliev.ml". A note states: "You can change where your domain points to here. Please be aware changes can take up to 24 hours to propagate." There are two radio button options: "Use default nameservers (Freenom Nameservers)" and "Use custom nameservers (enter below)". The second option is selected. Below this, there are three input fields for Nameserver 1, Nameserver 2, and Nameserver 3, each containing the text "ns1.digitalocean.com", "ns2.digitalocean.com", and "ns3.digitalocean.com" respectively.

Finally, it could take up to 24 hours for the DNS to propagate over the

Globe and after that, you will be able to see your Laravel application when visiting your free domain name in your browser!

Conclusion

Now you know how to get a free domain name for your Laravel Project and add it to DigitalOcean where you could manage your DNS zone and point it to your Laravel server.

[Originally posted here](#)

8 Awesome VS Code Extensions for Laravel Developers

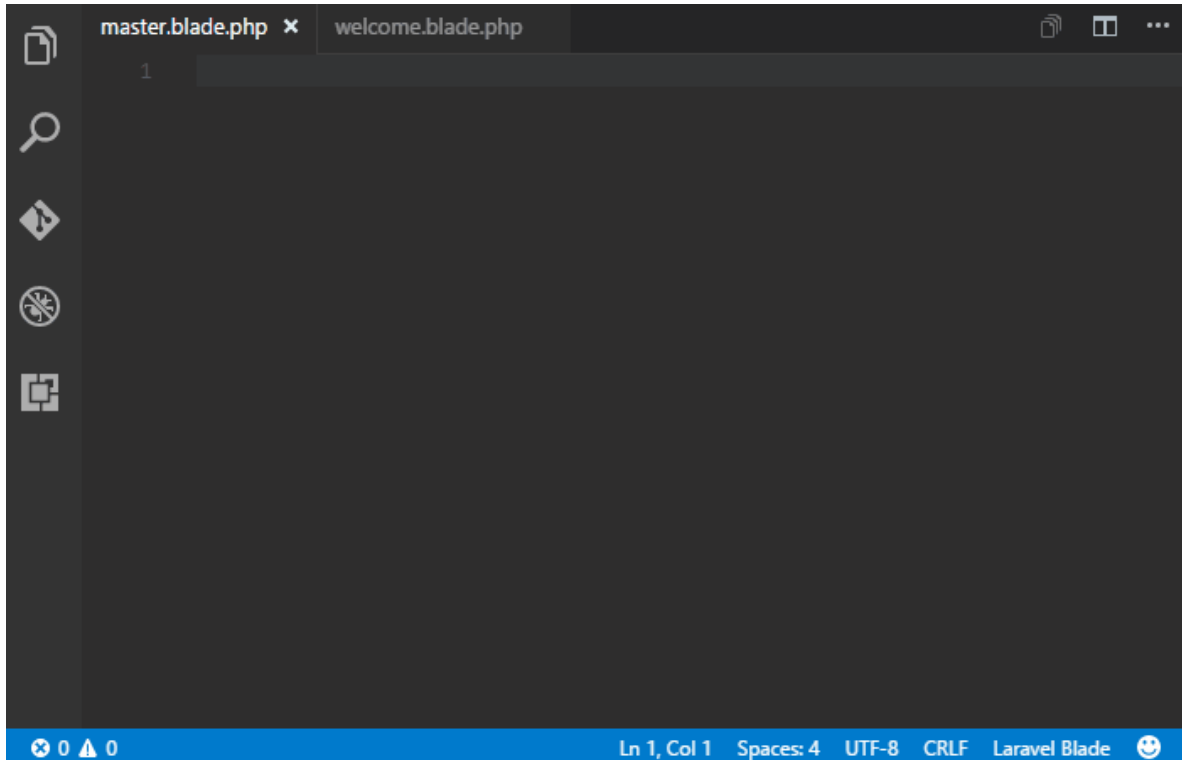
While I'm still a sublime fan for quite some time, I've been mainly using VS Code.

For anyone who is just getting started with Laravel, I would recommend going through this [Laravel basics course here!](#)

Here is a list of my top 8 VS Code extensions for Laravel developers, which would help you be more productive!

1. Laravel Blade Snippets

The [Laravel blade snippets](#) extension adds syntax highlight support for Laravel Blade to your VS Code editor.



Some of the main features of this extension are:

- Blade syntax highlight
- Blade snippets
- Emmet works in blade template
- Blade formatting

In order to make sure that the extension works as expected, there is some additional configuration that needs to be done. Go to **File** -> **Preferences** -> **Settings** and add the following to your **settings.json**:

```
"emmet.triggerExpansionOnTab": true,  
"blade.format.enable": true,  
"[blade]": {  
    "editor.autoClosingBrackets": "always"  
},
```

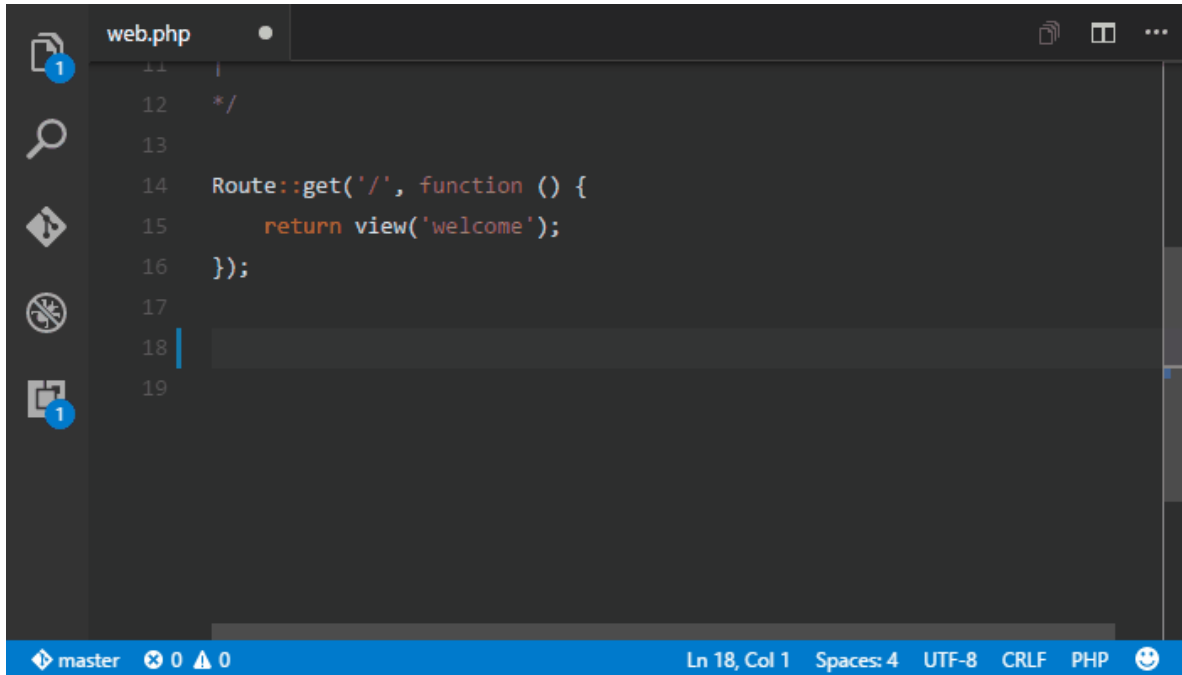
This will enable tab completion for emmet tags and if enable blade formatting.

For more information on the available snippets, make sure to check the documentation here:

[VSCode extensions for laravel](#)

2. Laravel Snippets

This one is probably my personal favorite! The [Laravel Snippets extension](#) adds snippets for the Facades like `Request::`, `Route::` etc.



Some of the supported snippet prefixes include:

- Auth
- Broadcast
- Cache
- Config
- Console
- Cookie
- Crypt
- DB
- Event
- View

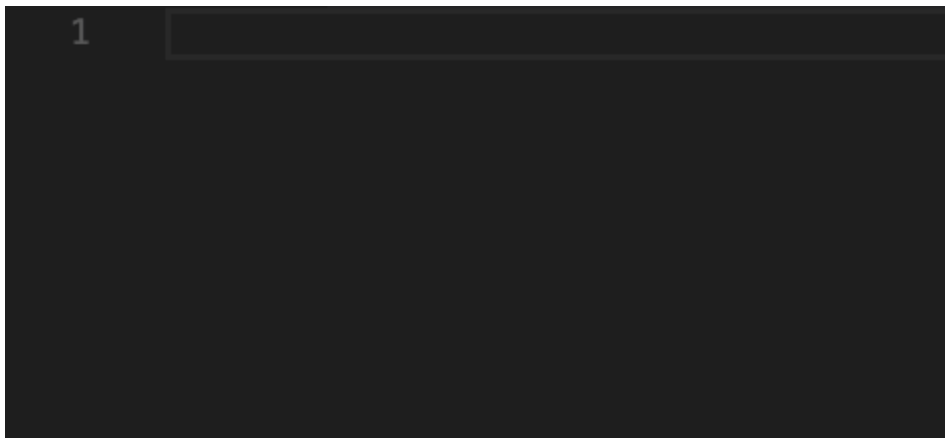
For more information on the available snippets, make sure to check the [documentation](#) here:

VSCode extensions for laravel

3. Laravel Blade Spacer

Isn't it annoying when you try to echo out something in your Blade views with `{{ }}` and your whole line going back 4 spaces? Well, luckily, the [Laravel Blade Spacer](#) fixes that!

The Laravel blade spacer extension automatically adds spacing to your blade templating markers:

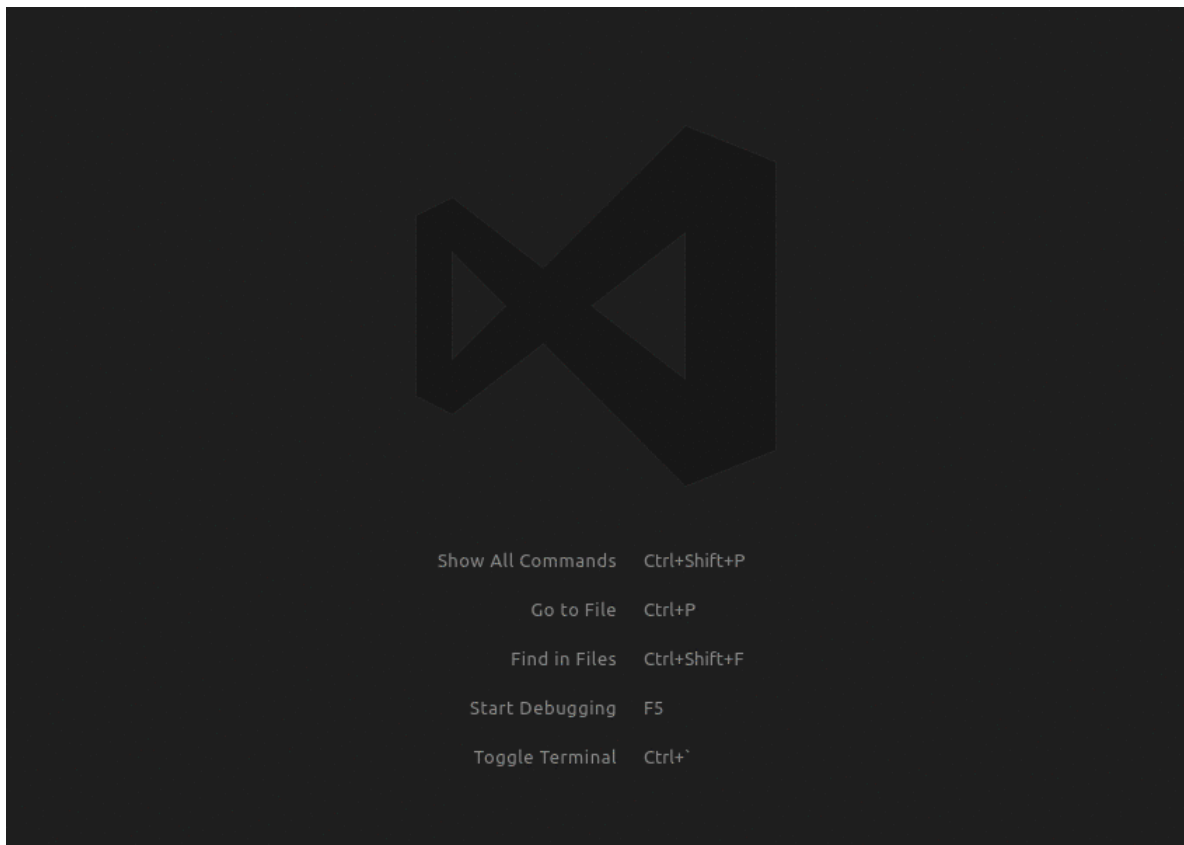


For more information, make sure to check the documentation here:

[VSCode extensions for laravel](#)

4. Laravel Artisan

I personally like to use the command line all the time, but I have to admit that the [Laravel Artisan](#) extension is awesome! It lets you run Laravel Artisan commands from within Visual Studio Code directly!



Some of the main features are:

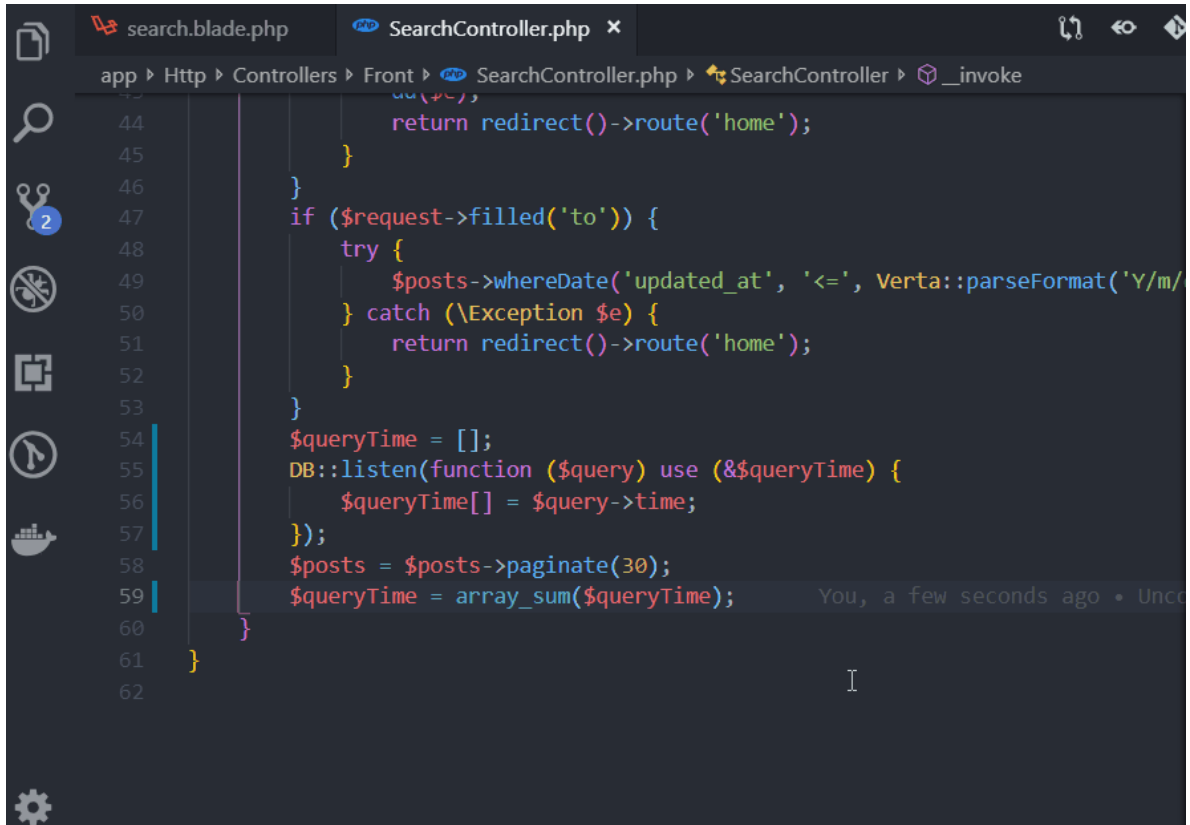
- Make files like Controllers, Migrations, etc.
- Run Your own Custom Commands
- Manage your database
- Clear the Caches
- Generate Keys
- View all application routes
- Manage your local php server for test purposes

For more information, make sure to check the documentation [here](#):

VSCode extensions for laravel

5. Laravel Extra Intellisense

The [Laravel Extra Intellisense](#) extension provides autocompletion for Laravel in VSCode.



The extension comes with auto-completion for:

- Route names and route parameters
- Views and variables
- Configs
- Translations and translation parameters
- Laravel mix function
- Validation rules
- View sections and stacks
- Env
- Route Middlewares

For more information, make sure to check the documentation [here](#):

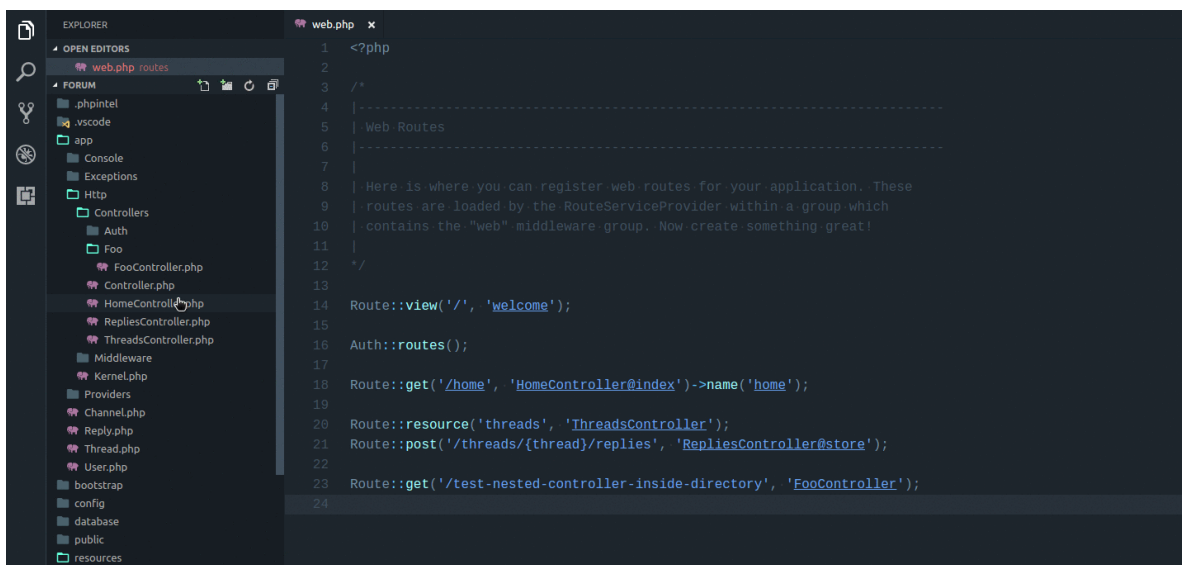
VSCode extensions for laravel

6. Laravel goto Controller

As your application grows, the number of your Controllers grows as well, so at some point, you might end up with hundreds of controllers. Hence finding your way around might get tedious.

This is the exact problem that the [Laravel-goto-controller](#) VScode extension solves.

The extension allows you to press **Alt** + click on the name of the controller in your routes file, and it will navigate you from the route to the respective controller file:



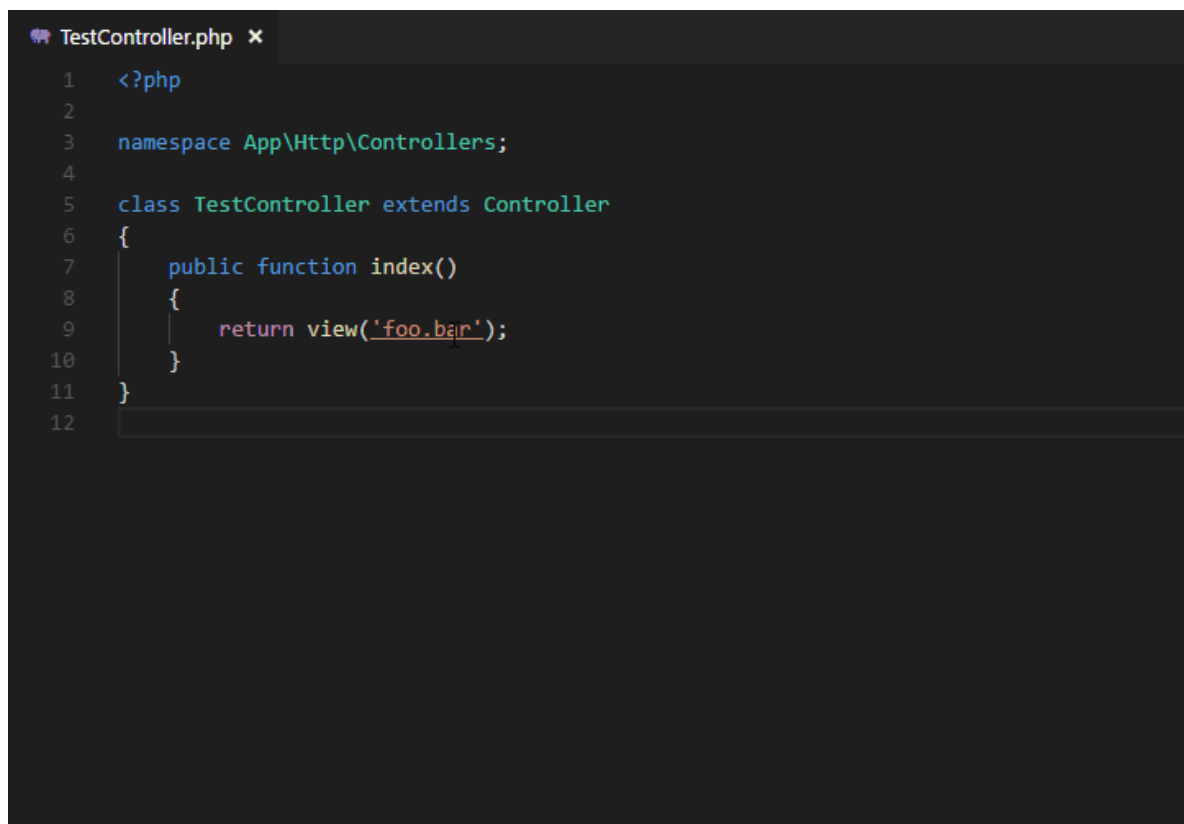
For more information, make sure to check the documentation here:

[VSCode extensions for laravel](#)

7. Laravel goto View

Similar to the Laravel goto Controller extension, the [Laravel goto View VSCode extension](#) allows you to go from your Controller or Route to your view. This can save you quite a bit of time!

You can use **Ctrl** or **Alt** + click to jump to the first matched Blade View file:



```
TestController.php x
1  <?php
2
3  namespace App\Http\Controllers;
4
5  class TestController extends Controller
6  {
7      public function index()
8      {
9          return view('foo.bar');
10     }
11 }
12
```

For more information, make sure to check the documentation here:

[VSCode extensions for laravel](#)

8. DotENV syntax highlighting

This one is pretty simple but handy. The [DotENV](#) VS Code extension is used to highlight the syntax of your `.env` file, which could be quite handy for spotting some problems:



For more information, make sure to check the documentation here:

[VSCode extensions for laravel](#)

Book recommendation

If you are a Laravel fan, make sure to check out the [The Laravel Survival Guide](#) ebook!

Conclusion

If you like all those extensions, you can take a look at the [Laravel Extension Pack for Visual Studio Code](#), where you could get all of the mentioned extensions as 1 bundle!

The only extension not included in the pack is the Laravel Blade Spacer, so make sure to install it separately!

[Originally posted here.](#)

What is Laravel Jetstream and how to get started

[Laravel 8](#) was released on September 8th 2020 along with Laravel Jetstream.

[Laravel Jetstream](#) is a new application scaffolding for Laravel. Laravel Jetstream replaces the legacy Laravel authentication UI available for previous Laravel versions.

In this tutorial, I will give you a quick introduction to what exactly Laravel Jetstream is and how to get started with it.

If you want to follow along, you would need a LEMP server together with [composer](#) or the latest Laravel installer.

I will use DigitalOcean for the demo. If you do not have a DigitalOcean account yet, you can use the following referral link to get free \$100 credit that you could use to deploy your servers and test the guide yourself:

[DigitalOcean \\$100 Free Credit](#)

What is Laravel Jetstream

Jetstream gives you a better starting point for your new projects. It includes the following components:

- Login and registration functionality
- Email verification
- Two-factor authentication
- Session management
- API support via Laravel Sanctum

Laravel Jetstream replaces the legacy Laravel authentication UI available for previous Laravel versions.

Jetstream uses Tailwind CSS, and you can choose between Livewire or Inertia.

Laravel Jetstream is free and opensource.

Installing Laravel Jetstream

You can choose between a couple of ways of installing Laravel Jetstream. You could either use `composer` or the Laravel installer.

Installing Jetstream with Laravel installer

If you already have the latest version of the [Laravel installer](#), you just need to use the `--jet` flag in order to install a new Laravel Jetstream project:

```
laravel new project-name --jet
```

After that, as usual, make sure to run your migrations:

```
php artisan migrate
```

Installing Jetstream with Composer

If you prefer using composer, you need to run the following command inside your Laravel directory just like you would with any other package:

```
composer require laravel/jetstream
```

Note: you need to have Laravel 8 installed. Otherwise, the above command will fail.

After that, you would need to run `artisan jetstream:install` and specify the stack that you want to use:

- If want to use Livewire with Blade run:

```
php artisan jetstream:install livewire
```

- And if you want to use Inertia with Vue run:

```
php artisan jetstream:install inertia
```

You may also add the `--teams` flag to enable [Laravel Jetstream team support](#).

After that, execute:

```
npm install && npm run dev
```

The command above will build your assets.

Finally, make sure to run your migrations:

```
php artisan migrate
```


Authentication

Your new Jetstream application comes out of the box with:

- Login form
- Two-factor authentication
- Registration form
- Password reset
- Email verification

You can find those views at:

```
resources/views/auth
```

The backend logic is powered by [Laravel Fortify](#).

You can find the Fortify actions at the following directory:

```
app/Actions/Fortify/
```

And you can find the Fortify configuration at:

```
config/fortify.php
```

In the `fortify.php` config file, you can make some changes like enable and disable different features like:

```
'features' => [  
  Features::registration(),  
  Features::resetPasswords(),  
  // Features::emailVerification(),  
  Features::updateProfileInformation(),  
  Features::updatePasswords(),  
  Features::twoFactorAuthentication(),  
],
```

Profile management

Right out of the box, Jetstream provides you and your users with user profile management functionality which allows users to update their name, email address, and their profile photo.

The user profile view is stored at:

```
resources/views/profile/update-profile-information-  
form.blade.php
```

And in case you are using Inertia, the view can be found at:

```
resources/js/Pages/Profile/UpdateProfileInformationForm.vue
```

The following file handles the user update logic:

```
app/Actions/Fortify/UpdateUserProfileInformation.php
```

In case that you want to, you could also disable the user profile picture via your Jetstream config file at:

```
config/jetstream.php
```

Just comment out the `Features::profilePhotos()` line:

```
'features' => [  
  // Features::profilePhotos(),  
  Features::api(),  
  // Features::teams(),  
],
```

Jetstream Security

Laravel Jetstream comes with the standard functionality that allows users to update their password and log out:



However, what's more, impressive is that Jetstream also offers two-factor authentication with QR code, which the users could enable and disable directly:



Another brilliant security feature is that users can logout other browser sessions as well.



The profile Blade views can be found at:

```
resources/views/profile/
```

And the if you are using Inertia, you can find them at:

```
resources/js/Pages/Profile/
```

Jetstream API

Laravel Jetstream uses [Laravel Sanctum](#) to provide simple token-based API.

With Sanctum, each user can generate API tokens with specific permissions like Create, Read, Update, and Delete.

Then to check the incoming requests, you can use the `tokenCan` method like this:

```
$request->user()->tokenCan('read');
```

Again you can disable API support in your `config/jetstream.php` config file.

Jetstream Teams

If you used the `--team` flag during your Jetstream installation, your website would support team creation and management.

With the Jetstream teams feature, each user can create and belong to multiple different teams.

For more information about Jetstream teams, you can take a look at the official documentation [here](#).

Conclusion

Laravel Jetstream gives you a great head start when starting a new project!

I also suggest going through this post here on [what's new in Laravel 8!](#)

[Originally posted here](#)

How to check Laravel Blade View Syntax using artisan

The Laravel community has been growing exponentially, and there are a lot of fantastic packages out there. I recently came across a Laravel package that provides an artisan command to check the syntax of your blade templates.

In this tutorial, I will show you how to use the [Laravel Blade Linter package](#) to check your blade views syntax!

Before getting started, you need to have a Laravel project up and running.

You can use the following referral link to get free \$100 credit that you could use to deploy your servers and test the guide yourself:

[DigitalOcean \\$100 Free Credit](#)

After that, you can follow the steps on [How to Install Laravel on DigitalOcean with 1-Click here!](#)

Installation

```
composer require magentron/laravel-blade-lint
```

Testing

Once you have the package installed, in order to test your Blade syntax, you need to run the following command:

```
php artisan blade:lint
```

If you don't have any errors, you would see the following output:

```
All Blade templates OK!
```

On another note, if there are any issues anywhere in your blade files, you would get an error like this one:

```
PHP Parse error: syntax error, unexpected ':', expecting '('  
in /var/www/html/resources/views/welcome.blade.php on line 103  
Found 1 errors in Blade templates!
```

In my case, it tells me that I have a syntax error on line 103 in my `welcome.blade.php` file.

If you don't want to check all of your views, you can specify the path to a specific directory:

```
php artisan blade:lint resources/views/blog
```

Conclusion

The Laravel Blade Linter is a great package that can help you avoid errors before pushing your code to production!

If you like the package, make sure to contribute at on [Github](#)!

[Originally posted here](#)

How to speed up your Laravel application with PHP OPcache

Using PHP [OPcache](#) is a great way to improve your overall performance. OPcache stores pre-compiled script bytecode in memory, which eliminates the need for PHP to load and parse scripts on every request.

In this tutorial, you will learn how to use the Laravel along with OPcache to speed up your website!

In order for you to be able to follow along, you need to have a Laravel project up and running.

You can use the following referral link to get free \$100 credit that you could use to deploy your servers and test the guide yourself:

[DigitalOcean \\$100 Free Credit](#)

After that, you can follow the steps on **[How to Install Laravel on DigitalOcean with 1-Click here!](#)**

Enable PHP OPcache

There are a couple of ways of checking if you already have PHP OPcache enabled.

You could either add a PHP info file in your Laravel **public** folder with the following content:

```
<?php  
phpinfo();
```

And then visit the file via your browser. After that, use **CTRL+F** and search for OPcache. You will see the following information:



Another way of checking if OPcache is enabled is to run the following command via your terminal:

```
php -m | grep -i opcache
```

The output that you would see the following result:

```
Zend OPcache
```

If you don't have OPcache enabled, you can install it with the following command on Ubuntu:

```
sudo apt install php-opcache
```

If you are not using Ubuntu, you can install PHP OPcache using **pecl**:

<https://pecl.php.net/package/ZendOPcache>

If you used the 1-Click image from the prerequisites, OPcache would already be installed for you.

Configure PHP OPcache

Once you have OPcache installed, you can adjust some of the default configuration settings to optimize the performance.

To make those changes, you need to edit your `php.ini` file. If you are not sure where exactly the `php.ini` file is located at, you can again use a PHP info file to check the location.

If you have used the 1-Click image from the prerequisites, you will find the `php.ini` file at:

```
/etc/php/7.4/fpm/conf.d/10-opcache.ini
```

Using your favorite text editor, open the file:

```
nano /etc/php/7.4/fpm/conf.d/10-opcache.ini
```

Then at the bottom of the file add the following configuration:

```
opcache.memory_consumption=256
opcache.interned_strings_buffer=64
opcache.max_accelerated_files=32531
opcache.validate_timestamps=0
opcache.enable_cli=1
```

A quick rundown of the values:

- `opcache.memory_consumption=256`: This is the size of the memory storage used by OPcache. You can increase the `opcache.memory_consumption=256` value in case that you have enough RAM on your server.

- `opcache.interned_strings_buffer=64`: The amount of memory allocated to storing interned strings. The value is in megabytes.
- `opcache.max_accelerated_files=32531`: The number of scripts in the OPcache hash table or, in other words, how many scripts can be cached in OPcache.
- `opcache.validate_timestamps=0`: With this directive, we specify that OPcache should not be cleared automatically, which means that we would need to do this manually.
- `opcache.enable_cli=1`: This enables the OPcache for the CLI version of PHP, which could be beneficial if you are using any `artisan` commands.

Once you make the change, you need to restart PHP FPM:

```
systemctl restart php7.4-fpm.service
```

For more information on the OPcache configuration, make sure to go through the official documentation here:

<https://www.php.net/manual/en/opcache.configuration.php>

Configure Laravel OPCache

In order to have some better management over the caching, we will use the [Laravel OPCache](#) package.

To use the package, you would need to have Laravel 7 or newer.

To install the package, just run the following command:

```
composer require appstract/laravel-opcache
```

One important thing that you need to keep in mind is that your `APP_URL` needs to be set correctly in your `.env` file so that it matches your domain.

Once you have the package installed, it provides you with some excellent PHP `artisan` commands which you could use to help you manage your OPCache:

- Artisan command to clear OPCache:

```
php artisan opcache:clear
```

- Artisan command to show OPCache config:

```
php artisan opcache:config
```

- Artisan command to show OPCache status:

```
php artisan opcache:status
```

- Artisan command to pre-compile your application code:

```
php artisan opcache:compile {--force}
```

Conclusion

Enabling PHP OPcache is an excellent and quick way to boost the performance of your Laravel application without having to make any significant changes.

If you like the Laravel OPcache package, make sure to give it a star on [Github](#)!

I hope that this helps!

[Originally posted here](#)

What is Laravel Sail and how to get started

Laravel is a fantastic open-source PHP framework that is designed to develop web applications following the model-view-controller scheme. And now here comes Laravel Sail.

Laravel Sail was just recently released, and it is a light-weight command-line interface for interacting with Laravel's default Docker development environment. What this means is that you won't be needing to use Docker to create different containers manually, but you can just use use Laravel Sail to do that for you.

By using Laravel Sail and you will get a fully working local development environment.

Laravel Sail uses the `docker-compose.yml` file and the sail script that is stored at the vendor folder of your project at `vendor/bin/sail`. The sail script provides a CLI with convenient methods for interacting with the Docker containers defined by the `docker-compose.yml` file.

In this post, you are going to learn how to install and get started with Laravel Sail.

In order to get started with Laravel Sail, all that you need is to have Docker and Docker Compose installed on your Laptop or your server.

If you don't know what Docker is or how it works, you can check out these post series to help you understand what Docker is and how it works.

Introduction to Docker

Installing And Setting Up Laravel Sail

One of the many great things about Laravel Sail is that it is automatically installed with all new Laravel applications so this way you can start using it straight away.

To get a new blank project you can run the following command:

```
curl -s https://laravel.build/devdojo-sail | bash
```

This will install a new Laravel application in a `devdojo-sail` folder. Note that you can change the `devdojo-sail` with the name of the folder that you would like to use.

You will see the following output:



Another great thing about it is how easy it is to set up. By default, Sail commands are invoked using the `vendor/bin/sail` script that is included with all new Laravel applications, so to make life easier let's configure the Bash alias that allows us to execute Sail's commands faster.

```
alias sail='bash vendor/bin/sail'
```

What this does is it sets `sail` to point to the `vendor/bin/sail` script, so that you can execute Sail commands without having to type `vendor/bin` each time.

Laravel Sail Commands

Now by simply typing `sail`, you may execute Sail commands. So let's start up our Sail!

One important thing to keep in mind is that before starting it up make sure that **no** other web servers or databases are up and running on your local computer.

To start all of the Docker containers defined in your application's `docker-compose.yml` file, just type in the `up` command:

```
sail up
```

This will start pulling all of the necessary Docker images and it will setup a full blown development environment on your machine:



Another thing that you could do is to start the Docker containers in the background by just adding `-d` after the `up` command. The `-d` stands for **detached**.

```
sail up -d
```

Note that if you run this initially it will take some time to get everything prepared, but after that it should be much quicker



Once you've ran the `sail up` command you can visit your server IP address or `localhost` if you are running it on your Laptop, and you will see a brand new Laravel installation.

Also if you run `docker ps -a` you will be able to see all of the Docker containers that have been started automatically thanks to Laraval Sail in order to get your Laravel Development environment working:



In case that you want to stop the running containers, you can press `Control + C`, or if you are running the containers in the background, type in the `down` command.

```
sail down
```

You can also execute PHP, Composer, Artisan, and Node/NPM commands. For example, if you want to run a PHP command it should look something like this:

```
sail php --version
```

Or if you wanted to run some Laravel Artisan commands you would usually run `php artisan migrate` but with Laravel sail it would be a matter of adding `sail` before the artisan command:

```
sail artisan migrate
```

Video Introduction

Make sure to check out the official introduction video here:

{% youtube mgyo0kfV7Vg %}

Conclusion

Laravel Sail will definitely make your work on your new Laravel application be much more enjoyable and more comfortable.

In case that you get a brand new server or a Laptop, all that you would need to get started with Laravel is Docker and Docker compose installed. Then thanks to Laravel Sail you will not have to do any fancy server configuration or install any additional packages on your Laptop like PHP, MySQL, Nginx, Composer and etc. It all comes out of the box with Sail!

Be sure to check out the documentation on Sail to understand it even more:

<https://laravel.com/docs/8.x/sail#executing-artisan-commands>

[Originally posted here](#)

How to add simple search to your Laravel blog/website

There are many ways of adding search functionality to your Laravel website.

For example, you could use [Laravel Scout](#) which is an official Laravel package, you can take a look at this tutorial here on [how to install, setup and use Laravel scout with Algolia](#).

However, in this tutorial here, we will focus on building a very simple search method without the need of installing additional packages.

Before you begin you need to have Laravel installed.

If you do not have that yet, you can follow the steps on how to do that [here](#) or watch this video tutorial on [how to deploy your server and install Laravel from scratch](#).

You would also need to have a model ready that you would like to use. For example, I have a small blog with a `posts` table and `Post` model that I would be using.

Controller changes

First, create a controller if you do not have one already. In my case, I will name the controller `PostsController` as it would be responsible for handling my blog posts.

To create that controller just run the following command:

```
php artisan make:controller PostsController
```

Then open your controller with your text editor, and add the following `search` method:

```
public function search(Request $request){
    // Get the search value from the request
    $search = $request->input('search');

    // Search in the title and body columns from the posts
    table
    $posts = Post::query()
        ->where('title', 'LIKE', "%{$search}%")
        ->orWhere('body', 'LIKE', "%{$search}%")
        ->get();

    // Return the search view with the results compacted
    return view('search', compact('posts'));
}
```

Note that you would need to change the table names which you would like to search in.

In the example above we are searching in the title and body columns from the posts table.

Route changes

Once our controller is ready, we need to add a new route in the `web.php` file:

```
Route::get('/search/',  
    'PostsController@search')->name('search');
```

We will just use a standard get request and map it to our `search` method in the `PostsController`.

Blade view changes

Now that we have the route and the controller all sorted out, we just need to add a form in our `search.blade.php` file.

I will use the following simple `GET` form:

```
<form action="{{ route('search') }}" method="GET">
  <input type="text" name="search" required/>
  <button type="submit">Search</button>
</form>
```

Feel free to add this to your existing view and style it accordingly with your classes.

Then to display the results, what you could do is use the following `foreach` loop in your view:

```
@if($posts->isNotEmpty())
    @foreach ($posts as $post)
        <div class="post-list">
            <p>{{ $post->title }}</p>
            
        </div>
    @endforeach
@else
    <div>
        <h2>No posts found</h2>
    </div>
@endif
```

The above would display your posts if there are any found and would print `No posts found` message if there are none.

Conclusion

This is just one way of building a simple search Laravel!

If you are just getting started with Laravel I would recommend going through [this Laravel introduction course](#).

[Originally posted here](#)

How to Create Custom Laravel Maintenance Page

More often than not you would need to perform some maintenance for your Laravel website.

Luckily Laravel makes it super easy to put your application in maintenance mode!

In this tutorial, I will show you how to do that, and also how to change the default Laravel maintenance page with a custom maintenance page that matches your website design!

Before you begin you need to have Laravel installed. If you do not have that yet, you can follow the steps on how to do that [here](#) or watch this [video tutorial on how to deploy your server and install Laravel from scratch](#).

You would also need SSH access to your server.

Enable Default Maintenance Mode

In order to put your application in maintenance mode all that you need to do is to run the following artisan command:

```
php artisan down
```

Then after that if you visit your website in your browser you would a 503 page that looks like this:



If you wanted to still be able to access your website, what you could do is allow your IP range so that you could browse your website as normal rather than get the 503 as all other.

To do that you just need to use the `--allow` option followed by your IP address, for example:

```
php artisan down --allow=127.0.0.1 --  
allow=your_ip_address_here/32
```

Change the `your_ip_address_here/32` part with your actual IP range, that you would use to access your website.

That way you would still be able to access your site and do some troubleshooting if needed.

For Laravel 8, you could use a secret instead:

```
php artisan down --secret="1630542a-246b-4b66-afa1-  
dd72a4c43515"
```

Then you could visit your website by adding the secret in your URL:

```
https://example.com/1630542a-246b-4b66-afa1-dd72a4c43515
```

Again with Laravel 8, alternatively you could force a redirect to a specific page:

```
php artisan down --redirect=/
```

Create Custom Maintenance Page

In order to change override the default 503 maintenance page, we need to add our own view at: `resources/views/errors/503.blade.php`.

To keep things simple I will use [this simple maintenance page](#) from GitHub.

I will just copy the content of the [file](#) here:

```
<!doctype html>
<title>Site Maintenance</title>
<style>
  body { text-align: center; padding: 150px; }
  h1 { font-size: 50px; }
  body { font: 20px Helvetica, sans-serif; color: #333; }
  article { display: block; text-align: left; width: 650px;
margin: 0 auto; }
  a { color: #dc8100; text-decoration: none; }
  a:hover { color: #333; text-decoration: none; }
</style>

<article>
  <h1>We're back soon!</h1>
  <div>
    <p>Sorry for the inconvenience but we're
performing some maintenance at the moment. If you need to you
can always <a href="mailto:#">contact us</a>, otherwise
we'll be back online shortly!</p>
    <p>&mdash; The Team</p>
  </div>
</article>
```

And paste it into the `resources/views/errors/503.blade.php` file.

After that, if I visit my website I would not see the default Laravel maintenance page, but the following page:



Of course, you can modify the page above to match your exact website design and needs!

Disable Maintenance Mode

In order to disable the maintenance mode of your website, you can just use the following artisan command:

```
php artisan up
```

This would disable the maintenance mode and your visitors would see your website as normal.

Conclusion

Being able to put your website into maintenance mode with a native `artisan` command is one of the many reasons why I love Laravel!

For more information, I would recommend checking the official Laravel documentation [here](#).

Hope that you find this helpful!

[Originally posted here](#)

What is Laravel Blade UI Kit and how to get started

The Blade UI Kit is a collection of components that you could use in your Laravel Blade.

The package was created by [Dries Vints](#) who is also a developer at Laravel.

If you are not familiar with how Laravel Blade Components work, recommend going through this tutorial here:

[The Benefits of Blade View Components](#)

In this tutorial I will give you a quick introduction to the Laravel Blade UI Kit package, show you how to install it and how to get started!

For you to be able to follow along, you would need a Laravel application up and running. If you don't have one, you can follow the steps here on [how to install Laravel with 1-Click on DigitalOcean](#).

If you are new to DigitalOcean, you can use the following referral link to get **free \$100 DigitalOcean credit** that you could use to deploy your servers and test the guide yourself:

[DigitalOcean \\$100 Free Credit](#)

Installation

To install the package, you can just use **composer**. First, go to your terminal and **cd** into your project's directory.

Then as a good practice, clear your config cache before installing the new package:

```
php artisan config:clear
```

After that, run the following command to install the Blade UI Kit package:

```
composer require blade-ui-kit/blade-ui-kit
```

After that, you need to include the Blade UI Kit CSS and JS files. To do that, just add the following:

- To include the CSS files, right before the closing **</head>** tag add:

```
@bukStyles
```

- To include the JS files, right before the closing **</body>** tag add:

```
@bukScripts
```

Next, you can publish the Blade UI Kit config if you have to make any adjustments.

Configuration

To publish the Blade UI Kit configuration file so that you could make changes to it, you can run the following command:

```
php artisan vendor:publish --tag=blade-ui-kit-config
```

The above command will copy the `./vendor/blade-ui-kit/blade-ui-kit/config/blade-ui-kit.php` to `./config/blade-ui-kit.php`.

Here's a quick rundown of the configuration file:

- First, you have a list with all of the available components:

```
/*
|-----
| Components
|-----
|
| Below, you reference all components that should be
loaded for your app.
| By default all components from Blade UI Kit are loaded
in. You can
| disable or overwrite any component class or alias that
you want.
|
*/

'components' => [
    'alert' => Components\Alerts\Alert::class,
    'form-button' => Components\Buttons\FormButton::class,
    . . .
```

You can use different names for those components by just changing the name, for example:

```
'notification' => Components\Alerts\Alert::class,
```

- After that, you can include all of your Livewire components:

```
/*
| -----
| -----
|  Livewire Components
| -----
| -----
|
|  Below you reference all the Livewire components that
should be loaded
|  for your app. By default all components from Blade UI
Kit are loaded in.
|
*/

'livewire' => [
    //
],
```

- Finally, you have a list with all of the third-party assets like AlpineJS, etc:

```

/*
|-----
|
| Third Party Asset Libraries
|-----
|
| These settings hold reference to all third party
libraries and their
| asset files served through a CDN. Individual components
can require
| these asset files through their static `$assets`
property.
|
*/

'assets' => [

    'alpine' =>
    'https://cdn.jsdelivr.net/gh/alpinejs/alpine@v2.3.5/dist/alpin
e.min.js',

    'easy-mde' => [
        'https://unpkg.com/easymde/dist/easymde.min.css',
        'https://unpkg.com/easymde/dist/easymde.min.js',
    ],
],

```

Example

Once you've installed the package, you are all set to use the out of the box components.

For example, let's say that you wanted to include a date picker to your website. Blade UI Kit makes this super easy to integrate, all you need to do is add the following component to your view:

```
<x-pikaday name="someday" />
```

The HTML out put will look like this:

```
<input  
  name="someday"  
  type="text"  
  id="someday"  
  placeholder="DD/MM/YYYY"  
>
```

On your website, the output will look like this:



Another example would be to implement a Counter on your website. To do so, you can use the following component:

```
<x-countdown :expires="$date"/>
```

The `$date` variable can be a `DateTimeInterface` instance.

Another cool thing that I like is the Color picker component. To get it on your website, you just need to use the following component:

```
<x-color-picker name="color" />
```

This would look like this:



For more awesome examples, make sure to check out [the official documentation!](#)

Conclusion

If you like the Blade UI Kit, make sure to star it on GitHub!

Also, feel free to join the [official Blade UI Kit Discord server](#), where you can meet a lot of other Laravel Developers.

I also would recommend checking out [the Blade Icons package](#), which allows you to use SVG icons in your Laravel Blade easily!

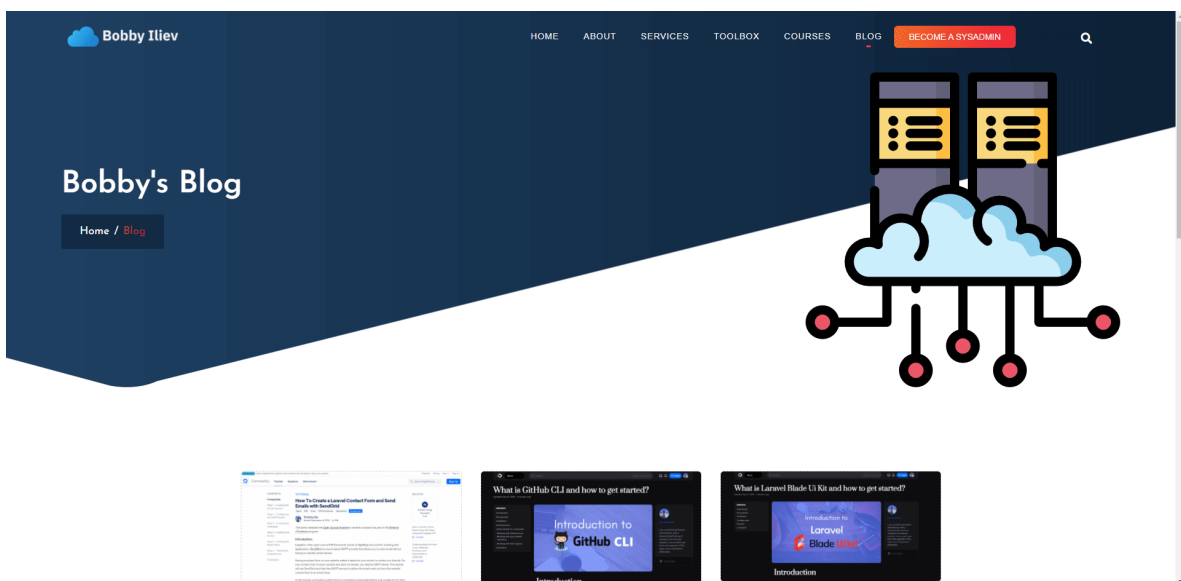
[Originally posted here](#)

How to Add a Simple Infinite Scroll Pagination to Laravel

In order to optimize your website load time, you should not load too many resources on one page as it could result in longer page load times.

That is why it is good to use pagination to limit the number of resources on each page.

In this post, I will show you how to use the built-in Laravel pagination and enhance it with infinite scrolling using [jQuery Scroll!](#)



Before you get started, you need to have a Laravel application up and running.

If you don't have that ready, you can follow the steps [How to Install Laravel on DigitalOcean with 1-Click](#)

If you are new to DigitalOcean, you can use the following referral link to get free \$100 credit that you could use to deploy your servers and test the guide yourself:

[DigitalOcean \\$100 Free Credit](#)

I will also assume that you already have a Model, a table, and some data that you can work with.

Configuring your Controller

For this example, I will have prepared a Post model and posts table with some demo posts.

So the next thing that I will do is to add a new controller:

```
php artisan make:controller BlogController
```

You will get the following output:

```
Controller created successfully.
```

In my controller, I will add a simple method to return all of my published posts:

```
public function index()
{
    $posts = \App\Post::where('status', '=', 'published')
        ->orderBy('created_at', 'desc')
        ->paginate(6);

    return view('blog.index', compact('posts'));
}
```

In my case, I will be returning 6 blog posts on each page. Feel free to change the `paginate(6)` number so that it could match your needs.

So far, this is a pretty standard way of adding pagination. In the next step, we will prepare our Blade view.

Configuring your Blade View

You can either use an existing Blade view or create a new one. In my case, as I'm returning the `blog.index` view, I need to create a new folder at `resources/views/` called `blog` and add a file inside that folder called `index.blade.php`.

I can do that with the following command:

```
mkdir resources/views/blog
```

And then create the `index.blade.php` file with the following command:

```
touch resources/views/blog/index.blade.php
```

After that, using your text editor of choice, open that file.

Now the standard way of looping through all of your posts that were returned by your controller would be something like this:

```

@foreach($posts as $post)
    <div class="main-blog">
        <div class="blog-img">
            <a href="/blog/{{ $post->slug }}">
                title }}">
            </a>
        </div>
        <div class="blog-detail">
            <a href="/blog/{{ $post->slug }}">
                <h6 class="">{{ $post->title }}</h6>
            </a>
        </div>
    </div>
@endforeach
{{ $posts->links() }}

```

In the above example, we are just looping through the `$posts` collection and displaying the different properties of every single post like the title, the image, and the slug.

Finally with the `{{ $posts->links() }}` part, we render the default Laravel pagination elements.

As we will be using `jscroll` you have to wrap that `foreach` in a `div` with a class that we will use to select and loop through with `jscroll`.

So at the end your view will look like this:

```

<div class="scrolling-pagination">
  @foreach($posts as $post)
    <div class="main-blog">
      <div class="blog-img">
        <a href="/blog/{{ $post->slug }}">
          title }}">
        </a>
      </div>
      <div class="blog-detail">
        <a href="/blog/{{ $post->slug }}">
          <h6 class="">{{ $post->title }}</h6>
        </a>
      </div>
    </div>
  @endforeach
  {{ $posts->links() }}
</div>

```

Note the `<div class="scrolling-pagination">` division which is wrapping up the whole `foreach` loop.

In the next step, we will add the jScroll plugin and hide the default pagination!

Adding and configuring jScroll

Once you have your Controller and Blade view ready, you need to include JQuery and the jScroll scripts to your project:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.5.1/jquery.min.js"></script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/jscroll/2.4.1/jquery.jscroll.min.js"></script>
```

If you prefer, you could download the scripts instead rather than using CDNJS.

Then the last thing that we need to do is configure the

```
</script>
<script type="text/javascript">
    $('ul.pagination').hide();
    $(function() {
        $('.scrolling-pagination').jscroll({
            autoTrigger: true,
            padding: 0,
            nextSelector: '.pagination li.active + li a',
            contentSelector: 'div.scrolling-pagination',
            callback: function() {
                $('ul.pagination').remove();
            }
        });
    });
</script>
```

A quick rundown of the script:

- `$('ul.pagination').hide();`: first we hide the default Laravel pagination buttons

- `$('.scrolling-pagination')`: then we select the `div` that is wrapping up our `foreach` loop and all the posts inside it
- `.jscroll();`: this is how we call the `jscroll` method on our `scrolling-pagination` `div`
- After that inside that `jscroll` method, we specify different parameters like `autoTrigger`, which triggers the autoloading of the next set of posts automatically when the user scrolls to the bottom of the containing element.

For more information on the other options, I strongly recommend going through the [documentation here](#).

Conclusion

This is pretty much it! Now you know how to add infinite scroll to your Laravel application easily!

If you like the `jScroll` plugin, make sure to star the project on [GitHub](#)!

[Originally posted here](#)

How to add a simple RSS feed to Laravel without using a package

RSS stands for Really Simple Syndication and is a feed that returns information in XML format.

Having an RSS feed would allow your users to track the latest posts on your website easily.

But having an RSS feed will also allow you to sign up for services like <http://daily.dev/> or <http://dev.to> and post your latest articles there automatically.

You can add an RSS feed to Laravel Application by using the [laravel-feed](#). However, in this tutorial, I will show you how to do that easily without adding a whole package!

To get started, all that you need is a Laravel application.

If you don't have one, you can follow the steps here on [how to install Laravel on DigitalOcean with 1-Click](#).

If you are new to DigitalOcean, you can use my referral link to get a free \$100 credit so that you can spin up your own servers for free:

[Free \\$100 DigitalOcean credit](#)

You would also need a model which you would work with. For this example, I will use my **Post** model that comes by default with the

Laravel Voyager. And I will expose the latest posts from my **Laravel Voyager blog to my RSS feed!**

Configuring RSS Controller

Let's start by creating a new controller called `RssFeedController`. You can do that with the following `php artisan` command:

```
php artisan make:controller RssFeedController
```

This will create a new controller for you, and it will add it to the `app/Http/Controllers/RssFeedController.php` directory.

After that, using your favorite text editor, open the file and add a new method called `feed` for example:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use TCG\Voyager\Models\Post;

class RssFeedController extends Controller
{
    public function feed()
    {
        $posts = Post::where('status', 'published')->
            orderBy('created_at', 'desc')->
            limit(50)->get();
        return response()->view('rss.feed',
            compact('posts'))->header('Content-Type', 'application/xml');
    }
}
```

Here is a quick rundown of the method:

- `use TCG\Voyager\Models\Post;`: First, we include the Voyager Post model. If you are not using Voyager, make sure to adapt this

accordingly

- `public function index()`: Then specify the name of the method
- `$posts = Post::where('status', 'published')`: Then we define a new variable called `$posts` which would contain all of our `published` posts
- `orderBy('created_at', 'desc')`: We sort the posts so that we could get the newest on top
- `limit(50)->get();`: here we limit the results to 50 only so that we don't get too many posts back
- `return response()->view('rss.feed', compact('posts'))`: Here we return the posts result to our `rss/feed.blade.php` view

With that, our Controller is ready! Next, we will configure our Blade view!

Configuring your Blade view

Let's start by creating a folder called `rss` inside our `resources/views/` folder:

```
mkdir resources/views/rss
```

Then in that folder, create a file called `feed.blade.php`:

```
resources/views/rss/feed.blade.php
```

Then add the following content:

```

<?=  

'<?xml version="1.0" encoding="UTF-8"?>' .PHP_EOL  

?>  

<rss version="2.0">  

    <channel>  

        <title><![CDATA[ DevDojo ]]></title>  

        <link><![CDATA[ https://your-website.com/feed  

]]></link>  

        <description><![CDATA[ Your website description  

]]></description>  

        <language>en</language>  

        <pubDate>{{ now() }}</pubDate>  

        @foreach($posts as $post)  

            <item>  

                <title><![CDATA[{{ $post->title }}]]></title>  

                <link>{{ $post->slug }}</link>  

                <description><![CDATA[!!! $post->body  

!!!]]></description>  

                <category>{{ $post->category }}</category>  

                <author><![CDATA[{{ $post->user->username  

}}]]></author>  

                <guid>{{ $post->id }}</guid>  

                <pubDate>{{ $post->created_at->toRssString()  

}}</pubDate>  

            </item>  

        @endforeach  

    </channel>  

</rss>

```

Note: make sure to update the **<title>** and the description of your site!

With that, we define our RSS feed XML structure, and inside we use a **foreach** loop to print out all of our tutorials.

In case that you are not using Laravel Voyager, make sure to adjust the **\$post->** properties so that they match your model!

Configuring your Route

Finally, we need to create a new route and map it to our RSS Controller.

To do that, open the `routes/web.php` file and add the following line:

```
Route::get('feed', 'RssFeedController@feed');
```

Essentially with that, when someone visits `yoursite.com/feed`, they will get an RSS feed response back!

You can take a look at the following link as an example:

[RSS feed example](#)

Conclusion

That is pretty much it. Now you have a fully functional RSS feed on your Laravel website!

Of course, you prefer to save some time, instead of adding the RSS functionality yourself, could use the [laravel-feed](#) package instead!

[Originally posted here](#)**

What is Laravel Zero and how to get started

[Laravel Zero](#) is an open-source PHP framework that can be used for creating console applications.

Laravel Zero is not an official Laravel package but was created by [Nuno Maduro](#), who is also a Software Engineer at Laravel, so I have no doubts about the code quality.

This tutorial, will give you a quick introduction on how to get started with Laravel Zero and built a simple **Hello World** command-line application.

Before you get started, you would need to have PHP and **composer** installed.

I will be using an Ubuntu server on DigitalOcean for this demo. If you are new to DigitalOcean, you can use my referral link to get a free \$100 credit so that you can spin up your own servers for free:

Free \$100 DigitalOcean credit

Once you have a DigitalOcean account, you can follow the steps here on how to install **composer** and PHP:

- [How To Install and Use Composer on Ubuntu 20.04](#)

Installation

Before you start with the installation, make sure that you have the following PHP modules installed:

- `php-mbstring`
- `php-xml`

To check if you have those installed already, you can just run this command:

```
php -m
```

If you are following along and you've started an Ubuntu server, you can install the module with the following command:

```
sudo apt install php-mbstring php-xml
```

To create a new Laravel Zero project, you can run the following command:

```
composer create-project --prefer-dist laravel-zero/laravel-zero hello-world
```

Note: You can change the `hello-world` part with the of your own name

Now instead of `php artisan`, you will need to run `php application`, for example:

```
php application
```

Output:

```
Application  unreleased

USAGE: application <command> [options] [arguments]

inspiring    Display an inspiring quote
test         Run the application tests

app:build    Build a single file executable
app:install  Install optional components
app:rename   Set the application name

make:command Create a new command

stub:publish Publish all stubs that are available for
customization
```

If you want to, you could also change the name of your **application** by running the following command:

```
php application app:rename hell-world
```

This will rename the **application** executable to **hello-world** so from now on, instead of running **php application** you will need to run **php hello-world**:

```
php hello-world

Hello-world unreleased

USAGE: hello-world <command> [options] [arguments]

inspiring    Display an inspiring quote
test         Run the application tests

app:build    Build a single file executable
app:install  Install optional components
app:rename   Set the application name

make:command Create a new command

stub:publish Publish all stubs that are available for
customization
```

The content of the folder will look like this:

```
README.md
app
bootstrap
box.json
composer.json
composer.lock
config
hello-world
phpunit.xml.dist
tests
vendor
```

With that, we now have our Laravel Zero application ready. Next, let's dive into some of the available commands!

Commands

In order to create a new command, you can run the following:

```
php hello-world make:command HelloWorldCommand
```

The output that you will get will be:

```
Console command created successfully.
```

This will generate a new file at:

`app/Commands/HelloWorldCommand.php`.

Open the file with your favourite text editor and change the following things:

- Change this with the name of the command that you want to have:

```
protected $signature = 'command:name';
```

- Change this with the description of your command:

```
protected $description = 'Command description';
```

- Inside the `handle()` method you can add your logic, in our case we could just output a simple message:

```
public function handle()  
{  
    echo 'Hello World';  
}
```

If you've ever created a [custom artisan command](#), you would probably find the process very similar.

Configuration

The configuration files for your Laravel Zero application are stored inside the `config` directory.

By default there are two files in there:

- `app.php`: it contains some information for your application
- `commands.php`: you can configure the list with the default commands in that file

If you add a new file inside the `config` directory, it will be automatically registered. For example, let's add a file called `hello.php`:

```
touch config/hello.php
```

Then add the following content:

```
<?php
return [
    'greeting' => 'Hello World!',
];
```

You will also be able to access it with `config('hello.greeting')`, as an example you can update the `handle()` method of your test command that you created in the last step to:

```
public function handle()
{
    echo config('hello.greeting');
}
```

If you want to have the `.env` file back, you can install the `Dotenv` addon

by running the following command:

```
php hello-world app:install dotenv
```

Next, we will go through some more of the available addons for Laravel Zero!

Addons

Out of the box Laravel Zero is quite stripped so you don't have any unnecessary code in your application, that way you can keep it as light as possible.

You can use the `app:install` command in order to include an addon.

To get a list of the available addons, just run:

```
php hello-world app:install
```

You will see the following output:

```
Laravel Zero - Component installer:
[console-dusk ] Console Dusk: Browser automation
[database     ] Eloquent ORM: Database layer
[dotenv       ] Dotenv: Loads environment variables from
".env"
[http         ] Http: Manage web requests using a fluent
HTTP client
[log          ] Log: Robust logging services
[logo         ] Logo: Display app name as ASCII logo
[menu         ] Menu: Build beautiful CLI interactive menus
[queue        ] Queues: Unified API across a variety of
queue services
[schedule-list] Schedule List: List all scheduled commands.
[self-update  ] Self-update: Allows to self-update a build
application
```

You can use the interactive menu to choose an addon and install it!

In this post I will through a couple!

Database

In order to include the Laravel's Eloquent component, you need to run the following command:

```
php hello-world app:install database
```

Note: don't forget to change the `hello-world` part with the name of your application.

In the background the `app:install` command uses composer, so the output that you would see will be quite familiar for you if you've ever used composer:

```
Installing database component...
Require package via composer: loading...
Do not run Composer as root/super user! See
https://getcomposer.org/root for details
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 1 install, 0 updates, 0 removals
  - Installing illuminate/database (v8.8.0): Downloading
(100%)
```

This will add a new config file for you at `config/database.php` where you can configure your database details. By default it uses SQLite so you don't need to make any changes unless you want to use a different SQL engine.

Once Database the addon has been installed, you can use it just as you would use Eloquent in Laravel!

Usage:

```
php hello-world make:migration create_users_table  
php hello-world migrate
```

```
use DB;
```

```
$users = DB::table('users')->get();
```

Logging

In order to install the Logging addon, you just need to run:

```
php <your-app-name> app:install log
```

This will include the Laravel logging service into your Laravel Zero project!

The usage is as follows:

```
use Log;
```

```
Log::emergency($message);  
Log::alert($message);  
Log::critical($message);  
Log::error($message);  
Log::warning($message);  
Log::notice($message);  
Log::info($message);  
Log::debug($message);
```

Building the application

Once you are ready with building the application, you can run the following command to build a single file executable:

```
php hello-world app:build
```

This will ask you for your build version and it will generate a single executable inside the **builds** folder!

Conclusion

This is pretty much it! Now you know how to use Laravel Zero and build cool command-line applications!

For more information, make sure to checkout the official documentation here:

<https://laravel-zero.com/docs/>

If you like the project, make sure to star it on [GitHub](#).

[Originally posted here](#)

How to build a blog with Laravel and Wink

In 2021 with great blogging platforms like [the Developer Blog](#), you can save yourself the hassle of building your own blog from scratch.

However, if you are planning to build a Laravel blog by yourself, you don't have to start from scratch! There is a Laravel great package called [Wink](#) developed by [Mohamed Said](#) which gives you a very nice headstart with an amazing UI where you can manage your posts!

In this tutorial, you will learn how to build a blog with Laravel and Wink!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Installation

One thing to keep in mind is that as of the time being Wink does not seem to be compatible with Jetstream, this might change in the future, or if you are feeling enthusiastic, make sure to submit a PR with a fix!

Once you have your fresh new Laravel installation-ready, you can add the Wink package by running the following command:

```
composer require themsaid/wink
```

Once the package has been added, to install Wink run the following command:

```
php artisan wink:install
```

As Wink will store the uploaded images, you need to also run the following storage command:

```
php artisan storage:link
```

Finally, update your database details at `config/wink.php` if you want to use your default one, just change `wink` to `mysql` and run the Wink migrations, :

```
php artisan wink:migrate
```

Output:

```
Migrating: 2018_10_30_000000_create_tables
Migrated: 2018_10_30_000000_create_tables (240.88ms)
Migrating: 2018_11_16_000000_add_meta_fields
Migrated: 2018_11_16_000000_add_meta_fields (86.68ms)
Migrating: 2020_05_17_000000_add_markdown_field
Migrated: 2020_05_17_000000_add_markdown_field (26.09ms)

Wink is ready for use. Enjoy!
You may log in using admin@mail.com and password:
some_pass_here
```

This will provide you with your Wink credentials, so make sure to note them down as you will need them later on to access the Wink UI!

To access the Wink UI, visit [yourdomain.com/wink](#) in your browser!

Once you login you will get to the following screen:



The Wink dashboard is super intuitive, to create a new post you just need to hit the **Create** button, this will take you to a page where you can choose your text editor:



And you can start writing some awesome content!

Wink also has some nice features like Tags, Teams and you can add multiple authors!



Now Wink provides you with all that you need to create and manage your posts, but you would need to take care of the frontend yourself! So let's go ahead and do that next!

Creating a Controller

Let's go ahead and create a controller with 2 methods, an index method for a page where we will see all of our blog posts and a method which will return a single blog post based on the slug!

To create a controller run the following command:

```
php artisan make:controller PostsController
```

This will create the file at

`app/Http/Controllers/PostsController.php`.

Open the file with your favourite text editor and first include the `WinkPost` class:

```
use Wink\WinkPost;
```

Then add your index method, here we will paginate the posts so that only 10 posts show up on each page:

```
public function index()
{
    $posts = WinkPost::with('tags')
        ->live()
        ->orderBy('publish_date', 'DESC')
        ->paginate(10);
    return view('posts.index', [
        'posts' => $posts,
    ]);
}
```

Then add the following method for the single post page:

```

    public function single($slug)
    {
        $post =
WinkPost::live()->whereSlug($slug)->firstOrFail();

        return view('posts.single', compact('post'));
    }

```

At the end your controller will look like this:

```

<?php

namespace App\Http\Controllers;

use Wink\WinkPost;
use Illuminate\Http\Request;

class PostsController extends Controller
{

    public function index()
    {
        $posts = WinkPost::live()
            ->orderBy('publish_date', 'DESC')
            ->paginate(10);
        return view('posts.index', [
            'posts' => $posts,
        ]);
    }

    public function single($slug)
    {
        $post =
WinkPost::live()->whereSlug($slug)->firstOrFail();

        return view('posts.single', compact('post'));
    }

}

```

Creating Views

I already have prepared the following layout file as per this blog post on how to create a [blog with Laravel and Livewire](#), but feel free to change the design as you need to.

This example uses [TailwindCSS](#)

```
https://devdojo.com/tnylea/create-a-blog-in-laravel-and-livewire
```

Once we have the controller in place, let's go ahead and add the two views as well!

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>@yield('title', 'Laravel Blog')</title>
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/tailwindcss/1.9.6
/tailwind.min.css">
  <link rel="stylesheet"
href="https://unpkg.com/@tailwindcss/typography@0.2.x/dist/typ
ography.min.css" />
</head>
<body>

<header class="text-gray-700 border-b body-font">
  <div class="container flex flex-col flex-wrap items-center
p-5 mx-auto md:flex-row">
    <nav class="flex flex-wrap items-center text-base
lg:w-2/5 md:ml-auto">
      <a href="/" class="mr-5 hover:text-
gray-900">Home</a>
      <a href="/blog" class="mr-5 hover:text-
gray-900">Blog</a>
```

```

        <a href="/about" class="mr-5 hover:text-
gray-900">About</a>
    </nav>
    <a class="flex items-center order-first mb-4 font-bold
text-gray-900 lg:order-none lg:w-1/5 title-font lg:items-
center lg:justify-center md:mb-0">
        BLOG
    </a>
    <div class="inline-flex ml-5 lg:w-2/5 lg:justify-end
lg:ml-0">
        <a href="#_" class="inline-flex items-center px-3
py-1 mt-4 text-base bg-gray-200 border-0 rounded
focus:outline-none hover:bg-gray-300 md:mt-0">Login</a>
    </div>
</div>
</header>

    @yield('content')

</body>
</html>

```

Let's create a very basic page where we will display all of our posts, create the following file:

```
resources/views/posts/index.blade.php
```

And add the following content:

```

@extends('layouts.app')

@section('content')

    <div class="container p-5 mx-auto">
        <h1 class="mt-32 text-4xl font-extrabold leading-10
tracking-tight text-center text-gray-900 sm:text-5xl
sm:leading-none md:text-6xl">
            Welcome to The Blog
        </h1>

        <div class="max-w-xl mx-auto mt-10">
            @foreach($posts as $post)
                <div class="pb-5 mb-5 border-b border-
gray-200">
                    <a href="/post/{{ $post->slug }}"
class="mb-2 text-2xl font-bold">{{ $post->title }}</a>
                    <p>{{ Str::limit($post->body, 100) }}</p>
                </div>
            @endforeach
        </div>
    </div>

@endsection

```

Let's do the same for the single posts:

```
resources/views/posts/single.blade.php
```

And again add the following content:

```
@extends('layouts.app')

@section('content')

    <div>
        <div class="max-w-4xl py-20 mx-auto prose lg:prose-xl">
            <h1>{{ $post->title }}</h1>
            <p>{!! $post->body !!}</p>
        </div>
    </div>

@endsection
```

With that our views are all set and we just need to add our routes!

Adding routes

The last thing that we need to do in order to be able to access the posts is to add our routes.

Open the `routes/web.php` file and add the following 2 routes:

```
Route::get('/posts', 'PostsController@index');  
Route::get('post/{slug}', 'PostsController@single');
```

After that once you visit `yourdomain.com/posts` you will be able to see all of your posts and brows through them:



Conclusion

As a next step, make sure to redesign the blog with your own theme, and if you do so, share it with me by tagging me on Twitter:

[@bobbyiliev_!](#)

Of course, this article aims to do a quick introduction to Wink, you can improve a lot of things like, it would be to good implement the meta data to your page for better SEO and etc.

If you enjoy the Wink package, make sure to star it on [GitHub](#) and contribute!

If you are new to Laravel, make sure to check out this video series on [how to build a Blog with Laravel and Voyager!](#)

[Originally posted here](#)

How to copy or move records from one table to another in Laravel

In some cases when a particular table grows too much it might slow down your Laravel website. You might want to clear old records or archive them to another table so that you could keep your main table as light as possible.

As of the time of writing this post, Laravel does not provide a helper method to move records from one table to another, so here is how you could achieve that!

As an example, I have a `posts` table and Model, and I would show you how to copy or move all posts that have not been updated in the past 5 years to another table called `legacy_posts`.

Before you begin you need to have Laravel installed.

If you do not have that yet, you can follow the steps on how to do that [here](#) or watch this [video tutorial on how to deploy your server and install Laravel from scratch](#).

You would also need to have a model ready that you would like to use. For example, I have a small blog with a `posts` table and `Post` model that I would be using.

Copy all records from one table to another

The `replicate()` method provides an optimized way of creating a clone of an entire instance of a Model.

For this example, I will use my Posts model and copy all posts that have not been updated in the past 5 years to a new table called

`legacy_posts`:

```
Posts::query()  
->where('updated_at', '<', now()->subYears(5))  
->each(function ($oldPost) {  
    $newPost = $oldPost->replicate();  
    $newPost->setTable('legacy_posts');  
    $newPost->save();  
});
```

A quick rundown of the query:

- First we select all entries from the posts table which have not been updated in the past 5 years using the `where('updated_at', '<', now()->subYears(5))` statement
- Then we create a new instance with the `replicate()` method. This would hold the result of the above query
- After that by using the `setTable('legacy_posts')` we specify the new table where we would like to store the results in
- At the end you can see that we use the `save()` method to store all entries in the new `legacy_posts` table.

Note: you need to have the `legacy_posts` table already created in order to use the above, otherwise you would get an error that the table does not exist.

Move all records from one table to another

Let's say that you wanted to not only replicate the records but also remove the old records from your main posts table as well.

In this case you could use the following query:

```
Posts::query()  
->where('updated_at', '<', now()->subYears(5))  
->each(function ($oldRecord) {  
    $newPost = $oldRecord->replicate();  
    $newPost ->setTable('legacy_posts');  
    $newPost ->save();  
  
    $oldPost->delete();  
});
```

This will delete all entries from posts table which have not been updated in the past 5 years and copy them to a new table called `legacy_posts`.

Conclusion

For more information, I would recommend going through the [official Laravel documentation regarding Replicating Models](#).

[Originally posted here](#)

How to generate title slugs in Laravel

If you are not sure [what a slug exactly is](#), you should go through this post here first:

<https://devdojo.com/devdojo/what-is-a-slug>

There are many reasons why you would want to have a nice slug for your posts rather than accessing them via their ID, for example.

In this tutorial, you will learn how to use the title of your post and generate slugs for your Laravel blog or website!

All that you would need to follow along is a Laravel installation.

If you do not have a Laravel installation, you can follow the steps here on how to install Laravel with 1-Click also, get a **free \$100**

DigitalOcean credit:

<https://devdojo.com/bobbyiliev/how-to-install-laravel-on-digitalocean-with-1-click>

Creating a slug in Laravel

To generate a slug, we can use one of the nice Helpers provided by Laravel, mainly the `Str::slug` method.

To do so, first, make sure to include the `Str` class to your Controller:

```
use Illuminate\Support\Str;
```

Then all that you need to do is to use the `Str::slug` method and pass in your title:

```
$slug = Str::slug('Your Awesome Blog Title', '-');
```

If you echo the `$slug` variable with `echo $slug`, the result will be the following:

```
your-awesome-blog-title
```

Of course, rather than passing a static string, you could pass the title from your POST request for example:

```
$slug = Str::slug($request->title, '-');
```

After that, you can store this in your database and later on retrieve your posts by slug rather than ID for example:

```
$post = Post::where('slug', $slug)->firstOrFail();
```

Conclusion

This is pretty much it! Now you know how to create title slugs in your Laravel projects!

For more useful helpers, make sure to check the official documentation here:

<https://laravel.com/docs/8.x/helpers>

[Originally posted here](#)

What is Laravel Enlightn and how to use it

The [Laravel Enlightn](#) was developed by [Miguel Piedrafita](#) and [Paras Malhotra](#).

It is an **artisan** command-line tool that checks your code and provides you with actionable recommendations on improving your application's performance, security & more.

The [Laravel Enlightn](#) comes with a free community version, which includes 60 checks, and also there is a Pro version for solo developers or businesses and teams. You can check out the pricing [here](#).

Miguel announces the release today (14 Jan 2021) via this tweet [here](#).

In this tutorial, you will learn how to install and use Laravel Enlightn to scan your website!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Installation

In order to install Laravel Enlightn, all that you need to do is run the following composer command:

```
composer require enlightn/enlightn
```

This will install the free open source Enlightn version. If you've purchased the premium one, make sure to follow the installation steps for the Pro package [here](#).

Once you have the package installed, you can publish the assets as normal with the `artisan vendor:publish` command:

```
php artisan vendor:publish --tag=enlightn
```

To check if Enlightn was installed, you can use the following command:

```
php artisan | grep enlightn
```

Output:

```
enlightn  
application!
```

```
Enlightn your
```

Configuration

Once you've installed the package and published the assets, you can find the configuration file at:

```
config/enlightn.php
```

In that file, you can configure which checks you want to be executed.

The Enlightn Pro version comes with 120 checks, whereas the free community version comes with 60.

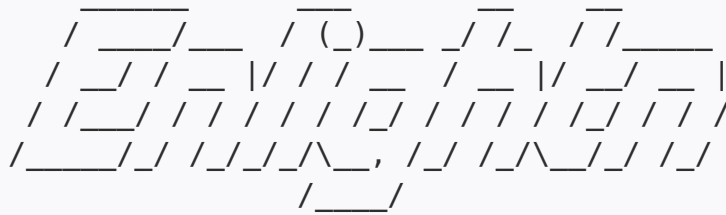
Usage

All that you need to do in order to run Enlightn is to use the following **artisan** command:

```
php artisan enlightn
```

If you are using the free version, this will trigger the 60 checks against your code, which would analyze your code's performance, security, and reliability!

Output:



Please wait **while** Enlightn scans your codebase...

```
|-----  
| Running Performance Checks  
|-----
```

Check 1/60: A proper cache driver is configured. Passed

Check 2/60: Your application caches compiled assets **for** improved performance. Passed

Check 3/60: Aggregation is **done** at the database query level rather than at the Laravel Collection level. Passed

Check 4/60: Application config caching is configured properly. Passed

Check 5/60: Your application does not use the debug log level **in** production. Passed

Check 6/60: Dev dependencies are not installed **in** production. Passed

...

You would see a lot of valuable feedback that will help you patch and optimize your code. This could protect your website against malicious attacks.

Analyzing the output

If your code passes a certain check, you will get a green message next to the check saying **Passed**.

However, if a specific check fails, you will get a summary of the problem.

For example, I'm running the script against a new SaaS that I'm working on based on [Laravel Wave](#). Thanks to Laravel Enlightn, I've noticed the following problem with one of the methods that I've created:

```
Check 30/60: Your application does not contain invalid method calls. Failed
Your application seems to contain invalid method calls to methods that either does not exist or do not match the method signature or scope.
At /app/Http/Livewire/ServiceRatings.php: line(s): 72 and 88.
Documentation URL:
https://www.laravel-enlightn.com/docs/reliability/invalid-method-call-analyzer.html
```

The output informs me where the problem is, what is wrong, and where to find more information about it.

This allows me to patch the problem before I push my code to production!

Here is another great finding where I've seen to have used an undefined variable in one of my controllers:

```
Check 39/60: Your application does not reference undefined
variables. Failed
Your application seems to reference undefined variables.
At /app/Http/Controllers/ServiceController.php: line(s): 67
and 68.
Documentation URL:
https://www.laravel-enlightn.com/docs/reliability/undefined-variable-analyzer.html
```

This will help you prevent such unnecessary mistakes in the future.

The tool also scans your environment setup like PHP, MySQL, etc., and gives you suggestions about possible optimizations.

Conclusion

I strongly recommend checking out Laravel Enlightn as it is an awesome tool that could help you protect and optimize your Laravel application.

After testing the tool on a couple of the Laravel projects that I work on, I could admit that this tool is going to be a game-changer for me, and I will definitely use it for all of my projects from now on!

For more information, make sure to check out the official documentation here:

[Laravel Enlightn](#)

[Originally posted here](#)

How to consume an external API with Laravel and Guzzle

Laravel provides a wrapper for the Guzzle HTTP client. It allows you to quickly make HTTP requests to communicate with external APIs.

In this tutorial, you will learn how to use the Laravel HTTP Client, and consume an external API and store the data in a database.

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to [get free \\$100 DigitalOcean credit](#) to spin up your servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

You would also need a [QuizAPI](#) account and an API Key.

QuizAPI is a simple REST API that is free for developers, and it provides a large number of different tech related quizzes and questions.

Creating a new table

Let's start by creating a new table called **questions** where we will store the output of the requests to the QuizAPI.

To create a new table, you could use the following **artisan** command:

```
php artisan make:migration create_questions_table
```

Output:

```
Created Migration: 2021_01_09_192430_create_questions_table
```

This would generate a new migration file for you at:

```
database/migrations/2021_01_09_192430_create_questions_table.php
```

The Laravel migrations will use the **Schema facade to create and modify database tables and columns**. To keep this simple we will only store the question itself and the available answers:

```
Schema::create('tasks', function (Blueprint $table) {  
    $table->bigIncrements('id');  
    $table->string('question');  
    $table->string('answer_a')->nullable();  
    $table->string('answer_b')->nullable();  
    $table->string('answer_c')->nullable();  
    $table->string('answer_d')->nullable();  
    $table->timestamps();  
});
```

After that, to run the migration, use this **artisan** command here:

```
php artisan migrate
```

Output:

```
Migrating: 2021_01_09_192430_create_questions_table  
Migrated:  2021_01_09_192430_create_questions_table (0.02  
seconds)
```

For more information about Laravel migrations, make sure to check out this post [here](#).

Create the Model

Once we have our `questions` table ready, let's go ahead and add a `Question` model:

```
php artisan make:model Question
```

Output:

```
Model created successfully.
```

Create the Controller

```
php artisan make:controller QuestionController
```

Output:

```
Controller created successfully.
```

QuizAPI overview

The URL that we will be hitting on the QuizAPI is the following:

```
https://quizapi.io/api/v1/questions
```

There we need to pass a couple of parameters:

- Our API key, which you can get from here:

QuizAPI Key

- And the number of questions that we want to pull

For more information, make sure to check out the official [QuizAPI documentation](#).

The output that you would get will look like this once you hit the API endpoint:

```
[
  {
    "id": 711,
    "question": "Are arrays supported in shell scripts?",
    "description": null,
    "answers": {
      "answer_a": "True",
      "answer_b": "False",
      "answer_c": "Yes but only under certain conditions",
      "answer_d": null,
      "answer_e": null,
      "answer_f": null
    },
    "multiple_correct_answers": "false",
    "correct_answers": {
      "answer_a_correct": "true",
      "answer_b_correct": "false",
      "answer_c_correct": "false",
      "answer_d_correct": "false",
      "answer_e_correct": "false",
      "answer_f_correct": "false"
    },
    "correct_answer": null,
    "explanation": null,
    "tip": null,
    "tags": [
      {
        "name": "BASH"
      },
      {
        "name": "Linux"
      }
    ],
    "category": "Linux",
    "difficulty": "Easy"
  }
]
```

For simplicity, we will want to grab only the question title and the answers from A to D.

Feel free to extend the method and the database table to grab all of the information.

Building the method

Once we have all that in place, we are ready to start building our method, which will be used to trigger the HTTP requests to the QuizAPI, get a question, and store it in our `questions` table.

With your favorite text editor, open the `QuestionController.php` file at:

```
app/Http/Controllers/QuestionController.php
```

First, make sure to include the Question model:

```
use App\Models\Question;
```

Note: if you are on Laravel 7, you need to use the following instead:

```
use App\Question;
```

After that, also include the HTTP client:

```
use Illuminate\Support\Facades\Http;
```

And then create a new public method called `fetch`, for example:

```
public function fetch()  
{  
  
}
```

Inside the `fetch` method, we can start adding our logic:

- First let's make an HTTP request to the QuizAPI questions endpoint:

```
$response =  
Http::get('https://quizapi.io/api/v1/questions', [  
    'apiKey' => 'YOUR_API_KEY_HERE',  
    'limit' => 10,  
]);
```

With the `Http` client, we are making a `GET` request, and we are hitting the `/api/v1/questions1` endpoint. We are also passing 2 parameters: the API Key and the number of questions that we want to get.

Next, as the output would be in a JSON format we can add use the following to decode it:

```
$quizzes = json_decode($response->body());
```

Then once we have the response body, let's go ahead and use a `foreach` loop to store the response in our `questions` table:

```
foreach($quizzes as $quiz){  
    $question = new Question;  
    $question->question = $quiz->question;  
    $question->answer_a =  
$quiz['answers']->answer_a;  
    $question->answer_b =  
$quiz['answers']->answer_b;  
    $question->answer_c =  
$quiz['answers']->answer_c;  
    $question->answer_d =  
$quiz['answers']->answer_d;  
    $question->save();  
}  
return "DONE";
```

The whole `fetch` method will look like this:

```
public function fetch()
{
    $response =
Http::get('https://quizapi.io/api/v1/questions', [
        'apiKey' => 'YOUR_API_KEY_HERE',
        'limit' => 10,
    ]);
    $quizzes = json_decode($response->body());
    foreach($quizzes as $quiz){
        $question = new Question;
        $question->question = $quiz->question;
        $question->answer_a =
$quiz->answers->answer_a;
        $question->answer_b =
$quiz->answers->answer_b;
        $question->answer_c =
$quiz->answers->answer_c;
        $question->answer_d =
$quiz->answers->answer_d;
        $question->save();
    }
    return "DONE";
}
```

Now let's create a simple route which we will hit and trigger the `fetch` method.

Add the route

Let's now add the route! To do so, edit the `web.php` file at:

```
routes/web.php
```

And add the following GET route:

```
Route::get('/fetch', 'QuestionController@fetch');
```

Now, if you hit the route and then check your database, you will see 10 new questions in there:



Conclusion

This is pretty much it! Now you know how to use the Laravel HTTP Client to consume an external API and store the information in your database. For more information, make sure to check out the official documentation [here](#).

The next step would be to create a view where you could render the data that you've stored in your `questions` table!

[Originally posted here](#)

How to send Discord notifications with Laravel

[Discord](#) is a messaging platform, and its popularity has been growing exponentially! It is a great alternative to Slack.

[Webhooks](#) is one of the awesome features of Discord that allows you to send any messages to a specific channel on your Discord server with just a simple POST request.

In this tutorial, you will learn how to use Discord Webhooks and send messages to your Discord channel using Laravel!

Before you start, you would need to have **composer** and Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Another thing that you would need is a Discord account and a Discord Server. You can sign up for a free account [here](#) and then follow the

steps on how to start a new server here:

<https://discord.com/register>

Once you have Laravel installed and your Discord server setup, you are ready to follow along!

Creating a Discord Channel and Webhook

Let's start by creating a new Discord channel. To do so, go to your Discord server and then click on the + next to the 'Text channels':



After that, set the channel to 'Text Channel', choose a channel name and choose if you want to have the channel private or not:



Once you have your channel ready, click on edit -> and then go to the **Integrations** tab -> and in there click on the **Create Webhook** button.

In there, choose the name of the webhook and copy the Webhook URL:



Make sure to note down your Webhook URL as we will use it in the next step when we set up our Laravel Controller!

Installing the http-client

Guzzle is a great http-client package for Laravel.

If you don't have Guzzle already installed, you can install it with the following command:

```
composer require guzzlehttp/guzzle
```

For more information about Guzzle, make sure to check the official documentation here:

<https://laravel.com/docs/8.x/http-client>

After that, let's get our controller sorted!

Adding the Controller

Let's start by creating a controller:

```
php artisan make:controller DiscordNotification
```

You will get the following response:

```
Controller created successfully.
```

And this will create your controller at:

```
app/Http/Controllers/DiscordNotification.php
```

With your text editor, open that file and add the following content:

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\Http;

class DiscordNotification extends Controller
{
    public function notification()
    {
        return Http::post('your_webhook_url_here', [
            'content' => "Learning how to send notifications
with DevDojo.com!",
            'embeds' => [
                [
                    'title' => "An awesome new notification!",
                    'description' => "Discord Webhooks are
great!",
                    'color' => '7506394',
                ]
            ],
        ]);
    }
}

```

Note:

Change the `Your_webhook_url_here` with your actual Discord Webhook URL!

A quick rundown of the controller:

- `use Illuminate\Support\Facades\Http;` - First add the `Http` so that we could make HTTP calls to the Discord Webhook URL
- `public function notification` - Then we define a new public method called `discordNotification`

- `return Http::post()` - here we do a POST HTTP request to our Discord Webhook
- `content =>` - This holds the body of the POST request. You can change this according to your needs!

One more thing to point out here is that recently [Kim Hallberg](#) showed me a website that you could use to construct embedded messages instead of just text, so I recommend checking it out here:

<https://discohook.org/>

Next, let's get our route sorted so we could test the notifications!

Adding the Routes

Once we have our controller in place, let's map it to a route we will use to access and trigger a notification!

With your text editor, open the routes file at:

```
routes/web.php
```

And add the following route:

```
Route::get('notification',  
    'DiscordNotification@notification');
```

After that, visit the `/notification` route via your browser, which will trigger the notification.

You will instantly receive a notification in your Discord channel:



Conclusion

This is pretty much it! Now you know how to send Discord notifications from your Laravel application to your Discord channels!

Of course, you could change the controller so that you pass some specific information each time base on some parameters. For example, you could send a notification every time someone posts a new article on your website, etc.

If you like Discord, you should also check out this article here on [How to Use Discord Webhooks to Get Notifications for Your Website Status](#)

[Originally posted here](#)

How to encrypt and decrypt a string in Laravel

Encryption is the process of encoding information so that it can not be understood or intercepted.

Encryption has been used long ago before computers were invented; actually, the first known evidence dates back to 1900 BC in Egypt.

Another very popular encryption technique is the [Caesar cipher](#). It is one of the simplest techniques, and how it works is each letter of the text is replaced by a specific number of positions down the alphabet.

Laravel, on the other side, provides out-of-the-box encryption, which uses OpenSSL to provide AES-256 encryption so that you don't really have to come up with your own encryption techniques.

In this tutorial, you will learn how to use the Laravel encryption to encrypt text!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

App Key Configuration

Before you get started, you need to make sure that you have an App Key generated.

If you do not have a key already generated, to do that, you can run the following command:

```
php artisan key:generate
```

The Laravel encryption will still work without a key, but the encrypted values might be insecure.

Creating a Route

Now that we have our App Key ready let's go ahead and create two routes, one for testing the Laravel encryption and one for testing the Laravel decryption.

To do that, open the `routes/web.php` file and add the following:

```
Route::get('encrypt', EncryptionController@encrypt');  
Route::get('decrypt', EncryptionController@decrypt');
```

The first route is the `/encrypt` route, which will generate an encrypted string for us, and the second one will decrypt that string.

With that, let's create the `EncryptionController` controller!

Creating a Controller

For this example, let's create a new controller called EncryptionController:

```
php artisan make:controller EncryptionController
```

This will create a new controller at:

`app/Http/Controllers/EncryptionController.php`.

Next, open that file in your text editor and first add the `Crypt` facade:

```
use Illuminate\Support\Facades\Crypt;
```

Then let's create two new public methods: one for encrypting a string and one for decrypting that same string called `encrypt` and `decrypt`.

Encryption method

Let's start with a simple Encryption method which would encrypt a hardcoded string for us:

```
public function encrypt()
{
    $encrypted = Crypt::encryptString('Hello DevDojo');
    print_r($encrypted);
}
```

Rundown of the method:

- `public function encrypt()` : first we define the method
- `Crypt::encryptString('Hello DevDojo')` : then using the

`Crypt::encryptString()` static method we encrypt a simple `Hello DevDojo` string

- `print_r($encrypted);` : finally, we would print out the encrypted string on the screen.

After that visit, the `/encrypt` URL via your browser and you will see an encrypted string similar to this one:

```
string(188)
"eyJpdiI6ImxSdGhkeVg2VHlCNUs0citKT0V4NHc9PSIsInZhbHVlIjoISEM5V
0pVWURySnVabGlnenNwTDgzUT09IiwibWFjIjoIYTJlYWVhYmI2OTJmZWJkZWV
hOTg3Nzc1ZTQwNDBlNmI3ODIzZTY5YTgwZGM3N2YwYTRmYTEwYmJiYmNjZmE2N
iJ9"
```

Note down this string and go back to your text editor to prepare the `decrypt` method!

Decryption method

Then create a new public method called `decrypt`:

```
public function decrypt()
{
    $decrypt=
    Crypt::decryptString('your_encrypted_string_here');
    print_r($decrypt);
}
```

Note:

Make sure to change the `your_encrypted_string_here` with your actual encoded string that you've got from the last step

Similar to the Encrypt method, we are again using the `Crypt` faced with the `decryptString` static method to decrypt the string!

Then this time visits the `/decrypt` URL in your browser, and you will see the decrypted `Hello DevDojo` message!

Of course, the above example is just showing `Crypt` functionality. In a real-life scenario, you would get most likely to get your string from a POST request or an API call for example.

Conclusion

For more information on how to use Laravel Encryption, make sure to check out the official documentation here:

<https://laravel.com/docs/8.x/encryption>

[Originally posted here](#)

How to remove a package from Laravel using composer

Every PHP developer should be familiar with how to use frameworks. Frameworks help make the process of development easier by simplifying common practices used in developing major web projects such as packages, modules, plug-ins, and even components.

Adding a package or packages is the primary way of adding functionality to Laravel. They might be anything from a great way to work with dates like Carbon, an entire BDD testing framework like Behat, or a package that helps users add a developer toolbar to their application that is mainly used for debugging purposes like Debugbar.

If you don't already have a Laravel application up and running, then I suggest using DigitalOceans Ubuntu Droplet. I will be using it for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Adding a package to Laravel

Let's first add an example package to our Laravel website, which we will later use as an example package to remove.

We are going to use the Laravel Sitemap package. To install the package, just use the following composer command:

```
composer require spatie/laravel-sitemap
```

This will automatically add the following entry in your `composer.json` file:

```
"spatie/laravel-sitemap": "^5.8"
```

And will download the package files in the `vendor/spatie/laravel-sitemap` folder.

Now that we have added a package, we can use it as an example and remove it.

Removing a package from Laravel using composer

To remove a Laravel package, we just need to run a single command:

```
composer remove spatie/laravel-sitemap
```

Change the `spatie/laravel-sitemap` with the name of the package that you want to remove.

This will remove the line from your `composer.json` file and also the files from the `vendor` folder.

Don't forget that you have to edit your code and ensure that the package is not being used.

Conclusion

Composer makes it super easy to add and remove Laravel packages from your projects.

Make sure to star the project on GitHub:

- [Composer GitHub](#)

If you want to learn more about Laravel packages, then I suggest you check out this [Create a Laravel Package](#) course.

[Originally posted here](#)

What is Laravel Breeze and how to get started

Laravel 8 was released on September 8th along with Laravel Jetstream. If you are not familiar with Jetstream, you should check out this introduction tutorial here:

[What is Laravel Jetstream and how to get started](#)

Due to a lot of Laravel community members complaining about the overall complexity that Jetstream adds, [Taylor Otwell](#), decided to release another package called [Laravel Breeze](#) which is much simpler than Laravel Jetstream.

In this tutorial, you will learn what Laravel Breeze is and how to get started!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Overview

Laravel Breeze provides you with a good basic starting point for building a Laravel application with authentication, which is a lot like the Laravel UI.

Laravel Breeze is built with pure Laravel Blade. However, unlike Laravel UI, which was built with Bootstrap, as of the time being, Laravel Breeze comes with Tailwind CSS only.

Also, all of the routes and controllers are exported directly to the application, so there is no hidden magic in the background, and you could see and edit everything just as you would with a regular Laravel application.



As it is quite similar to Laravel UI, the learning curve for many people would not be that steep compared to Laravel Jetstream.

Installation

In order to install Laravel Breeze, you need first to run the following `composer` command:

```
composer require laravel/breeze --dev
```

And then, to complete the installation, run this `artisan` install command:

```
php artisan breeze:install
```

You will get the following output:

```
Breeze scaffolding installed successfully.  
Please execute the "npm install && npm run dev" command to  
build your assets.
```

In order to get your static assets, you need to run the `npm install && npm run dev` command.

If you don't have `npm` installed, you can follow the steps here:

- [Install npm](#)

Then just run the command:

```
npm install && npm run dev
```

Finally, if you visit your domain name or server IP address via your browser, you will be able to see the default Laravel page with a `login`

and [register](#) link at the top right:



File structure

Once you have the Laravel Breeze package installed, you can find your Routes, Controllers, Views at the standard locations:

- Routes:

The route files are located in the `routes/auth.php` file, which, on the other hand, is included directly in your `web.php` file with the following line:

```
require __DIR__ . '/auth.php';
```

There you would have all of your auth routes like `login`, `register`, `logout`, `reset-password` etc.

- Controllers:

Just like with the Laravel UI package, the Auth controllers are stored at:

```
app/Http/Controllers/Auth/
```

Though the naming is a bit different, in that folder, you have the following controllers:

```
AuthenticatedSessionController.php  
ConfirmablePasswordController.php  
EmailVerificationNotificationController.php  
EmailVerificationPromptController.php  
NewPasswordController.php  
PasswordResetLinkController.php  
RegisteredUserController.php  
VerifyEmailController.php
```


- Views:

As you would expect, all of the auth views are stored in the following folder:

```
resources/views/auth/
```

Available Blade views:

```
confirm-password.blade.php  
forgot-password.blade.php  
login.blade.php  
register.blade.php  
reset-password.blade.php  
verify-email.blade.php
```

Video Overview

Here is a good video overview from Povilas Korop:

<https://www.youtube.com/watch?v=dofUcl1PkUA>

Conclusion

It is great to see how dedicated the whole Laravel team is, and it's super exciting to be part of that great community!

If you like the package, make sure to star it on GitHub and contribute!

<https://github.com/laravel/breeze>

[Originally posted here](#)

What are signed routes in Laravel and how to use them

Signed routes allow you to create routes accessible only when a signature is passed as a GET parameter. This could be used for sharing a preview of a draft article or any other route that you want to be public but only accessible by people who have the signature.

You could also use signed routes to allow them access to a specific route for a set period of time. For example, if you are launching a new course or SaaS, you might want to open the registration only for a day and allow just a small number of people to sign up for your product initially.

In this tutorial, you will learn how to create signed routes for your Laravel application!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Defining a signed route

To keep things simple, as an example, we will create a route that returns a discount code. The route would only be available for 10 minutes so that only the first people who get to the route would get the discount code.

To create a signed route, open your `routes/web.php` file with your favorite text editor and add the following:

```
Route::get('/discount', function(){  
    return 'some_discount_code_here';  
})->name('discountCode')->middleware('signed');
```

A quick rundown of the route:

- First, we define a get route accessible at the `/discount` URL.
- Then we create a closure that only returns a string here. In a real-life scenario, you would use a controller and possibly a Model which returns a discount code stored in your database.
- Then we specify that this is a named route with the `name()` method.
- Finally, with the `middleware('signed')` method, we specify that this is a signed route.

Now thanks to the signed middleware, the route would only be accessible if the user has the signature hash specified as a parameter.

If you try to visit the `/discount` route directly without passing the signature, you will get a `403 Invalid signature` error.

Next, we will learn how to get the temporary signed route URL!

Generating the signature

We can use the `URL::temporarySignedRoute` method to generate the signature.

Let's quickly create a Laravel controller that will generate the signed route for us:

```
php artisan make:controller DiscountController
```

This will generate the new controller at:

```
app/Http/Controllers/DiscountController.php
```

Open the file with your favorite text editor and first include the `URL` facade:

```
use Illuminate\Support\Facades\URL;
```

After that, create a public method with the following content:

```
public function discount()
{
    return URL::temporarySignedRoute(
        'discountCode', now()->addMinutes(30)
    );
}
```

Now let's create a route that will hit this method and return the signed route!

Using the signed route

We will start by adding a new route, to do so open the `routes/web.php` file and add the following:

```
use App\Http\Controllers\DiscountController;

Route::get('/generate-signature', [DiscountController::class,
'discount']);
```

Now if you access the `/generate-signature` URL via your browser you will get a similar output:

```
http://example.com/discount?expires=1617628939&signature=20a44
d614b6e4448c1f3a5bf78d3a44ca9b64b2afa940757bdb66ca2b1537974
```

Quick rundown:

- First, we define a public method called `discount`.
- After that, we use the `temporarySignedRoute` static method provided by the `URL` facade, and we give the following parameters:
 - `discountCode` is the name of our route
 - `now()->addMinutes(30)` this specifies that the signature will be valid for the next 30 minutes only

Now, if you visit the URL with the signature, you will get your discount code as the response.

Note, if you get an error that the `signed` class does not exist, make sure to add the following to the `protected $routeMiddleware` array in the `app/Http/Kernel.php` file:


```
'signed' =>  
\Illuminate\Routing\Middleware\ValidateSignature::class,
```

The above URL will only be valid for 30 minutes, but you could generate signed URLs that are valid forever if you decided to do so.

Conclusion

Now you know how to create signed routes in your Laravel application! There is a wide variety of scenarios when you might want to use this!

For more information on Laravel Signed Routes, make sure to check out the official docs here:

[Signed URLs](#)

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

[Originally posted here](#)

How to Quickly Change the Password for a User in Laravel

In some cases, you might want to reset the password for your Laravel user quickly. Another reason might be that you could be having problems with your emails, and the reset password email is not being delivered.

However, unlike WordPress, for example, where you could simply use **MD5** to encrypt your password and update it in your **users** table, Laravel uses hashing for the password encryption, so you can not change the password directly in your database.

Here are a couple of quick ways that you could use to reset your user password quickly!

Before you begin, you need to have Laravel installed.

If you do not have that yet, you can follow the steps on how to do that [here](#) or watch this video tutorial on [how to deploy your server and install Laravel from scratch](#).

You would also need to have a model ready that you would like to use. For example, I have a small blog with a **posts** table and **Post** model that I would be using.

Reset password with **tinker**

If you have been working with Laravel for a while, you've most probably used **tinker** to do some testing directly from your terminal.

To get started, run the following command:

```
php artisan tinker
```

Depending on the PHP version that you are using, you will see the following output:

```
Psy Shell v0.9.9 (PHP 7.4.14 - cli) by Justin Hileman  
>>>
```

After that, you can write a standard query to get the user that you want to change the password for. For example, if you know the email of the user, you could do something like this:

```
$user = App\Models\User::where('email',  
'your_email@example.com')->first();
```

Note: if you are using Laravel 7 or below, you might have to change **App\Models\User** to **App\User**, depending on where you store your models at.

Once you run the above, you will get the following output with some information for your user:

```
=> App\User {#3365
  id: 8,
  role_id: 1,
  name: "Bobby",
  email: "bobby@bobbyiliev.com",
  avatar: "users/default.png",
  settings: '{"locale":"en"}',
  created_at: "2021-01-29 08:21:25",
  updated_at: "2021-02-01 11:52:08",
}
```

With that, we've got our user defined as `$user` object so we can now change the password property with:

```
$user->password =
Hash::make('your_super_strong_password_here');
```

Note: Change `your_super_strong_password_here` with the new password that you want to use.

As soon as you run this, you will get an output of your hashed password.

Then finally, to persist the changes run:

```
$user->save();
```

Output:

```
=> true
```

If you are not a fan of the terminal, you could also do the same thing, but via a controller, or for the sake of simplicity, you could add a closure in your routes file with more or less the same code as above. Let's see

how to do this next!

Reset password via a route

To do that, open the `routes/web.php` file with your favorite text editor and add the following right after the opening PHP tag:

```
<?php
Route::get('temporary-password-reset', function() {
    $user = App\Moleds\User::where('email',
    'your_email@example.com')->first();
    $user->password =
    Hash::make('your_super_secure_password');
    $user->save();

    return 'Success!';
});
```

Note: if you are using Laravel 7 or below, you might have to change `App\Models\User` to `App\User`, depending on where you store your models at.

Save that, and then visit the link `/temporary-password-reset` URL via your browser. This will trigger the password change.

Note: Make sure to delete the route, once you've changed your password!

Conclusion

Those are 2 ways that you could use to quickly reset the password for your Laravel user!

If you are just getting started with Laravel, I would recommend going through [this Laravel introduction course](#).

[Originally posted here](#)

How to convert markdown to HTML in Laravel and Voyager

Markdown was created by [John Gruber](#) and [Aaron Swartz](#) who contributed on the syntax.

Nowadays Markdown is everywhere, from readme files and writing messages in online discussion forums. I'm even writing this very article in Markdown as well!

Markdown widely used as it offers an easy to write and read plain text format, which then gets converted to HTML.

In this tutorial, I will show you how to use Markdown with your Voyager Admin panel for your Laravel website and how to render the output to HTML on your views!

Before you start you would need to have a **composer** and Laravel application up and running along with Voyager installed.

I will be using a DigitalOcean Ubuntu Droplet for this demo, if you wish you can use my affiliate code to **[get free \\$100 DigitalOcean credit](#)** to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [Install Laravel on Ubuntu Server](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Once you have Laravel and Voyager installed, you are ready to follow along!

Install the Laravel Markdown Package

For this example, we will use the following Laravel Markdown package developed by [Graham Campbell](#):

<https://github.com/GrahamCampbell/Laravel-Markdown>

In order to install the package, first `cd` into the directory of your Laravel application and run the following command:

```
composer require graham-campbell/markdown
```

This might take a minute or so to complete.

After that, as stated in the GitHub Readme file, in case that you are not using automatic package discovery, you have to register the `GrahamCampbell\Markdown\MarkdownServiceProvider` service provider in your `config/app.php`.

You can also optionally alias our facade:

```
'Markdown' =>  
GrahamCampbell\Markdown\Facades\Markdown::class,
```

The package also supports different configurations which you could learn more [about here](#). To keep things simple, we will stick with the default settings.

Configure your Controller

Once you have the package installed, we are ready to change our Post controller and include the MArkdown facade.

To do so, edit your PostController and add the following line at the top:

```
use GrahamCampbell\Markdown\Facades\Markdown;
```

As an example, my single post method looks like this:

```
public function single($slug){
    $post = Post::where([[ 'slug', '=', $slug],[ 'status',
    '=' , 'published' ]])->firstOrFail();

    return view('blog.single', compact('post'));
}
```

In order to incorporate the Markdown changes, you just need to add the following line:

```
$post->body = Markdown::convertToHtml($post->body);
```

So your method would look something like this:

```
public function single($slug){  
    $post = Post::where(['slug', '=', $slug], ['status',  
    '=', 'published'])->firstOrFail();  
  
    $post->body = Markdown::convertToHtml($post->body);  
  
    return view('blog.single', compact('post'));  
}
```

All that we had to do was to first include the Markdown facade and then use the `convertToHtml` static method to parse our Markdown to HTML and return this to our Post view.

Change the Input type in Voyager

Now that we have our controller configured to parse markdown to HTML, we need to make sure that we actually, store our posts as markdown in our database.

Luckily Voyager makes that super easy! All you need to do is:

- First, go to your Voyager admin panel
- Then go to Tools -> BREAD
- Find your Posts table and click edit
- Then search for **body** and from the dropdown change input type from **Rick text box** to **Markdown Editor** and then save the changes



Once this has been done, go to Posts and edit or add a new post.

That way you will be able to write in Markdown via your Voyager control panel and have your markdown parsed to HTML on your frontend!

Conclusion

Now you know how to use Voyager's Markdown Editor and write your posts in Markdown directly via Voyager!

Hope that you find this helpful!

[Originally posted here](#)

How to Create Response Macros in Laravel

Response macros allow you to create a custom response which you could later on re-use in different routes and controllers.

This is quite beneficial in order to reduce code duplication.

You could actually built macros for other Laravel components as well, but in this tutorial, you will learn how to create a route macro for your Laravel application!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Creating a service provider

You could add your macro directly in the `app/Providers/AppServiceProvider.php` file under the `boot` method, but let's take this a step further and create our own service provider instead.

In order to create our service provider run the following command:

```
php artisan make:provider ResponseMacroServiceProvider
```

You will get the following output after running the command:

```
Provider created successfully.
```

This will generate a new file at:

```
app/Providers/ResponseMacroServiceProvider.php
```

After that, in order to register this new service provider open the `config/app.php` file and at the end of the `providers` array add the following:

```
App\Providers\ResponseMacroServiceProvider::class,
```

With that in place we can move to the next step where we will create our response macro!

Creating the Response Macro

Once we have the service provide in place it's time to build our response macro.

With your favorite text editor open the `app/Providers/ResponseMacroServiceProvider.php` file and first include the `Response` facade:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Response;

use Illuminate\Support\ServiceProvider;
```

To keep things simple, our macro will expect a string as the input, then it will `base64` encode it and return the encoded string.

To do that, in the `boot()` method we will define our macro:

```
public function boot()
{
    Response::macro('bEncode', function ($value) {
        return Response::make(base64_encode($value));
    });
}
```

The new macro is called `bEncode` for base64 encode.

Using the Response Macro

Let's quickly test the new macro that we created by adding a simple get route to the `routes/web.php` file:

```
Route::get('encode/{string}', function($string){  
    return response()->bEncode($string);  
});
```

Now if you were to visit the `/encode/some-string-here` route, you would get a response with the `some-string-here` string base64 encoded!

Conclusion

Now you know how to create a macro in your Laravel application and use it in your routes and controllers.

For more information on Laravel macros, make sure to check out the official docs here:

[Laravel Response Macros](#)

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

[Originally posted here](#)

How to Get the Base URL in Laravel

Hardcoding the domain name in your Blade files or in your controllers is not a good practice. If you ever decided to change your website's domain name, you would have to manually go over all of your files and change the references of your website from the old domain to the new one.

This is why Laravel provides a clean way of doing this by only defining your application URL in one place and then accessing it via some handy Laravel helper functions.

In this tutorial, **you will learn how to get the Base URL in your Laravel application!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

We will use a model called **Post** as an example in this tutorial.

Access the Laravel Base URL

There are multiple ways of accessing the Laravel base URL. You could use one of the following approaches:

- The `url()` helper function:

```
echo url('');
```

You can also use this in your Blade template as follows:

```
{{ url('') }}
```

This will return the value of the `APP_URL` defined in your `.env` file:

```
https://domain.com
```

Access the current URL in Laravel

The `url()` helper function comes with a lot of handy methods that you could use. Here are some of the most common ones:

- Get the current URL:

```
echo url()->current();
```

It can also be used in your Blade template directly as follows:

```
{{ url()->current() }}
```

- Get the current URL, including the GET parameters:

```
echo url()->full();
```

For more information about the `url()` helper, make sure to check out the official documentation [here](#).

Named routes

If you are using named routes, you could access the specific routes by using the `route()` helper.

This is my preferred way of doing things as you would only be referencing the name of the route, so if the URL ever changes, you would not have to make any changes to your controllers and views as the routes would still be referenced by name rather than the specific URLs.

To access a named route, you could use the following:

```
echo route('posts');
```

This would check your routes file and return the correct route with name `posts`, e.g. `https://yourdomain.com/posts`.

You can also reference this directly in your Blade templates:

```
<a href="{{ route('posts') }}">My Posts</a>
```

The great thing about the `route` helper is that you can also pass parameters. So let's say that the route for your single posts accepts a `slug` to access a specific post. You could pass the slug to the route as follows:

```
{{ route('post', ['slug' => 'awesome-post-slug']) }}
```

Alternatively to the `url()` helper, you could use the URL facade.

Access Assets URLs

In most cases, when referring to static assets stored in your public folder, like images, JavaScript and CSS files, the best approach is to use the `asset()` helper.

```

```

This will return `https://yourdomain.com/logo.png`, which is particularly handy if you are visiting the site from sub-pages like `/post/your-post` as if you don't specify the `asset()` helper, the logo would be loaded from `https://yourdomain.com/post/logo.png` which would result in 404. This is why I tend to use the `asset()` helper whenever possible.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)

How to limit the length of a string in Laravel

There are many different ways to limit the length of a string. For example, you could use CSS, JavaScript, or do it through PHP.

Laravel also provides a nice helper to make this easy! We will be using the `Str` class from `Illuminate\Support\Str` namespace.

Before you begin you need to have Laravel installed. If you do not have that yet, you can follow the steps on how to do that [here](#) or watch this video tutorial on [how to deploy your server and install Laravel from scratch](#).

You would also need SSH access to your server.

Limit string length in Blade

In order to limit the length of a string directly in your blade files, you can use the following:

```
<p>
    {{ Str::limit($your_string, 50) }}
</p>
```

You don't have to import the namespace as this is a global "helper" PHP function available out of the box with Laravel.

Just change the `$your_string` part with your actual string and also the `50` part with the number of characters that you would like to limit your string to.

Limit string length in Model

You can use the same approach but directly in your Model rather than doing this in your views each time:

```
...
use Illuminate\Support\Str;

class Product
{
    const LIMIT = 50;

    protected $fillable = [
        ..., 'description'
    ]

    public function limit()
    {
        return Str::limit($this->description, YourClass::LIMIT
    )
    }
}
```

Then in your blade you would just need to call this method:

```
<p>
    {{ $product->limit }}
</p>
```

This is a bit cleaner as you specify the length in one file and then reuse it in multiple places.

Limit string length in Controller

You could also add the logic in your controller. So before returning your view it would look like this:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Product;

class ProductController extends Controller
{
    public function show($id)
    {
        $product = User::findOrFail($id);
        $product->description =
        Str::limit($product->description, 50);
        return view('user.profile', compact('product'));
    }
}
```

Then in your view you would just need to do:

```
<p>
    {{ $product->description }}
</p>
```

Of course, you could use the same approach and even create a service provider or limit the length of your string before storing it in your database.

Conclusion

Laravel makes it quite easy to limit the length of a string both in your blade view, your controller, and even in your model.

For more useful Laravel helpers, I would recommend checking out the official documentation [here](#).

[Originally posted here](#)

How to check 'if not null' with Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

In this tutorial, you will learn how to check if the value of a specific column is **NULL** or not with Laravel Eloquent!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Check if null: `whereNull`

Let's have the following example and let's say that we have a Users table, and we want to get a list of all users that have not specified their last name, for example.

In case we were using pure SQL, the query would look like this:

```
SELECT *  
FROM users  
WHERE last_name IS NULL;
```

Notice the `IS NULL` expression at the end

The above query would return all of the users in your database where their last name is set to `NULL`.

Laravel Eloquent provides a handy method called `whereNull`, which you could use to verify if a specific value of a given column is `NULL`.

So in this case, the above example would look like this with Eloquent:

```
$users = User::whereNull('last_name')  
->get();
```

Or in case that you prefer using the `DB` facade it would look like this:

```
$users = DB::table('users')  
->whereNull('last_name')  
->get();
```

Check if not null: `whereNotNull`

Let's have a similar example, but in this case, we want to get all users that have specified their last names.

In pure SQL, we would use the `IS NOT NULL` condition, and the query would look like this:

```
SELECT *  
FROM users  
WHERE last_name IS NOT NULL;
```

The equivalent to the `IS NOT NULL` condition in Laravel Eloquent is the `whereNotNull` method, which allows you to verify if a specific column's value **is not NULL**.

So in this case, by using the `whereNotNull` method, the above example would look like this with Eloquent:

```
$users = User::whereNotNull('last_name')  
->get();
```

And again, in case that you prefer using the `DB` facade it would look like this:

```
$users = DB::table('users')  
->whereNotNull('last_name')  
->get();
```

Conclusion

For more information and useful tips on Laravel Eloquent, make sure to check this tutorial here:

[Laravel Eloquent Make](#)

I strongly suggest checking out the official Laravel Documentation:

- [Laravel Eloquent](#)

[Originally posted here](#)

How to get the current date and time in Laravel

Working with date and time could be pretty challenging. Luckily we have the **Carbon** package that makes this super easy!

Carbon is a simple PHP API extension for DateTime. You can find more information about Carbon on their official website [here](#).

Laravel also provides a lot of handy methods that you could use throughout your application.

In this tutorial, **you will learn how to get the current date and time in Laravel!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Using Carbon to get the current date and time

As of the time being, **Carbon** comes included out of the box with all new Laravel applications.

In order to include **Carbon** to a specific controller, for example, is to include this line:

```
<?php
use Carbon\Carbon;
```

Then you will be able to use all of the handy methods that **Carbon** provides.

The main method that we will cover is the **now()** method, and in order to use it you could do the following:

```
<?php
use Carbon\Carbon;

$currentTime = Carbon::now();
```

Now, if you were to use **dd(\$currentTime)**, you would get the following output:

```
Carbon @1624629716 {#849 ▼
  date: 2021-06-25 14:01:56.305923 UTC (+00:00)
}
```

To get this to a more human-readable format, you could use the **toDateTimeString()** method:

```
<?php
use Carbon\Carbon;

$currentTime = Carbon::now();

echo $currentTime->toDateTimeString();
```

Output:

```
"2021-06-25 14:04:25"
```

Or alternatively, you could convert this to an array:

```
$currentTime->toArray();
```

If you used `dd($currentTime->toArray());` you would get the following output:

```
array:12 [▼
  "year" => 2021
  "month" => 6
  "day" => 25
  "dayOfWeek" => 5
  "dayOfYear" => 176
  "hour" => 14
  "minute" => 14
  "second" => 58
  "micro" => 12380
  "timestamp" => 1624630498
  "formatted" => "2021-06-25 14:14:58"
  "timezone" => CarbonTimeZone {#853 ▼
    timezone: UTC (+00:00)
    +"timezone_type": 3
    +"timezone": "UTC"
  }
]
```

In some cases, you might want to output the current date directly in your Blade template. Let's see how to easily do that!

Using the `now()` helper function

Laravel provides a nice helper function called `now()`, which creates a new `Carbon` instance.

You can call the `now()` helper function directly in your Blade view as follows:

```
<p>{{ now()->toDateTimeString() }}</p>
```

The output would be again as follows:

```
"2021-06-25 14:06:16"
```

If you wanted to get the time for a specific timezone, you could pass the timezone as a parameter to the `now` helper function:

```
now("Europe/Paris");
```

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)**

How to Get Current Route Name in Laravel

[Laravel routes](#) allow you to map your URLs to specific functionality, for example, a method in a controller or directly to a specific Blade view. For example, if you visit [yourdomain.com/contact](#), Laravel will return the contact page of your website.

In this tutorial, you will learn how to get your current route name in your Laravel application!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Get route name in Controller or Middleware

There are a few ways that you could get your Route's name in your Controller or Middleware.

You could use either the `Request` facade or the `Route` facade directly.

Get route name using Route facade

To get your route's name by using the `Route` facade, you could use the following:

- First include the `Route` facade to your controller:

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

- And then use the following in your method:

```
public function yourMethodHere(){  
    $routeName = Route::currentRouteName();  
    dd($routeName);  
}
```

Or if you prefer not to include the facade, you could use the following instead:

```
public function yourMethodHere(){  
    $routeName = \Route::currentRouteName();  
    dd($routeName);  
}
```

Get route name using Request facade

Quite similar to using the **Route** facade, when using the **Request** facade you can get your current route's name by using the following:

```
public function yourMethodHere(){
    $routeName = \Request::route()->getName();
    dd($routeName);
}
```

Get route name in Blade View

In some cases, you might want to check your current route name directly in your Blade view.

To do so, you could use the **Route** facade directly in your Blade view:

```
{{ Route::currentRouteName() }}
```

Or if you wanted to get your URI, you could also use the **request** function directly:

```
{{ request()->route()->uri }}
```

Getting additional information about your route

In case that you wanted to get all of the information about the route you could call the `current()` method:

```
public function yourMethodHere(){
    $routeInfo = \Route::current();
    dd($routeInfo);
}
```

You would get the following information:



You can then call specific properties like the `uri` for example:

```
public function yourMethodHere(){
    $routeInfo = \Route::current()->url;
    dd($routeInfo);
}
```

This would return only your URI!

Conclusion

For more information on Laravel Routes, make sure to check this [video on how to add custom route files!](#)

I strongly suggest checking out the official Laravel Documentation:

<https://laravel.com/docs/8.x/requests>

[Originally posted here](#)

How to Count and Detect Empty Laravel Eloquent Collections

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

Laravel provides a lot of handy methods that you could use to work with your Eloquent collections.

In this tutorial, **you will learn how count and detect empty Laravel Eloquent collections!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

We will use a model called **Post** as an example in this tutorial.

Check if a collection is empty

In order to check if a collection is empty, you could use the `isEmpty()` method. This would look as follows:

```
$posts= Model::where('active', true)->get();

if ($posts->isEmpty($posts)) {
    // Returns true when there are *no* posts
}
```

Check if a collection is not empty

Similar to the `isEmpty()` method, there is a `isNotEmpty()` method. The syntax is the same:

```
$posts = Model::where('active', true)->get();

if ($posts->isNotEmpty($posts)) {
    // Returns true when there are posts
}
```

This could be quite handy when displaying the posts on a website for example, that way you could indicate that the user has not yet added any posts.

Counting the records of a collection

With SQL if you wanted to get how many records match a specific query, you could use the `COUNT(*)` function, the syntax would look like this:

```
SELECT COUNT(*) FROM posts WHERE active='1';
```

In case that you had 5 active posts, the result would be 5.

When counting the total elements in a collection, the method is also called `count()`. The syntax would look like this:

```
$posts = Model::where('active', true)->get();

if ($posts->count() > 0) {
    // More than 0 posts
} else {
    // 0 posts
}
```

The `count()` method basically returns the total number of records in result.

If you wanted to check if [check a specific record is “not null” with Laravel Eloquent](#) make sure to check out this post here.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course [here](#):

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)

How to use Forelse loop in Laravel Blade

If you have ever done any coding you are most likely well familiar with the `foreach` loops. Without any doubt, a `foreach` loop is one of the best ways to iterate over the elements of the collection.

However, in case you have an empty collection, you would need an additional `if` statement so that you could point a valid message to your users.

Luckily Laravel provides awesome Blade Templates that you could use to make your life easier!

In this post, I will show you how to use `forelse` in Laravel!

Before you get started you would need to have Laravel already installed.

If this is not the case you can follow the steps here on [How to Install Laravel on DigitalOcean with 1-Click?!](#)

Check if not empty then

What I would usually do in order to check if a collection is empty in my Blade view is to use an `if-else` statement like this:

```
@if($posts->isEmpty())

    @foreach ($posts as $post)

        <p>This is the post title {{ $post->title }}</p>

    @endforeach

@else

    <p>No posts found</p>

@endif
```

In the above example, we first wrap up our `foreach` loop in an `if` statement, and by using the `isEmpty()` we check if the collection is empty, and in case that it is empty we then print the `No posts found` message.

This works pretty well, but Laravel has a more elegant way of doing things!

Forelse example

Rather than having to nest our `foreach` loop inside of an `if` statement, what we could do instead is use the `forelse` blade template:

```
@forelse ($posts as $post)

    <p>This is the post title {{ $post->title }}</p>

@empty

    <p>No posts found</p>

@endforelse
```

As you can see we would get the same result but with less code and it is much easier to read!

Conclusion

You now know how to use `forelse` in Blade views and have much more cleaner and easier to read code!

For more great Blade templates, I would recommend checking out the official documentation [here](#).

[Originally posted here](#)

How to Delete All Entries in a Table Using Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

Laravel provides a lot of handy methods that you could use to work with your Eloquent collections. More often than not, when getting some results from your database, you would want to order them based on specific criteria.

In this tutorial, **you will learn how to delete/truncate all entries in a table using Laravel Eloquent!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

We will use a model called **Post** as an example in this tutorial.

Using `truncate` to delete all entries from a table

If we were writing pure SQL, to delete all entries from a table, we could use the `TRUNCATE` function as follows:

```
TRUNCATE users;
```

This will basically truncate the entire table, and it would also reset the auto-incrementing IDs to zero.

To do the same thing with Eloquent, we could use the `DB` facade as follows:

```
DB::table('posts')->truncate();
```

Alternatively, you could call the `truncate()` method directly on your model:

```
Post::truncate();
```

Delete all entries using the `delete()` method

If you wanted to delete only some specific entries, you could first get the entries as follows:

```
Post::query()->delete();
```

The benefit of using `delete()` rather than using `truncate()` is that you could specify some conditions so that you could only delete specific entries and not all of them:

```
Post::where('active', false)->delete();
```

Alternatively, you could also use the `DB` facade rather than calling the model directly:

```
DB::table('posts')->delete();  
  
DB::table('posts')->where('active', false)->delete();
```

The main benefit of using `truncate` is that the auto-increment IDs will also be reset.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)

How to check if a record exists with Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

In this tutorial, you will learn how to check if a record exists with Laravel Eloquent!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Check if a record exists

Laravel provides a really nice method called `exists`, which allows you to check if your query returns any records rather than using the `count` method.

```
if (Post::where('slug', $slug)->exists()) {  
    // post with the same slug already exists  
}
```

I always prefer to use the `exists` method, but here is an alternative example using `count`:

```
if (Post::where('slug', $slug)->count() > 0) {  
    // post with the same slug already exists  
}
```

What we are doing here is to select all posts where the `slug` equals the slug that we've passed as an input and then do a count on that, and if it is greater than `0`, it means that a post with that slug already exists.

Create record if it does not exist already

Laravel provides another beneficial method called `firstOrCreate`. This allows you to quickly check if a specific entry already exists and, if not, create an entry for it with 1 query.

Let's have the following example, we have a post create form where we want to check if a post with a specific slug exists, and if it does not, then we want to create the post.

By using `firstOrCreate` this will look like this:

```
$post = Post::firstOrCreate([
    'slug' => $post->slug,
], [
    'title' => $post->title,
    'body' => $post->body,
    'slug' => $post->slug,
]);
```

Here first we check if a post with that slug exists by using the following statement:

```
[
    'slug' => $post->slug,
],
```

And then if it does not exist, we create it straight away with the following values:

```
[  
    'title'      => $post->title,  
    'body'       => $post->body,  
    'slug'       => $post->slug,  
]
```

Conclusion

For more information and useful tips on Laravel Eloquent, make sure to check this tutorial here:

[Laravel Eloquent Make](#)

I strongly suggest checking out the official Laravel Documentation:

- [Laravel Eloquent](#)
- [Laravel Database queries](#)

[Originally posted here](#)

How to Add Multiple Where Clauses Using Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

In this tutorial, **you will learn how to add multiple where clauses using Laravel Eloquent!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to [get free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Chaining multiple where clauses together

Generally speaking, you could chain multiple **where** methods together like this:

```
$posts = Post::where('status', 'published')
                ->where('featured', '')
                ->where('author_id', 1)
                ->toSql();

dd($posts);
```

The output of the above to pure SQL with the **toSql** method would look like this:

```
"select * from `posts` where `status` = ? and `featured` = ?
and `author_id` = ?"
```

So the first **where** method is translated to an actual **where** in SQL, and then each **where** clause is translated to an **AND** in SQL.

Passning an array to the **where** method

An alternative to the above would be to pass an array with your specific rules. This would look like this:

```
$posts = Post::where(['status' => 'published',  
                    'featured' => '',  
                    'author_id' => 1,  
                    ])  
->toSql();  
  
dd($posts);
```

Output:

```
"select * from `posts` where (`status` = ? and `featured` = ?  
and `author_id` = ?)"
```

In case that you needed to give different rules rather than the default **equals** to, you could use the following syntax:

```
$posts = Post::where([  
    ['status', '=', 'published'],  
    ['featured', '!=', 'no'],  
    ['author_id', '>', 0],  
    ])  
->toSql();  
  
dd($posts);
```

The raw SQL for the above would look like this:

```
"select * from `posts` where (`status` = ? and `featured` != ?  
and `author_id` > ?)"
```

Using `orWhere` Clauses

In case you want your clauses to be joined with `OR` rather than `AND`, you could use the `orWhere` method:

```
$posts = Post::where('status', 'published')
                ->orWhere('featured', '')
                ->orWhere('author_id', 1)
                ->toSql();

dd($posts);
```

Output:

```
"select * from `posts` where `status` = ? or `featured` = ? or
`author_id` = ?"
```


Conclusion

For more information and useful tips on Laravel Eloquent, make sure to check this tutorial here:

- [Laravel Eloquent Make](#)

I strongly suggest checking out the official Laravel Documentation:

- [Laravel Eloquent](#)

[Originally posted here](#)

How to Add a New Column to an Existing Table in a Laravel Migration

The Laravel Migrations allow you to manage your database structure by creating new tables and columns. The Laravel migrations are like version control for your database.

In this tutorial, **you will learn how to add a new column to an existing table in a Laravel Migration!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to [get free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Creating a new table with a migration

Let's first start by creating a new table called **tasks**.

In order to create a new table, you could use the following **artisan** command:

```
php artisan make:migration create_tasks_table
```

Output:

```
Created Migration: 2020_12_23_133641_create_tasks_table
```

This would generate a new migration file for you at:

```
database/migrations/2020_12_23_133641_create_tasks_table.php
```

The content of the file would look like this:

```

<?php

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

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

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

```

The Laravel migrations will use the **Schema** facade to create and modify database tables and columns:

```

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

```

Inside the facade, you could specify the different columns that you want

to add.

So let's go ahead and add a new column called `title` which would hold the title of our tasks:

```
Schema::create('tasks', function (Blueprint $table) {  
    $table->bigIncrements('id');  
    $table->string('title');  
    $table->timestamps();  
});
```

After that, to run the migration, use this `artisan` command here:

```
php artisan migrate
```

Output:

```
Migrating: 2020_12_23_133641_create_tasks_table  
Migrated:  2020_12_23_133641_create_tasks_table (0.02 seconds)
```

Next, let's look into how to add a new column to that existing table without modifying the old migration file.

Add a New Column to an Existing Table

Now, let's say that we wanted to add a `description` column to our existing `tasks` table where we would keep the description of the tasks that we have.

To do we can again use the `php artisan make:migration` command, but this time we can specify the table that we want to add the column to with the `--table` argument:

```
php artisan make:migration add_description_to_tasks --  
table="tasks"
```

Output:

```
Created Migration: 2020_12_23_134156_add_description_to_tasks
```

It is considered a good practice to use a descriptive name for your migration.

This will again generate a new file at:

```
database/migrations/2020_12_23_134156_add_description_to_tasks  
.php
```

Note: The timestamp at the beginning of the file will indicate when the file was created

The content of the file will look like this:

```
<?php

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

class AddDescriptionToTasks extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('tasks', function (Blueprint $table) {
            //
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('tasks', function (Blueprint $table) {
            //
        });
    }
}
```

To add the new column, just specify it again in the `up()` method:

```
public function up()
{
    Schema::table('tasks', function($table) {
        $table->text('description');
    });
}
```

Another important thing to keep in mind here is that you need to update the `down()` method as well and add a `dropColumn()` method indicating that in case someone runs a migration refresh, the column should be dropped:

```
public function down()
{
    Schema::table('tasks', function($table) {
        $table->dropColumn('description');
    });
}
```

Finally, save the file and rerun your migrations:

```
php artisan migrate
```

Output:

```
Migrating: 2020_12_23_134156_add_description_to_tasks
Migrated:  2020_12_23_134156_add_description_to_tasks (0.03
seconds)
```

And this is pretty much it! This is how you can add a new column to an existing table!

Conclusion

If you are just getting started with Laravel, make sure to check out this [introduction to Laravel course here](#).

I also strongly suggest checking out the official Laravel Documentation:

- [Laravel Documentation](#)

[Originally posted here](#)

How to Rollback Database Migrations in Laravel

Laravel comes with many convenient tools out of the box, which makes your life as a developer much more enjoyable.

One of the best Laravel features is the database migrations which essentially allow you to version control your database!

In this tutorial, you will learn how to rollback your database migrations in Laravel if you ever have to do so!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Rollback the Last Database Migration

Let's say that you've run a database migration but then realized that you should not have done so. Laravel makes it easy to revert/rollback your last migration. To do so, you could just run the following command:

```
php artisan migrate:rollback --step=1
```

In case that you need to revert multiple migrations, you could change the `--step=1` with the number of migrations that you want to rollback.

If you don't specify the `--step=` flag, and just run the following:

```
php artisan migrate:rollback
```

This would rollback your last batch of migrations. So let's say that you've added 5 migration files and run `php artisan migrate` to run all 5 migrations at once. This would be considered as 1 batch.

Rollback Specific Migration

In other cases, you might want to rollback just a specific migrations file rather than the last one. You can do that with the `--path` flag followed by the path to the migrations file.

Example:

```
php artisan migrate:rollback --  
path=/database/migrations/the_specific_migration_file.php
```

This could be particularly handy if you want to make a minor change in your local dev environment, but you need to be careful with table drops and foreign keys.

Rollback All Migrations

If you wanted to rollback all of your migrations, you could run the following:

```
php artisan migrate:reset
```

Note that you should never do that in a production environment as you will lose all of your data.

The above would essentially truncate your database. In case that you wanted to drop all tables and then run the migrations from scratch, you could use the following command:

```
php artisan migrate:refresh
```

That way, you will get a fresh new database with empty tables.

Conclusion

This is pretty much it! This is how you can rollback database migrations in Laravel!

If you are just getting started with Laravel, I would recommend going through [this Laravel introduction course](#).

How to Remove a Migration in Laravel

Adding columns or tables to your database manually could be an intimidating process and would more often than not lead to database inconsistencies between your different environments.

The Laravel migrations allow you to basically version control your database so that all members of your team could have a consistent database schema.

In this tutorial, **you will learn how to remove a migration for your Laravel application!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Creating a Laravel migration

In order to create a Laravel migration, you would use the following `artisan` command:

```
php artisan make:migration create_videos_table
```

The naming convention when creating new tables is the following:

- Start with the `create` keyword
- Followed by the name of the table, in our example this is `videos`
- Followed by the `table` keyword as we are adding a new table.

If you are adding a column to an existing table rather than a brand table, you could follow the steps in this tutorial here:

- **[How to Add a New Column to an Existing Table in a Laravel Migration](#)**

Once you run the command, a new file will be created at:

```
database/migrations/the_name_of_your_migration_file.php
```

Now that we've got the migration creation covered let's see how we could actually remove a migration.

Remove a Migration in Laravel

We've got the `php artisan make:migration` to make migrations, but there is no command for removing them. To do so, you would need to actually delete the migrations file.

Let's cover two cases:

Remove a migration that hasn't been executed yet

If you've only created the migration and you've not executed the `php artisan migrate` command, all you would need to do to remove the migration is to delete the file.

You could do that via your text editor or the command line with the `rm` command.

- First check if the migration has been executed yet, you could use the following command:

```
php artisan migrate:status
```

- If the migration has not been ran yet, remove the file:

```
database/migrations/the_name_of_your_migration_file.php
```

Removing a migration that has already been executed

In case that you've ran the migration already, in order to revert it, you could use the following command:

```
php artisan migrate:rollback --step=1
```

This will revert only the last migration. After that, you could again use the `rm` command as described in the past video to actually remove the migrations file.

Reverting all migrations (DEV environments only)

Alternatively, if you are on a local dev environment and you don't actually need the data in the database, you could run `migrate:fresh` to actually revert all migrations and then run them again.

Note: if you run migrate fresh, this will wipe all of your data, so you need to be careful with that!

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about SQL in general, make sure to check out this [free introduction to SQL eBook](#) here.

[Originally posted here](#)

How to create a contact form with Laravel Livewire

Laravel Livewire was created by Caleb Porzio that allows you to add reactivity to your Laravel applications.

If you are not familiar with Laravel Livewire, I would recommend reading this [introduction to Livewire tutorial](#) first.

In this tutorial, I will show you how to add a contact form to your Laravel Livewire project!

To get started, all that you need is a Laravel application.

If you don't have one, you can follow the steps here on [how to install Laravel on DigitalOcean with 1-Click](#).

If you are new to DigitalOcean, you can use my referral link to get a free \$100 credit so that you can spin up your own servers for free:

[Free \\$100 DigitalOcean credit](#)

You will also need an SMTP server. For example, I will be using [SendGrid](#). If you don't have a SendGrid account, make sure to follow step 1 from this tutorial [here](#).

Installing Livewire

Once you have your Laravel project ready, the first thing that you need to do is install Livewire.

You can do that using **composer** and running the following command:

```
composer require livewire/livewire
```

This will add the Livewire package to your project.

Adding new Livewire component

In order to create a new Livewire component, you have to run the following **artisan** command:

```
php artisan make:livewire contact-form
```

This will generate your Livewire component class and view:

```
CLASS: app/Http/Livewire/ContactForm.php  
VIEW:  resources/views/livewire/contact-form.blade.php
```

With that, we are ready to include Livewire to our main blade view and also include this new component!

Creating your Blade view

To keep things simple, I will use 1 single view for this example. For my design, I will grab the following component from the Tails project:

[TailwindCSS free template](#)

To create your view, add a file inside your `resources/views` folder called `contact.blade.php`. And paste the following HTML content:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/tailwindcss/1.8.1
0/tailwind.min.css">

  <!-- Include the Livewire styles -->
  @livewireStyles
</head>
<body class="overflow-x-hidden antialiased">

  <!-- CONTACT FORM -->

  @livewire('contact-form')

  <!-- END CONTACT FORM SECTION -->

  <!-- Include the Livewire Scripts -->
  @livewireScripts
</body>
</html>
```

The essential things that you need to keep in mind are the Livewire styles and scripts:

```
    . . .  
    @livewireStyles  
</head>  
<body>  
    . . .  
  
    @livewireScripts  
</body>  
</html>
```

After that, in the contact form section, you will notice that we've added `@livewire('contact-form')` which includes our `contact-form` Livewire view that we created in the previous step.

The next thing that we will do is prepare the actual Livewire view, which will contain our contact form.

Prepare your Livewire view

Once you have prepared your main Blade view by including the Livewire styles and scripts, it is time to work on the contact form itself.

Let's start by opening the `resources/views/livewire/contact-form.blade.php` file and adding the following content:

```
<div>
  <section class="relative py-6 bg-white bg-gray-200 min-w-
screen animation-fade animation-delay">
    <div class="container h-full max-w-5xl mx-auto
overflow-hidden rounded-lg shadow">
      @if ($success)
        <div class="inline-flex w-full ml-3 overflow-
hidden bg-white rounded-lg shadow-sm">
          <div class="flex items-center justify-
center w-12 bg-green-500">
            </div>
            <div class="px-3 py-2 text-left">
              <span class="font-semibold text-
green-500">Success</span>
              <p class="mb-1 text-sm leading-none
text-gray-500">{{ $success }}</p>
            </div>
          </div>
        @endif
        <div class="h-full sm:flex">
          <div class="flex items-center justify-center
w-full p-10 bg-white">
            <form
wire:submit.prevent="contactFormSubmit" action="/contact"
method="POST" class="w-full">
              @csrf
              <div class="pb-3">
                @error('email')
                  <p class="mt-1 text-
red-500">{{ $message }}</p>
                @enderror
                <input wire:model="email"
class="w-full px-5 py-3 border border-gray-400 rounded-lg
outline-none focus:shadow-outline" type="text">
```

```

placeholder="Email Address" name="email" value="{{
old('email') }}" />
    </div>
    <div class="py-3">
        @error('name')
            <p class="mt-1 text-
red-500">{{ $message }}</p>
        @enderror
        <input wire:model="name" class="w-
full px-5 py-3 border border-gray-400 rounded-lg outline-none
focus:shadow-outline" type="text" placeholder="Name"
name="name" value="{{ old('name') }}" />
    </div>
    <div class="py-3">
        @error('comment')
            <p class="mt-1 text-
red-500">{{ $message }}</p>
        @enderror
        <textarea wire:model="comment"
row="4" class="w-full h-40 px-5 py-3 border border-gray-400
rounded-lg outline-none focus:shadow-outline" name="comment"
placeholder="Your message here...">{{ old('comment')
}}</textarea>
    </div>
    <div class="pt-3">
        <button class="flex px-6 py-3
text-white bg-indigo-500 rounded-md hover:bg-indigo-600
hover:text-white focus:outline-none focus:shadow-outline
focus:border-indigo-300" type="submit">
            <span class="self-center
float-left ml-3 text-base font-medium">Submit</span>
        </button>
    </div>
</form>
</div>
</div>
</div>
</section>
</div>

```

Things that you need to keep in mind:

- In our `form` opening tag, we specify the Livewire method, which will be responsible for our form validation logic and sending out our

email: `form wire:submit.prevent="contactFormSubmit"`

- By using `wire:model` in the input fields, we are binding our class properties directly with our Livewire view
- We use the `@error('name')` helper to display any validation errors from our Livewire class

Finally, in the `resources/views/` add a file called `email.blade.php` with the following content:

```
Contact from enquiry from: {{ $name }}  
<p> Name: {{ $name }} </p>  
<p> Email: {{ $email }} </p>  
<p> Message: {{ $comment }} </p>
```

Feel free to play with this mail template and update it so that it matches your needs.

Next, we will implement our Livewire logic.

Prepare your Livewire Logic

With our view all setup, we are ready to configure our Livewire logic.

Edit the `app/Http/Livewire/ContactForm.php` file and add the following content:

```
<?php

namespace App\Http\Livewire;

use Livewire\Component;
use Mail;

class ContactForm extends Component
{
    public $name;
    public $email;
    public $comment;
    public $success;
    protected $rules = [
        'name' => 'required',
        'email' => 'required|email',
        'comment' => 'required|min:5',
    ];

    public function contactFormSubmit()
    {
        $contact = $this->validate();

        Mail::send('email',
            array(
                'name' => $this->name,
                'email' => $this->email,
                'comment' => $this->comment,
            ),
            function($message){
                $message->from('your_email@your_domain.com');
                $message->to('your_email@your_domain.com',
                    'Bobby')->subject('Your Site Contact Form');
            }
        );
    }
}
```

```

        $this->success = 'Thank you for reaching out to us!';

        $this->clearFields();
    }

    private function clearFields()
    {
        $this->name = '';
        $this->email = '';
        $this->comment = '';
    }

    public function render()
    {
        return view('livewire.contact-form');
    }
}

```

Note: Make sure to update the `your_email@your_domain.com` to match the actual email address that you would like to receive the emails to!

Rundown of the script:

- The very first thing that we do is to include the `Mail` facade: `use Mail;`
- Then we specify our public properties, which are the names of the input fields in our contact form:

```

public $name;
public $email;
public $comment;

```

- After that we add the validation rules:

```
protected $rules = [  
    'name' => 'required',  
    'email' => 'required|email',  
    'comment' => 'required|min:5',  
];
```

For more about the Livewire validation, make sure to check out the official documentation [here](#).

- After that, we create our `contactFormSubmit` method, which triggers our validation and, if successful, sends out an email
- As the page will not refresh on submit, we also create a `clearFields` method to clear the form fields

With that, our logic is all setup! Finally, we need to add the routes usual.

Add routes

In order to keep things simple, I will add a single route to return out `contact.blade.php` view:

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

The rest of the routes are all handled by Livewire.

Adding SMTP details

To receive an email, you need to make sure that you add your SMTP details to the `.env` file. You can follow the steps from this [tutorial here](#) on how to do that.

In the end, your `.env` file will look like this your SMTP settings:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.your_smtp_server.com
MAIL_PORT=587
MAIL_USERNAME=your_email_here@your_domain.com
MAIL_PASSWORD=your_password_here
MAIL_ENCRYPTION=tls
```

After you have your SMTP details ready, visit the `/contact` page and test the form!



Conclusion

Using Livewire, you could add some nice interactivity to your site without having to write a single line of JavaScript!

If you like the Livewire project, make sure to star it on [GitHub](#)!

For more information about Livewire, make sure to check out the [official Livewire documentation here](#).

If you are using vanilla Laravel, you could look at this article here on [how to create a contact form with Laravel and SendGrid](#).

For a more detailed tutorial make sure to check this post [here](#)!

[Originally posted here](#)

How To Display HTML Tags In Blade With Laravel

Laravel is a great PHP framework that allows you to use Blade Templates for your frontend.

Blade is a very powerful templating engine provided with Laravel that does not restrict users from using plain PHP code in their views.

Blade view files that use the .blade.php file extension and are typically stored in the resources/views directory.

In this tutorial, you will learn how to display HTML in Blade with Laravel 8!

If you don't already have a Laravel application up and running, I suggest using a DigitalOcean Ubuntu Droplet. You can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Display HTML In Laravel Blade Views

First, you need to make sure your file uses the `.blade.php` file extension and is located in the `resources/views` folder. It should look something like this:



By default you would use the following syntax `{{ $some_variable }}` to echo out the content of a specific variable in Blade. By default the `{{ }}` escapes the HTML tags.

So let's say that you have a blog post with a `$post` collection which has some HTML elements in the `body` property, if you were to use the following:

```
<div>{{ $post->body }}</div>
```

All of the HTML tags would not be rendered as HTML but plain text:



If you don't want your HTML tags to be escaped then you should use `{!! !!}` just like this:

```
<div>{!! $post->body !!}</div>
```

Output:



If you print out both and check the page source, you will see the following output:



```
&lt;h1&gt;Hello DevDojo&lt;/h1&gt;  
<h1>Hello DevDojo</h1>
```

In the first case, you can see how the HTML elements were escaped, and in the second case where we use `{!! !!}` we got the actual HTML tags.

It is more secure to use `{{ }}` as it will strip out any unwanted tags, but there are still times where you might need to use `{!! !!}`.

Conclusion

Blade templating is beneficial and can make your life easier.

If you want to learn more about Blade Templating, check out the official Laravel Documentation:

<https://laravel.com/docs/8.x/blade>

[Originally posted here](#)

How to Set a Variable in Laravel Blade Template

The Blade templating engine has been a real game-changer for me. Blade makes it working with PHP and HTML a breeze. It allows you to use plain PHP code directly in your template.

In most cases, you will pass your variables from your controller to your Blade views, but you might want to set a variable directly in your Blade view in some cases.

In this tutorial, **you will learn how to set a variable in your Laravel Blade Template directly!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Returning a variable from a controller

You could pass variables to the `view` helper function directly from your controller or route in some cases.

The syntax is the following:

```
return view('dashboard', ['name' => 'Bobby']);
```

The above will return the `dashboard.blade.php` view, and a variable called `$name` with the value of `Bobby` will be available so you could call it like this in your view:

```
<p> Welcome on board {{ $name }}!</p>
```

By default, this will escape all HTML tags, but in case that you needed to render HTML tags, you could follow the steps on how to do that here:

- [How To Display HTML Tags In Blade With Laravel](#)

Alternatively, you could use the `compact` helper function to pass multiple variables from your controller to your view:

```
$name = 'Bobby';  
$twitter = 'bobbyiliev_';  
  
return view('dashboard', compact('name', 'twitter'));
```

Now that we've got this covered, here is how you could define a variable directly in your Blade template rather than passing it from your controller.

Setting variables in Laravel Blade Template

As you could use plain PHP directly in your Blade template, all that you need to do in order to define a new variable is to use the following syntax in your view:

```
@php
    $name = "Bobby";
@endphp

<p>Hi there {{ $name }} </p>
```

Thanks to the `@php` directive, you could actually write pure PHP directly in your view. Even though that, it would probably be better to keep any complex logic in your controller. Still, in some cases, it is super handy to have this as an option.

Alternatively, you could actually use general PHP opening and closing tags, but I prefer to stick to the Blade PHP block instead:

```
<?php
    $name = "Bobby";
?>
```


Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)

How to limit the result with Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

Laravel provides a lot of handy methods that you could use to work with your Eloquent collections. More often than not, when getting some results from your database, you would want to order them based on specific criteria.

In this tutorial, **you will learn how to limit the results with Laravel Eloquent!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

We will use a model called **Post** as an example in this tutorial.

Limiting the result with pure SQL

In case that you were using pure SQL and wanted to limit the number of records returned by your query, you could use the **LIMIT** keyword, followed by the number of records that you would want to limit your result set to.

For example, let's say that we wanted to get only 5 posts from our posts table. The query would look like this:

```
SELECT * FROM posts LIMIT 5;
```

Now let's go through a couple of ways to do that with Laravel Eloquent.

The `limit()` method

If we were to convert the above query to Eloquent, the syntax would look like this:

```
$posts = Post::limit(5)->get();
```

If you were to use [the `toSql` method to get the actual query](#) and `dd()` as follows:

```
dd(Post::limit(5)->toSql());
```

You would get the following output:

```
"select * from `posts` limit 5"
```

You could also specify an offset in case that you needed to with the `offset()` method.

An important thing that you need to keep in mind is that you could use the `limit()` method only with Eloquent, it does not work with collections.

Alternatively, you could use the `take()` method, which works with both Eloquent and collections.

The `take()` method

I personally prefer to use the `limit()` method directly with Eloquent, but in some cases, it is handy to have the `take()` method on hand as well.

The syntax is more or less the same:

```
$posts = Post::take(20)->get();
```

Or if you already have your collection in place, you could again use the `take()` method as follows:

```
$posts = Post::get();  
$posts->take(20);
```

The `paginate()` method

In case that you are working on pagination, there is no need to use any of the above methods. Instead, you could just use the fantastic `paginate()` method that Laravel provides.

In your controller, you could specify the number of records that you would want to see on one page as follows:

```
$posts = Post::paginate(30)
```

And then, in your Blade view, you could use a `foreach` loop to loop through your posts and show the available pages as follows:

```
@foreach($posts as $post)
    <p>{{ $post->title }}</p>
@endforeach

{{ $posts ->links() }}
```

In the background, Laravel takes care of everything.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)

How to Select Specific Columns in Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

In some specific cases you might not want to get all of the columns from a specific table but just one or some of them.

In this tutorial, **you will learn how to select specific columns in Laravel eloquent!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Select specific columns with SQL only

Let's say that we had a table called users with the following columns:

name, username, age, email, phone.

If you wanted to get all of the information from that table with only SQL the query would look like this:

```
SELECT * FROM users WHERE username = 'bobbyiliev';
```

However, if you wanted to get only, let's say, the age of the user, you would need to change the * symbol with the name of the column:

```
SELECT age FROM users WHERE username = 'bobbyiliev';
```

If you wanted to get two columns, you would divide them by a comma:

```
SELECT name,age FROM users WHERE username = 'bobbyiliev';
```

Now that we know how to do this with SQL only, let's learn how we can achieve this with Laravel.

Select specific columns with Laravel Eloquent

Let's say that we already have our `User` model in place. To get all of the columns from the `users` table, we would use the following:

```
$user = User::where('username', 'bobbyiliev')->get();
```

However, if you wanted to get only a specific column, you could pass it as an argument to the `get()` method.

The argument needs to be an **array containing a list of the columns that you want to get**:

```
$user = User::where('username', 'bobbyiliev')->get(['name']);
```

You could do the same with the `first()` and `all()` methods as well:

```
$user = User::where('username', 'bobbyiliev')->first(['name', 'age']);
```

If you wanted to get the value of a single column, you could use the following:

```
User::where('username', 'bobbyiliev')->first(['name'])->name;
```

However, I would personally first check if there are any entires that match the condition and then get the property.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about SQL in general make sure to check out this [free introduction to SQL eBook](#) here.

[Originally posted here](#)

How to get the Laravel Query Builder to Output the Raw SQL Query

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

When troubleshooting problems with the Laravel query builder, you might want to see the actual SQL query that is being executed for debugging purposes.

Here are a couple of quick ways that you could use to quickly get the query Builder to output its raw SQL query as a string!

Before you begin, you need to have Laravel installed.

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

You would also need to have a model ready that you would like to use. For example, I have a small blog with a `posts` table and `Post` model that I would be using.

The `toSql()` method

When I first started using Laravel, I was amazed by how convenient and handy the Eloquent ORM and how much it speeds up the whole development process.

You can use the `toSql()` method to get the plain SQL representation of your query.

As an example, let's say that we want to get all posts that are published. If we had to write this in pure SQL, it would look like this:

```
SELECT * FROM posts WHERE status='published';
```

And here is how we would do this with Laravel:

```
$posts = Post::where('status', 'published')->get();
```

In order to use the `toSql()` method, you could just change the `get()` part in the above query with `toSql()`:

```
$posts = Post::where('status', 'published')->toSql();
```

Then you could use `dd()` to dump the output on the screen:

```
dd($posts);
```

You will see the following output:



```
"select * from `posts` where `status` = ?"
```

As you can see the binding in this case is represented as `?`, to get the bindings you could use the `getBindings()` method:

```
$posts = Post::where('status', 'published')->getBindings();  
dd($posts);
```

Laravel Debugbar

I love the `toSql()` method, but in some cases having to manually add this across your application while debugging could be intimidating.

This is why, as an alternative, you could use the [Laravel Debugbar Package](#).

It would give you a really nice bar with a lot of useful debug information.

To add the package to your application, you could use the following composer command:

```
composer require barryvdh/laravel-debugbar --dev
```

Note the `--dev` flag, it is crucial as you don't want to have the debug bar available in your production environment.

For more information on how to use the debug bar, make sure to check out the documentation [here](#).

Conclusion

The `toSql()` method could be very handy when troubleshooting problems with your eloquent queries!

If you are just getting started with Laravel, I would recommend going through [this Laravel introduction course](#).

[Originally posted here](#)

How to Get the Last Inserted Id Using Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

In this tutorial, **you will learn how to get the Last Inserted Id Using Laravel Eloquent!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to [get free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Last Inserted Id Using Laravel Eloquent

Let's say that we have a newsletter sign up form where people could sign up, and then after successful signup, we want to get the ID of the entry.

To achieve that, you would need to already have a Model and a controller. Here is an example method that we would use:

```
public function signUp() {  
  
    $subscriber = new Subscriber();  
    $subscriber->name = 'Bobby';  
    $subscriber->email = 'hello@bobbyiliev.com';  
  
    $subscriber->save();  
}
```

Once the new subscriber is saved, we can simply use `return $subscriber->id` to retrieve the ID.

So the complete method will look like this:

```
public function signUp() {  
  
    $subscriber = new Subscriber();  
    $subscriber->name = 'Bobby';  
    $subscriber->email = 'hello@bobbyiliev.com';  
  
    $subscriber->save();  
  
    return $subscriber->id; // this will return the saved  
    subscriber id  
}
```

In case that you are building an API, you could return the response in

JSON with:

```
return response()->json(array('success' => true,  
    'last_insert_id' => $subscriber->id), 200);
```

Conclusion

For more information and useful tips on Laravel Eloquent, make sure to check this tutorial here:

- [Laravel Eloquent Make](#)

I strongly suggest checking out the official Laravel Documentation:

- [Laravel Eloquent](#)

[Originally posted here](#)

How to Order the Results of `all()` in Laravel Eloquent

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

Laravel provides a lot of handy methods that you could use to work with your Eloquent collections. More often than not, when getting some results from your database, you would want to order them based on specific criteria.

In this tutorial, **you will learn how to order the results of `all()` in Laravel Eloquent!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

We will use a model called **Post** as an example in this tutorial.

Ordering the results on a query level

Before we cover how to sort the results on a collection level, let's first briefly cover how we would sort the results on a query level.

Let's say that we wanted to sort all of our posts by name. When getting the results from the database, we could use the `orderBy` method followed by the `get` method:

```
$posts = Post::orderBy('name')->get();
```

If you added the `toSql()` method instead of the `get()` and used `dd` to print out the output as follows:

```
$posts = Post::orderBy('name')->toSql();  
dd($posts);
```

The actual query, you would get the following output:

```
"select * from `posts` order by `name` asc";
```

So as you can see, the `orderBy` method actually represents the `ORDER BY` function in SQL.

By default, Eloquent will order by ascending, but you could also order by descending by using the `orderByDesc()` method:

```
$posts = Post::orderByDesc('name')->get();
```

To learn more about SQL, you can check out this [free eBook on how to get started with SQL here](#).

However, if you use `all()` rather than `get()` you would first get all of the entries from your Database and store them in a collection. And once you have the collection ready, you would need to use the available [Laravel collection methods](#) rather than the eloquent methods.

Ordering the results on a collection level with `all()`

To summarize the above, you can use the Eloquent methods while building your query, this would be a more efficient way of sorting your results. But you could also use the available collection methods to order the contents of a collection too.

This means that at a collection level, you could use the available `sortBy` method rather than the `orderBy` method, which is available at the query level.

Similar to the Eloquent methods, you can sort by ascending order using the `sortBy` method and sort by descending order using the `sortByDesc` method:

- Ascending:

```
$posts = Post::all();  
$posts->sortBy("name");
```

- Descending:

```
$posts = Post::all();  
$posts->sortByDesc("name");
```

Alternatively, you could chain the method too:

```
// Ascending
$posts = Post::all()->sortBy("name");

//Descending
$posts = Post::all()->sortByDesc("name");
```

Whenever I have the chance, I would use the `orderBy` method, but it is still quite handy to be able to easily order your results on a collection level.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about PHP in general, make sure to check out this [free PHP basics course here](#).

[Originally posted here](#)

How to fix Laravel Unknown Column 'updated_at'

The Eloquent ORM included with Laravel provides you with an easy way of interacting with your database. This simplifies all CRUD (Create, read, update, and delete) operations and any other database queries.

As described in the official [Laravel documentation](#), Eloquent expects the `created_at` and `updated_at` columns to exist on your model's corresponding database table.

However, in some specific cases you might not have those two tables in place.

In this tutorial, **you will learn how to fix the `Laravel Unknown Column 'updated_at'` error by disabling the Laravel timestamps for a specific model!**

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers!

If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Disable the timestamp columns in your Model

In order to disable the timestamp columns for a specific model, all that you would need to do is define the a public property called `$timestamps` and set this to false.

That way, when you try to update your model, you will not get the `Unknown Column 'updated_at'` error.

Let's have a model called `Video` as an example:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Video extends Model
{
    // Disable the model timestamps
    public $timestamps = false;
}
```

With that, Eloquent will no longer be looking for the `created_at` and the `updated_at` columns.

Change the name of the timestamp tables

In order cases, you might have the timestamp columns present, but with different names, so you would not want to disable the `timestamps` but instead specify the correct names of the tables.

To do so, all that you have to do is specify the following two constants in your Model:

```
<?php

class Video extends Model
{
    const CREATED_AT = 'creation_date';
    const UPDATED_AT = 'updated_date';
}
```

Adjust the constants accordingly depending on your needs

That way, Eloquent will be looking for a `updated_date` table rather than `updated_at` as default.

Conclusion

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Getting started with Laravel](#)

If you want to learn more about SQL in general make sure to check out this [free introduction to SQL eBook](#) here.

[Originally posted here](#)

How to Define Custom ENV Variables in Laravel

If you have some experience with Linux, you are probably quite familiar with environment variables. In Linux, you could check the available environment variables with the `printenv` command. A way to define environment variables in Linux would be to use the `export` command followed by the variable that you want to define, for example: `export name=DevDojo`.

In this tutorial, you will learn a couple of ways of defining Laravel variables and accessing them in your application!

Before you start, you would need to have a Laravel application up and running.

I will be using a DigitalOcean Ubuntu Droplet for this demo. If you wish, you can use my affiliate code to get [free \\$100 DigitalOcean credit](#) to spin up your own servers:



If you do not have that yet, you can follow the steps from this tutorial on how to do that:

- [How to Install Laravel on DigitalOcean with 1-Click](#)

Or you could use this awesome script to do the installation:

- [LaraSail](#)

Defining a variable in `.env`

The `.env` file holds your env variables for your current environment. It is powered by the [DotEnv](#) Library.

As the `.env` file often holds sensitive information like API keys or database credentials, you should never commit it to Git and never push it to GitHub.

In order to define a new environment variable, you could use the following syntax: `VAR_NAME=YOUR_VALUE`. For example, let's say that you wanted to define your GitHub API key in there, the syntax would be the following:

```
GITHUB_API_KEY=your_api_key_here
```

In case that you have spaces in the value, it is best to use quotes:

```
NAME="Bobby Iliev"
```

Accessing the env variable with `env()`

Once you've specified your environment variable in your `.env` file, you could then retrieve it in your application with the following helper function:

```
env( 'NAME' )
```

The `env` helper function allows you to pass a default value in case that the env variable is not defined or does not have a value specified in the `.env` file:

```
env( 'NAME' , 'DevDojo' )
```

In the above example, if the `NAME` env variable is not defined, it will take the default value of `DevDojo`.

Accessing the env variable with `config()`

If you are [creating a new Laravel Package](#) or if you are building some specific functionality, it is pretty handy to define different configuration values in a configuration file stored in the `config` folder.

For example, let's say that we are building functionality that consumes the DigitalOcean API. What we can do is create a new file inside the `config` folder called `digitalocean.php`:

```
config/digitalocean.php
```

The content of the file will be a single return statement that returns an array with different env configuration values:

```
<?php
return [
    'do_key' => env('DIGITALOCEAN_API_KEY',
    'default_value_here'),
];
```

With that we would also need to define the actual value of the `DIGITALOCEAN_API_KEY` variable inside the `.env` file just as we did in the previous step:

```
DIGITALOCEAN_API_KEY='your_digitalocean_api_key_here'
```

After that, by using the `config` helper method, you could access the `do_key` value with the following syntax:

```
config('digitalocean.do_key')
```

Rundown of the syntax:

- `config('')`: First, we call the config helper
- Then, we specify the name of the configuration file inside the `config` folder. As we named the file `digitalocean.php` we need to just use `digitalocean`
- Finally separated with a dot we specify the value that we want to call

With that, you could also use the `php artisan config:cache` command to cache your env variables and give your website a slight speed boost.

Important thing to keep in mind is that if you run the `config:cache` command, you will only be able to use the `env` helper inside configuration files in the `config` folder. And also, you need to have the `php artisan config:cache` command executed on every deployment.

Conclusion

Now you know how to define custom variables in your Laravel application! For more information about the Laravel configuration, make sure to check out the official documentation here:

<https://laravel.com/docs/8.x/configuration>

If you are just getting started with Laravel, make sure to check out this introduction course here:

[Introduction to Laravel](#)

[Originally posted here](#)

How to fix 'Please provide a valid cache path' error in Laravel

The other day, I was setting up a local development environment for an existing project when I encountered the '**Please provide a valid cache path**' error.

As the error itself is not very descriptive and could leave you thinking that something is wrong with your configuration, I've decided to write a short post on how to get this sorted out!

Creating the cache folders

By default Laravel stores its temporary files at the `storage/framework` directory. In there there should be 3 more folders: `sessions`, `views` and `cache`.

That is where Laravel will store its generated blade views, cache and sessions.

Without those 3 directories you might see the error above.

To create the directories you could run the following command.

- First use the `cd` command to access the location of your Laravel application and then run:

```
mkdir storage/framework/{sessions,views,cache}
```

The above command will create the 3 directories in question.

Clearing the cache

After that you could run the following commands to clear your cache and make sure that it works as expected:

```
php artisan cache:clear  
php artisan config:clear  
php artisan view:clear
```

You should see the following output:

```
Application cache cleared!  
Configuration cache cleared!  
Compiled views cleared!
```

Conclusion

This is pretty much it! I hope that this post helps you to solve the problem in question!

[Originally posted here](#)

How to check your exact Laravel version

As of Laravel 6.x, it now follows [Semantic Versioning](#), which essentially means that the major framework releases are released every six months (February and August), and minor and patch releases may be released as often as every week.

With that in mind your Laravel version could often change, so here are a couple of ways for you to find out the exact Laravel version that you have.

Check your Laravel version with artisan

Laravel comes with a command-line interface called Artisan.

Artisan provides a huge number of commands that help you manage and build your Laravel application.

In order to get your exact Laravel version you can just run the following command in the directory of your Laravel project:

```
php artisan --version
```

The output that you would get will look like this:

```
Laravel Framework 8.12.0
```

In the example above, we can see that we are Running Laravel **8.12.0**.

For more information, you can find the [official Artisan documentation here](#).

Check your Laravel version via your text editor

In case you do not have a terminal open, you might want to check your Laravel version via your text editor instead.

To do that, open the following file:

```
vendor/laravel/framework/src/Illuminate/Foundation/Application.php
```

Once you have the file open, just search for **VERSION**. The first constant in the **Application** class would be the Laravel version:

```
/**
 * The Laravel framework version.
 *
 * @var string
 */
const VERSION = '8.12.0';
```

Conclusion

Those were just 2 quick was of **checking your exact Laravel version**.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

Also, you could have a look at this course here on [how to build a blog using Laravel and Voyager](#). It shows you how to deploy and configure your own server with [DigitalOcean](#) as well.

[Originally posted here](#)

Contact Form with Voyager and Laravel

This is an easy step by step guide on how to add a **contact form** to your **Voyager** application.

Before you begin you need to make sure that you have **Laravel** and **Voyager** up and running and that you've sorted out your database connection.

If you are not sure how to do that you can follow the steps here on how to deploy **Laravel** App on **Digital Ocean Ubuntu** server with **Voyager**:

[**devdojo.com**](https://devdojo.com)

If you don't have a Digital Ocean account yet, you can sign up for Digital Ocean and get **\$50 free** credit via this link here:

[**digitalocean.com**](https://digitalocean.com)

Steps

The steps that we will take are:

- Create a Table with a Model
- Configure the Model
- Create the Routes
- Create and Configure the Controller
- Create the Views
- Configure the Email settings
- Test the Form

At the end we would be receiving emails sent though our contact form and we would also be able to see those entires in our Voyager admin panel

Configuration

Let's start with our configuration! I have a plain Laravel and Voyager installation that I'll be using. You can also implement that on an existing site, but make sure to **back everything up** before following this guide!

1. Create a Table with a Model

Go to your Voyager admin -> Tools -> Database -> Create New Table

Then go ahead and create a BREAD as well so that we could then see our contact form entries via our Voyager admin area:

You can use the default settings, just scroll to the bottom of the page and hit Submit.

2. Configure the Model

After that we need to make a slight change to the new Contact model that we've created. In the **app/Contact.php** file add the following lines:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Contact extends Model
{
    public $table = 'contact';
    public $fillable = ['name', 'email', 'message'];
}
```

3. Create a route

Add the following to your **routes/web.php** file:

```
Route::get('contact', 'ContactController@contact');
Route::post('contact',
    ['as'=>'contact.store', 'uses'=>'ContactController@contactPost'
    ]);
```

4. Create and Configure your Controller

You can run the following artisan command to create your Contact Controller:

```
php artisan make:controller ContactController
```

Then in the **app/Http/Controllers/ContactController.php** file add the following code:

```

<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use App\Contact;
use Mail;
class ContactController extends Controller
{
    public function contact()
    {
        return view('contact');
    }

    /** * Show the application dashboard. * * @return
    \Illuminate\Http\Response */
    public function contactPost(Request $request)
    {
        $this->validate($request, [ 'name' => 'required', 'email'
=> 'required|email', 'message' => 'required' ]);
        Contact::create($request->all());

        Mail::send('email',
            array(
                'name' => $request->get('name'),
                'email' => $request->get('email'),
                'bodyMessage' => $request->get('message')
            ), function($message)
            {
                $message->from('bobby@bobbyiliev.com');
                $message->to('bobby@bobbyiliev.com',
'Bobby')->subject('Bobby Site Contact Form');
            });
        return back()->with('success', 'Thank you for contacting
me!');
    }
}

```

5. Create the View

With all that out of the way we can now go ahead and create our view.

In your **resources/views/** create a file called `contact.blade.php` and add this code:

```
<!DOCTYPE html>
<html>
<head>
<title>DevDojo - Bobby Iliev - Contact Form</title>
<link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/boot
strap.min.css">
</head>
<body>

<div class="container">
<h1>Contact Me</h1>

@if(Session::has('success'))
    <div class="alert alert-success">
        {{ Session::get('success') }}
    </div>
@endif

{!! Form::open(['route'=>'contact.store']) !!}

<div class="form-group {{ $errors->has('name') ? 'has-error' :
'' }}">
    {!! Form::label('Name:') !!}
    {!! Form::text('name', old('name'), ['class'=>'form-
control', 'placeholder'=>'Name']) !!}
    <span class="text-danger">{{ $errors->first('name') }}</span>
</div>

<div class="form-group {{ $errors->has('email') ? 'has-error'
: '' }}">
    {!! Form::label('Email:') !!}
    {!! Form::text('email', old('email'), ['class'=>'form-
control', 'placeholder'=>'Email']) !!}
    <span class="text-danger">{{ $errors->first('email') }}</span>
</div>

<div class="form-group {{ $errors->has('message') ? 'has-
error' : '' }}">
    {!! Form::label('Message:') !!}
    {!! Form::textarea('message', old('message'),
['class'=>'form-control', 'placeholder'=>'Message']) !!}
    </div>
```

```

<span class="text-danger">{{ $errors->first('message')
}}</span>
</div>

<div class="form-group">
<button class="btn btn-success">Send mail</button>
</div>

{!! Form::close() !!}

</div>

</body>
</html>

```

Do not forget to install the **laravelcollective/html** package otherwise your form would not work:

```
composer require laravelcollective/html
```

This is a simple **Bootstrap** form that would look something like this, of course you can customize that so that it meets your requirements:

Let's take care of the email view as well. This would be the actual email that would be send out each time someone uses our form. Again in the **resources/views/** add a file called email.blade.php with the following content:

```

Contact from enquiry from: {{ $name }}
<p> Name: {{ $name }} </p>
<p> Email: {{ $email }} </p>
<p> Message: {{ $bodyMessage }} </p>

```

6. Configure the Email settings

I'll be using **SMTP** to send out the emails, so what I would need to do is to configure the basic **Laravel** mail function through PHP **Artisan**:

```
php artisan make:mail smtp
```

Then in your **.env** file specify your SMTP settings:

```
MAIL_DRIVER=smtp  
MAIL_HOST=smtp.gmail.com  
MAIL_PORT=587  
MAIL_USERNAME=your_email_here@gmail.com  
MAIL_PASSWORD=your_password_here  
MAIL_ENCRYPTION=tls
```

7. Test the Form

With all that sorted out we are ready to test our form!

You would also be able to see your emails via your **Voyager** Admin panel:

Conclusion

That's pretty much it! It is pretty straight forward but always make sure to backup your website before making any changes :)

[Originally posted here](#)

How to filter routes in Laravel

When your application grows the routes becomes difficult to read when you execute `php artisan route:list`

There would be nice if there is some flag to filter a route, well this is not available, but as it is a terminal command we can use the `grep` param to filter the output from the terminal.

Steps

In the terminal:

- Inside your project directory `cd <your-project>`
 - Show or hide columns
 - Search by term
-

Filter routes by term

1. Execute the command with columns flag

The output of the route list command returns a lot of useful data, but usually we only need the uri, the method and the name.

Laravel provide us with a `columns` flag to show only the columns that are necessary.

```
php artisan route:list --columns=uri,method,name
```

2. Filter a route

If you are using the route names convention like resource probably your project have a route like this: `users.index`.

If your app is serving an API it could have a route name like this: `api.users.index`.

And the app could probably have routes like:

```
* `api.users.index`  
* `api.users.show`  
* `api.users.update`  
* `api.users.store`  
* `api.users.delete`
```

You can filter partially or with a complete route string, in the example below it would return all the routes with the prefix `api.users`:

```
php artisan route:list --columns=uri,method,name |grep  
api.users
```

Conclusion

This was just a quick example of **how to filter some group a routes between all your routes**.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

How to add a gravatar profile picture in Laravel

Gravatar is a service that allows to use their profile images with external api calls, it is another way to handle user profile pictures instead of storing those in the database.

Here a little guide of how we can implement this feature.

Steps

- Create a trait to add gravatars to any model
 - Show a user avatar in blade templates
 - Add a gravatar as the default profile picture in jetstream
-

Create a trait to handle the gravatar

You can add a user method or if other models would use the gravatar image you can separate the code into a trait:

```
namespace App\Traits;

trait HasGravatar {

    /**
     * The attribute name containing the email address.
     *
     * @var string
     */
    public $gravatarEmail = 'email';

    /**
     * Get the model's gravatar
     *
     * @return string
     */
    public function getGravatarAttribute()
    {
        $hash =
md5(strtolower(trim($this->attributes[$this->gravatarEmail])))
        ;
        return "https://www.gravatar.com/avatar/$hash";
    }
}
```

Now any model can use the trait as User model:

```
use App\Traits\HasGravatar;  
  
class User extends Model  
{  
    use HasGravatar;  
}
```

Render an image in a blade view:

Now you only need to pass a user to the view and use it as below:

```

```

Use avatars with Jetstream

If you are working with Jetstream there is a simple way to change the default initial letters to a gravatar.

Just override the method that was originally added to the User model by this trait `Laravel\Jetstream\HasProfilePhoto`

Instead of this:

```
/**
 * Get the default profile photo URL if no profile photo has
 * been uploaded.
 *
 * @return string
 */
protected function defaultProfilePhotoUrl()
{
    return 'https://ui-
avatars.com/api/?name='.urlencode($this->name).'&color=7F9CF5&
background=EBF4FF';
}
```

In your User model you can override with this:

```
/**
 * Get the default profile photo URL if no profile photo has
 * been uploaded.
 *
 * @return string
 */
protected function defaultProfilePhotoUrl()
{
    return $this->getGravatarAttribute();
}
```

Conclusion

There are is a simple and flexible way to handle how to **add profile avatar images with gravatar in Laravel**.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

How to append a mutator to a collection

Mutators in Laravel provide us an easy way to transform a model property.

In this example a `Post` model implement a mutator based on `created_at` timestamps.

```
class Post
{
    protected $casts = [
        'created_at' => 'date',
    ];
    public function getPublishedAttribute()
    {
        return $this->created_at->diffForHumans();
    }
}
```

Append a mutator on a single model

Eloquent provides a method to append a mutator to a single model so you can return a single model with a mutator as it is another model property:

```
public function show(Post $post)
{
    return view('post.show', ['post' =>
$post->append('published_at')]);
}
```

Usage

```
$post->published_at;
```


Append a mutator to a collection

Ok, so right now we are able to append a mutator to a single model, if you want to append a mutator to a whole collection, you could be tempted to add the mutator to your model `append` property.

But, most of the time it is unnecessary and even a not performant action to append the mutator and of course calculate it, on every instance of a model or a collection.

Another action that could be potentially a bad practice would be to return only you eloquent collection and force a blade template to append every mutator on the fly, this would lead to an $N + 1$ problem and should be avoided as much as possible.

So instead of append the mutator all the time you can use the collection high order proxy to append the mutator for all items in the collection:

```
public function index()
{
    $posts = Post::query()->get();
    $posts->append('published_at');
    return view('posts.index', ['posts' => $posts]);
}
```

Now in your view you can iterate over your eloquent collection and get the mutator as it is another model property without $n+1$ problems and only append the mutator on this specific controller method.

```
@foreach($user as $users)
    <p>{{ $user->published_at }}</p>
@endforeach
```

Conclusion

This was just a quick example of **how to append a mutator to a laravel collection** and what could lead to a bad practice and n+1 query problems and how to avoid it.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

How to use env variables in javascript

In some project we are need to add some configuration variables from `.env` file to a javascript file like `webpack.mix.js` file.

Imagine that you need use the key `APP_URL` from `.env` file, the new mix version support the same `process` syntax as VueCli and other tools.

An example

Here an example where it configures Laravel mix to use the browserSync feature to get hot reload in our application:

ENV file

```
APP_URL=http://localhost
```

webpack.mix.js

Instead of hard code the string, you can use the `process` to use only one file to set all your configuration variables:

```
if (! mix.inProduction()) {  
    mix.browserSync({  
        proxy: process.env.APP_URL,  
        open: false,  
    })  
}
```

Conclusion

This was just a quick example of **how to use env variables in javascript files**.

It already has an example of how to use it to configure Laravel Mix `browserSync` feature to get hot reload, you can use this `process.env.<key>` syntax on any javascript file that would be processed by mix/webpack to get easily configuration values.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

How to customize forgot password notification

There are some cases where the default forgot password email template does not fit for your project, in these cases you can create a new custom notification and override how the **User** model would use it.

Create a custom notification

```
php artisan make:notification CustomResetPasswordNotification
```

You can make easier your development by extends from the default Notification class:

```
class ResetPasswordNotification extends Notification
{
    // custom code
}
```

Override the notification

```
class User
{
    public function sendPasswordResetNotification($token)
    {
        $this->notify(new
CustomResetPasswordNotification($token));
    }
}
```


Conclusion

This was just a quick example of **how to customize forgot password notification..**

Of course as anything in Laravel there are a lot of ways to make this changes, but here is a very simple approach to use a custom notification.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

How to fire an event when user is logged in

Laravel provides some events by default to handle Authentication flow natively, here an example with the login event.

The EventServiceProvider allow to tight events with listeners, in this case a custom listener to an event that the framework provides.

Create a custom service provider

```
php artisan make:listener UpdateUserLastLoginListener
```

Here you can add any logic, in this example the listener would update the new login timestamp in a `last_login_at` column in `User` model.

```
public function handle($event)
{
    $event->user->update('last_login_at' => now());
}
```

Then we need to tight the new listener to the Login event, this event already exists and is native from the framework, so in `EventServiceProvider.php` file:

```
protected $listen = [
    Login::class => [
        UpdateUserLastLoginListener::class,
    ],
];
```

And now everytime a user logged in this listener would be in charge to update the `last_login_at` column for you, without changing any code in the authentication code.

Conclusion

This was just a quick example of **how to fire an event when a user is logged in..**

This was just a simple example of how you can take an advantage of default events that already exists on the framework, there are more methods like, **Registered**, **Verified** and **Logout** events, where you can tight your own logic by using laravel listeners.

If you are new to Laravel, I would recommend checking out this [Introduction to Laravel 7](#) free course by **DevDojo**.

Scaling Laravel App with Multiple Databases

In this tutorial I'll show you how you could horizontally scale up your Laravel application by adding multiple database instances.

In case you are not familiar with the term '**horizontal scaling**', it simply means that you scale up by adding more machines to your setup where '**vertical scaling**' means that you scale by adding more power (CPU, RAM) to your current virtual machine.

For this demo I will setup the following:

- 1 CentOS server for my Apache, PHP and Laravel setup
 - 2 MySQL servers with Master-Slave replication
 - Configure Laravel to work with the Master-Slave setup
-

Prerequisites

- You would need 3 **CentOS** servers.

As I'll deploy the setup on **Digital Ocean**, if you do not already have an account you can use my referral link to get a **free \$50 credit** that you could use to deploy your virtual machines and test this setup yourself on a few **Digital Ocean servers**:

[Digital Ocean \\$100 Free Credit](#)

Steps

The steps that we would take in this tutorial are:

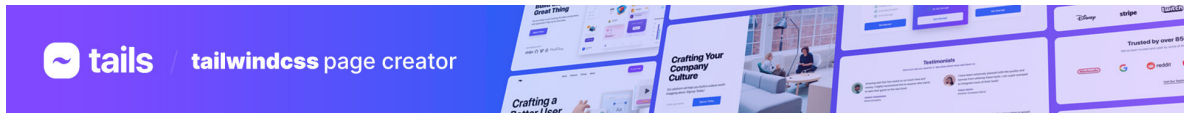
- Deploy 3 servers on Digital Ocean - 1 Web Server and 2 Database Servers
- Install and Configure our Web server - Install Apache, PHP and Laravel
- Install and configure our database servers along with our MySQL replication
- Configure Laravel to work with the Master-Slave MySQL setup
- Install the Voyager admin panel on the new setup

That's pretty much it! At the end we would have something like this:



1. Deploy 3 servers on Digital Ocean

Once you've got your [free \\$50 Digital Ocean credit](#) go to your **Digital Ocean** dashbord then hit the '**Create Droplet**' button and simply deploy **3 CentOS 7.x** servers. Make sure that you keep a good naming convension, here's an example:



[Checkout our latest product - the ultimate tailwindcss page creator](#) ☐

After that hit the 'Create' button. It would take less than 30 seconds for the VMs to be deployed.

Note that your droplet IPs!

2. Install and Configure our Web server

Let's begin with our web server configuration. Get your web server's IP address via your Digital Ocean control panel and SSH to the droplet.

2.1 Prepare our repositories

We would start by adding a few repositories which contain updated packages. This would allow us to install PHP 7.2. The two repos are called REMI and EPEL. To do that we just need to run the following commands in order to install and enable those repositories:

```
yum install epel-release
yum install
http://rpms.remirepo.net/enterprise/remi-release-7.rpm
yum install yum-utils
yum-config-manager --enable remi-php72
```

Then update your system to make sure that everything is up to date before we begin:

```
yum update -y
```

2.2 Install Apache

We would really quickly deploy Apache and PHP. You can do that by running the following commands:

```
yum install httpd -y
```

Then start, enable and check the status of your Apache installation:

```
systemctl start httpd
systemctl enable httpd
systemctl status httpd
```

The output that you would see should look something like this:



2.3 Install PHP

After that install PHP:

```
yum install php php-zip php-mysql php-mcrypt php-xml php-
mbstring
```

2.4 Install Composer

We would use Composer to install Laravel and its dependencies:

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/bin/composer
chmod +x /usr/bin/composer
```

After that if you just run **composer** you would see a similar output:



2.5 Install Laravel

Once we have composer we can simply run this command here to install Laravel:

```
cd /var/www

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

It might take a while for the installation to complete. Once it's done we can proceed with the final Apache configuration changes

2.6 Configure Apache

The best approach would be to create a virtual host for your domain name. I will be using **devdojo.bobbyiliev.com**

To do that you just need to add the following to the bottom of the file in the `/etc/httpd/conf/httpd.conf` file:

NOTE: change **bobbyiliev** with the path to your project!

```
<VirtualHost *:80>
    ServerName devdojo.bobbyiliev.com
    DocumentRoot /var/www/bobbyiliev/public

    <Directory /var/www/bobbyiliev>
        AllowOverride All
    </Directory>
</VirtualHost>
```

Then make sure that your permissions are set correctly:

```
chown -R apache.apache /var/www/bobbyiliev
chmod -R 755 /var/www/bobbyiliev/storage
chcon -R -t httpd_sys_rw_content_t /var/www/bobbyiliev/storage
setsebool httpd_can_network_connect_db 1
```

Finally run a config test and restart Apache:

```
apachectl -t  
systemctl restart httpd
```

Then you should be able to see your Laravel app by visiting your domain name:



Install and configure our database servers along with our MySQL replication

Note: if you want to skip all of the tedious MySQL configuration, you could try out the new [DigitalOcean Managed databases](#)!

We will start with the MySQL installation, the easiest way to do that is to just run these commands here on both of your database servers:

```
yum localinstall
https://dev.mysql.com/get/mysql57-community-release-el7-11.noa
rch.rpm
yum install mysql-community-server
```

Then start your MySQL instance and make sure that it's enabled:

```
systemctl enable mysqld
systemctl start mysqld
```

After that to get your MySQL password just run:

```
grep 'temporary password' /var/log/mysqld.log
```

Then secure your MySQL installation and set your root MySQL password:

```
mysql_secure_installation
```

Note that you need to do this on both servers!

Once you have your MySQL running and you've got your passwords it's

time to proceed with the replication setup.

Here's a really quick guide on how to install and configure your MySQL servers with MySQL Master to Slave replication:

CentOS MySQL Master-Slave Replication - Step by Step

Once you have your SQL servers and replication ready go ahead and create your database and database user that we would be using for our Laravel application on your master database server.

Verify that you are on the master:

```
mysql> show master status;
```

You should see something like this:



After that go ahead and create your laravel database and user:

```
mysql> CREATE DATABASE laravel_db;  
mysql> GRANT ALL PRIVILEGES ON laravel_db.* TO  
'larave_user'@'%' IDENTIFIED BY 'Laravel_Pa55word_here!';  
mysql> FLUSH PRIVILEGES;  
mysql> EXIT;
```

Now if you've setup your MySQL replication correctly you would have to do this only on your Master server and the changes will be replicated to your slave server, here's an example if I check the slave now:



That is pretty much it with the MySQL replication! We now have two MySQL servers: one is a master and one is a slave and all changes from the master are being replicated over to the slave.

Configure Laravel to work with the Master-Slave MySQL setup

Now that we have our Web server ready and our database servers ready as well, we can proceed and actually setup our Laravel application to work with our three servers.

You now need to get back to your web server and configure your Laravel application so that all write queries go to your Master and all read queries go to your slave server.

First in your .env file update your database user and password and comment out the DB_HOST part:

```
DB_CONNECTION=mysql
#DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_db
DB_USERNAME=larave_user
DB_PASSWORD=Larave_l_Pa55word_here!
```

After that in your config/database.php file under the mysql section specify your write and read servers:

```
'mysql' => [
    'read' => [
        'host' => [
            '157.230.25.155',
        ],
    ],
    'write' => [
        'host' => [
            '139.59.155.119',
        ],
    ],
],
```

The file should look something like that after your change:



NOTE: you need to comment out or remove the host part after the url as we've specified that in the write and wirte section.

Now this is pretty much all we have to do! It's time to test our setup!

Install the Voyager admin panel on the new setup

Voyager is super easy to install. After creating your new Laravel application you can include the Voyager package with the following command:

```
composer require tcg/voyager
```

You will also want to update your website URL inside of the APP_URL variable inside the .env file:

```
APP_URL=http://devdojo.bobbyiliev.com
```

Next, add the Voyager service provider to the config/app.php file in the providers array:

```
'providers' => [  
    // Laravel Framework Service Providers...  
    //...  
  
    // Package Service Providers  
    TCG\Voyager\VoyagerServiceProvider::class,  
    // ...  
  
    // Application Service Providers  
    // ...  
],
```

Then finally yo install Voyager with dummy data simply run

```
php artisan voyager:install --with-dummy
```

If you followed the steps correctly you should now be able to browse

through your voyager app and in the background the app would be using the three servers that we've configured!



Conclusion

Now you have your laravel application deployed on 1 Webserver and 2 Database servers. That way you can always easily scale up by adding more slave servers by just adding the new slave IP in your read servers array in your config/database.php file.

If you would like to actually check if the replication is working you can do the following:

1. Connect to the three servers
2. On the master server run:

```
show master status
```

3. Note down the master position
4. On the slave server run:

```
show slave status \G
```

5. Note down the Read_Master_Log_Pos and make sure that it matches the master position!
6. Then on your webserver you could run your database seeders again so that this would make some new in the database:

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

7. Go back to your database servers, check the values again and watch how the numbers change!



Hope that this helps! If you have any questions feel free to reach out to me!

[Originally posted here](#)

Conclusion

Congratulations for going through this Laravel Tips and Tricks eBook! I hope that you've found some of the tips useful and can apply them in your Laravel projects!

If you found this helpful, be sure to star the project on [GitHub](#)!

If you have any suggestions for improvements, make sure to contribute pull requests or open issues.

As a next step, make sure to try and build an awesome project! I would recommend using [Laravel Wave](#) to build your next great SaaS idea!

In case that this eBook inspired you to contribute to some fantastic open-source project, make sure to tweet about it and tag [@bobbyiliev](#) so that we could check it out!

Congrats again on completing this eBook!

Other eBooks

Some other opensource eBooks that you might find helpful are:

- [Introduction to Git and GitHub](#)
- [Introduction to Bash Scripting](#)
- [Introduction to SQL](#)