LARAVEL BLOG GUIDE

How to create a blog site using Laravel 5.3

[What is Laravel](#_uk1arhlrptrv)

[Laracasts: Laravel 5 Fundamentals](#_z2vq84f1tj)

[Lesson 1: Accessing your first Laravel website](#_40cuwhvnrdl6)

[Lesson 2: Routes](#_j3l3scdpmfe9)

[Lesson 3: Views](#_5t3qj2ipncom)

[Organizing your views with directories](#_lyighxu0cagi)

[Passing variables to views](#_n9ihsce2s1es)

[Printing variables in views](#_lcbjfr4jiuo)

[Layouting Using Blade](#_r1h4xabujco3)

[Conditionals and Loops in Blade](#_68iezytmp1fa)

[Lesson 4: Controllers](#_2dntiye36hzx)

[Lesson 5: Environment and Configurations](#_4uutzvgddxrt)

[Lesson 6: Database Migrations](#_oc6gbgsb9djw)

[Lesson 7: Database Query Builder](#_fkrcttiel22t)

[Lesson 8: Eloquent Model](#_8s7sh7l6qsg5)

[Tinker](#_b92zehop3n6u)

[Retrieving objects from database](#_qxze7sry82lk)

[Lesson 9: Route-Controller-Model-View](#_rfnru1nuo6ex)

[Lesson 10: GET Request](#_lo4p0oh6i46e)

[Lesson 11: POST Request and Form Validation](#_wgzl124rk5cw)

[Creating a form](#_hoxpc9fy5ai1)

[Defining submit behavior](#_1obucwui97ja)

[Getting the values from the form.](#_vqizppab7ywe)

[Deleting an article](#_jw2lgq1vmlw6)

[Lesson 12: Redirects](#_miqzo4mfu8x3)

[Lesson 13: Sessions](#_g3kjuomztl09)

[Lesson 14: Validation](#_vr5tmyk4k61f)

[Creating Validation Rules](#_b0uw4m9ieece)

[Displaying Error Messages](#_ii9bfbouxzu2)

[Lesson 15: User Authentication](#_456322mr4mxl)

[The make:auth command](#_aon5pwygdz07)

[Viewing the template for user signup](#_kpic6hjhyc9d)

[Accessing the currently logged in user](#_mu5mgfsn7huh)

[Protecting the access to controllers](#_xdfcee632fa9)

[Lesson 15 B: Customizing the User fields](#_5xjxgjgkw2op)

[Adding a new attribute to User model and changing the primary login credential.](#_e25ozteer13i)

[Step 1: Adding the username column in the database](#_jc1gctlo6pmf)

[Step 2: Updating our User.php model](#_fby0nz4xp93h)

[Step 3: Updating our RegisterController.php](#_dsivmn8mzk6j)

[Step 4: Updating our register.blade.php to add the username field in our form](#_p7dln2mp9ctp)

[Step 5: Updating our login.blade.php to change the email to username field in our form](#_v7htpms7cew)

[Step 6: Updating our LoginController.php](#_nxr4ku4sdoaj)

[Lesson 16: Eloquent Relationships](#_azw3qq7nzg6j)

[Part 1: Many-to-One Relationship](#_txrqi5rpadko)

[Creating a Comment Model](#_iokivwqew6uo)

[Using Relationship with Comment Model](#_1k9akf2cux9e)

[Part 2: One-to-Many Relationship](#_7ppfhjl551ja)

[Part 3: One-to-One Relationship](#_xp0nnkgqgg6)

[Part 4: Many-Many Relationship](#_ah7i7nu25h7o)

# What is Laravel

Components of Laravel

* Routes
* View
* Controller
* Model

Installation of Laravel

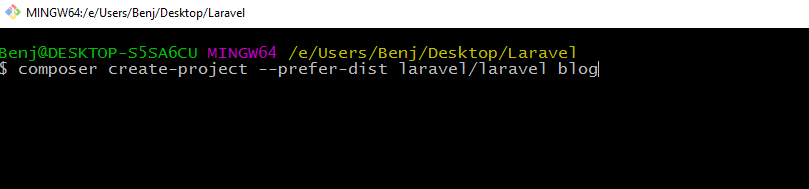
Requirements:

* Composer: getcomposer.org
* PHP 5.6.4 onwards (for Laravel 5.3 onwards)

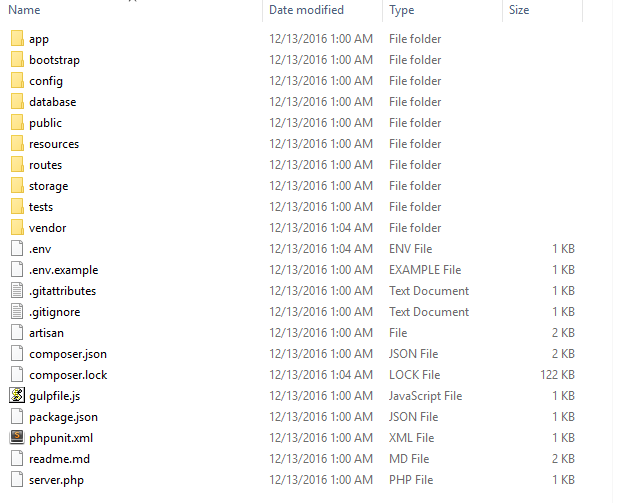
1. After installation of composer, create a directory where you want to create your project
2. Using command line or git bash, type

composer create-project --prefer-dist laravel/laravel <project\_name>

The installation may take a while...



1. Upon installation, there will be a initial files for our project.



# Laracasts: Laravel 5 Fundamentals

If you need video tutorials

<https://laracasts.com/series/laravel-5-fundamentals>

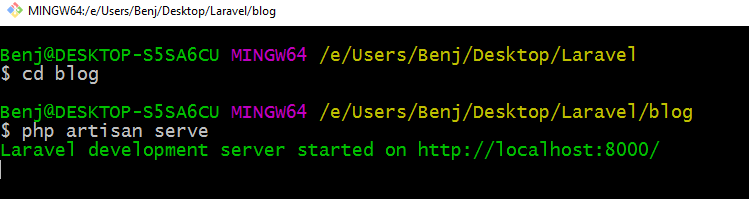
# 

# Lesson 1: Accessing your first Laravel website

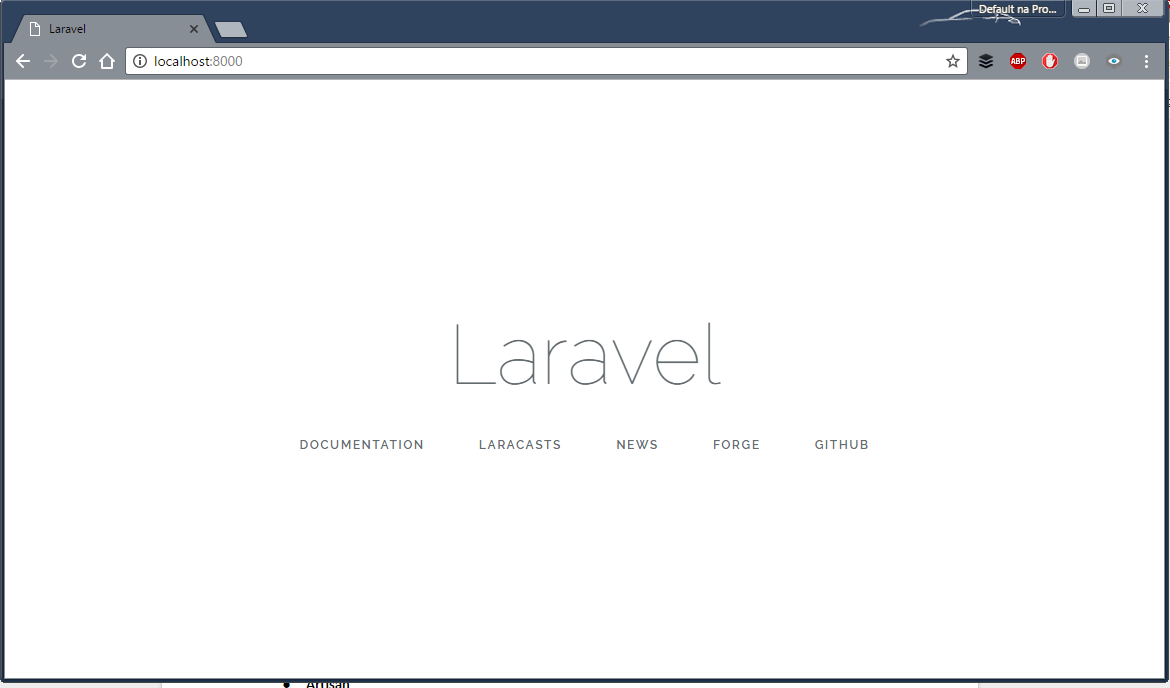
1. Using command line, go to your project directory and type:

***php artisan serve***

This will start your PHP server instead of the Apache in XAMPP. Keep this window open for the website to run



1. To view your site, in your browser, go to localhost:8000



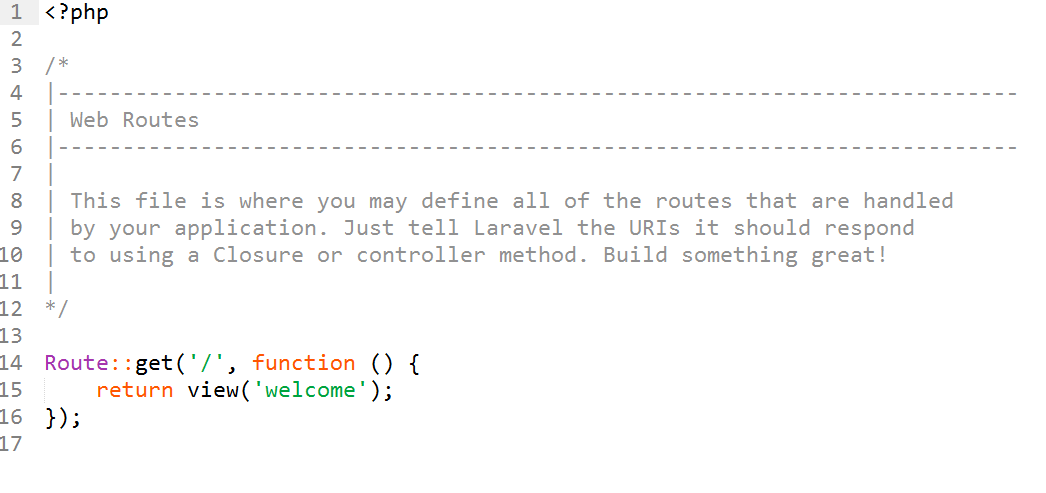
# 

# 

# Lesson 2: Routes

The routes file determine what to do upon receiving a particular URL.

For version 5.3, the routes file is located in **routes/web.php**



This file will contain entries that will identify what Laravel will do based on the url.

In the example above, when we go to localhost:8000/ , the router will pass the responsibility to the view to render the welcome page

To demonstrate, we’ll add a new entry to the routes file.



In lines 19-21, when we go to our site with the additional ***/hello*** (localhost:8000/hello) in the URL, instead of returning a welcome page, we return a string “Hello World”

In the future, we can define more routes for our site to be able to handle more type of requests

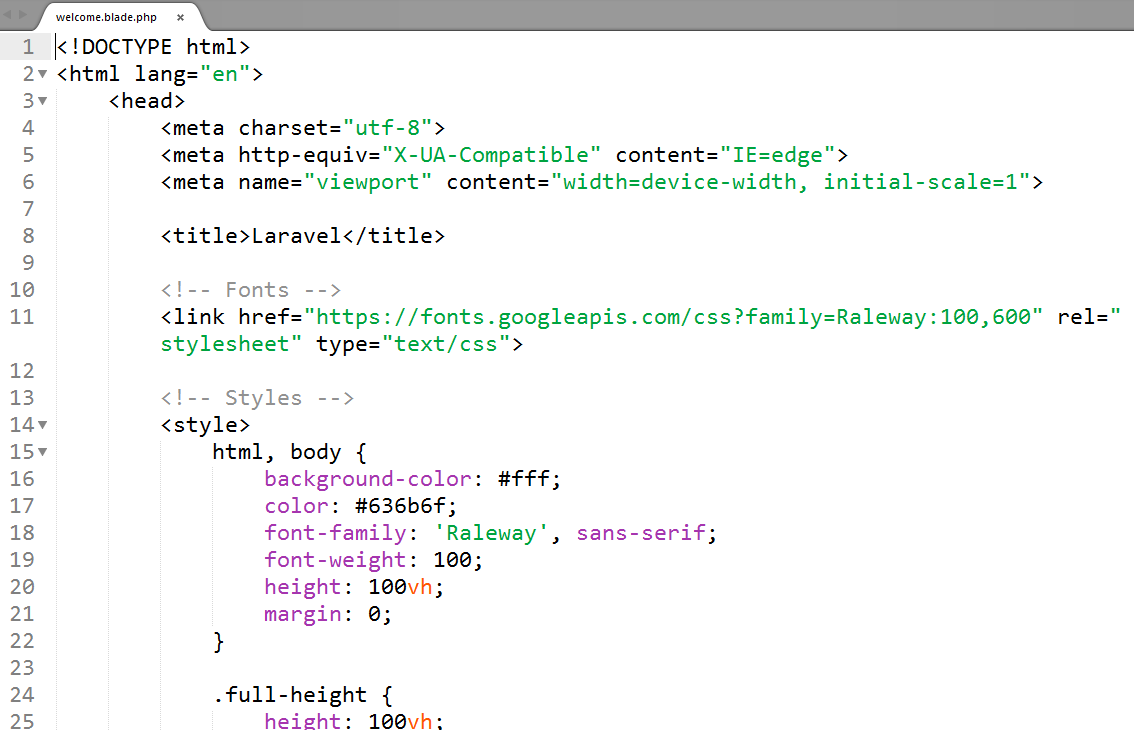
# 

# Lesson 3: Views

Views are the one who are responsible displaying most of the HTML/CSS contents to the user. The purpose of Views is to display our PHP variables along with the HTML contents. As much as possible, we want to minimize programming in views.

The View files are located under **resources/views**

In the previous lesson, the statement **return view(‘welcome’)** finds the view file, ***welcome.blade.php*** and displays it to the user

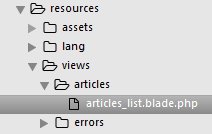


*welcome.blade.html*

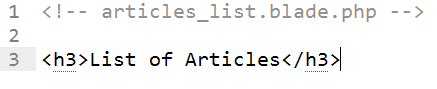
The views will have the **.blade.php** extension

## Organizing your views with directories

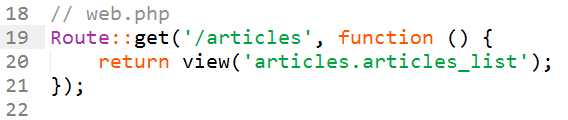
* To use a view within a directory, we add the directory name followed by a dot (.) then the filename



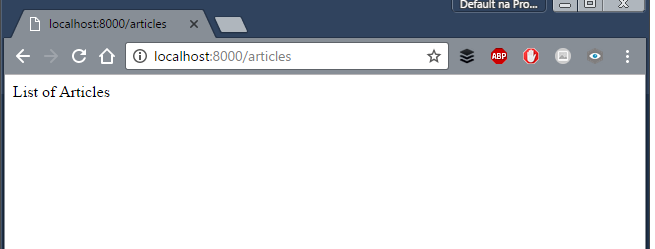
Directory structure



Contents of article\_list.blade.php

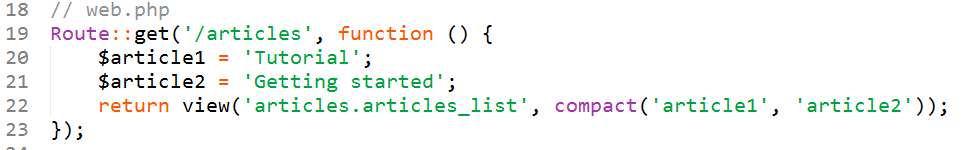


Contents of web.php (routes)



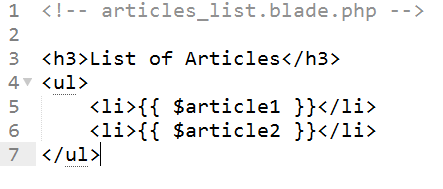
## Passing variables to views

To pass a variable to the view, we can add a 2nd parameter to the view() function as an array of variables. We can also use the compact() function for shorter syntax



## Printing variables in views

* <?php echo $var ?>
* {{ $var }}
* {!! $var !!}





## Layouting Using Blade

**Creating a Layout File**

Create your layout file as usual with the HTML and CSS elements and save it with the .blade.php extension.

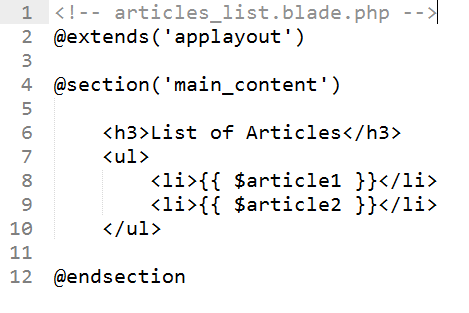
Add **@yield(<section name>)** where you want your section to be placed in your layout

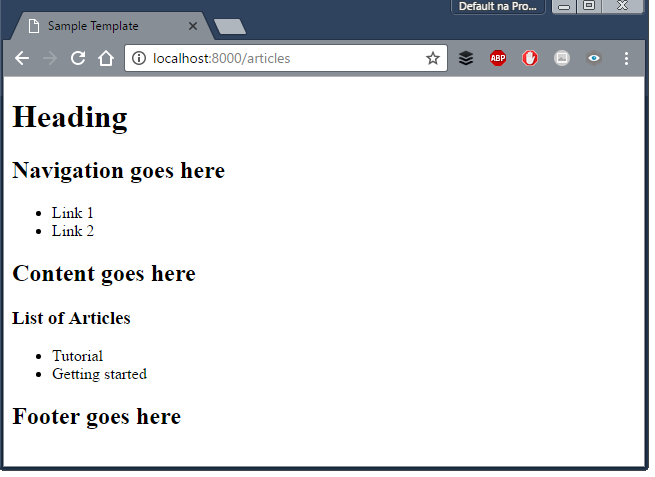


**Using your layout**

In your View file, use **@extends()** to specify the layout you want to use

Use **@section(<section\_name>)** and **@endsection** to specify the contents of your section to be placed in your layout.





## Conditionals and Loops in Blade

Blade includes if statements and for loops that can be used without placing them in <?php ?> tags

* If statement:

@if(condition)

…

@endif

* If-else statement:

@if(condition)

…

@else

...

@endif

* Foreach statement

@foreach($var in $list)

…

@endforeach

# 

# 

# Lesson 4: Controllers

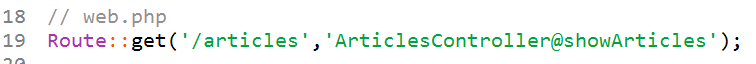
To create a controller, type php artisan make:controller <ControllerName>



It will create the controller in the **app/http/controllers** directory

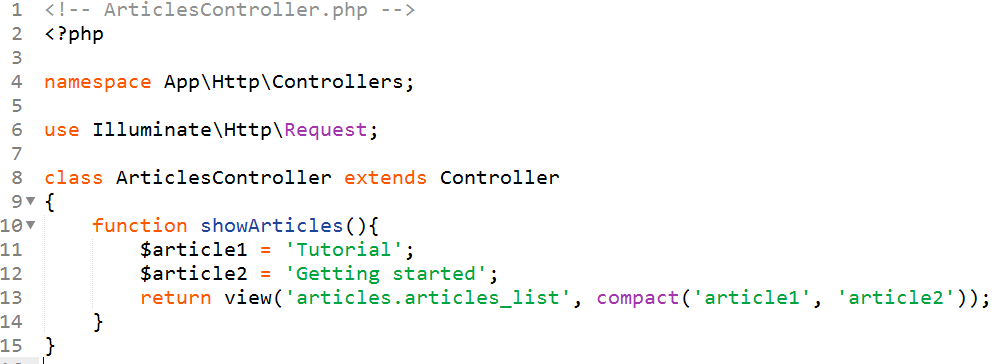
Most of the time, the routes file can only handle minimal programming logic. The purpose of the controller is to determine the actions of the website. Here is the **common workflow**

1. The routes file determines which controller to call



*The showArticles() method in ArticlesController.php will handle the request*

1. The controller handles the PHP programming part
   * The controller creates variables for the result
   * The controller passes the variables to a View for output (line 13 below)



# 

# 

# Lesson 5: Environment and Configurations

The environment and configuration files are:

* **config/database.php**



* **.env file**



We can set the database connection settings here. By default, values in the .env file are used. If the **.env** file is missing, the values in the **database.php** are used.

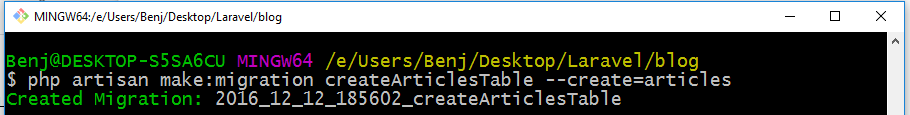
# 

# Lesson 6: Database Migrations

**Step 1: Making a migration file**

Migrations are like version control for your databases. They are like PHP scripts to create tables in your database. To create a migration file, type ***php artisan make:migration <migration\_file>***

* Use **--create=<tablename>** if we want to create a table
* Make your migration file to be descriptive
* Migration files are stored in **database/migrations**



*Migration file to create the articles table*



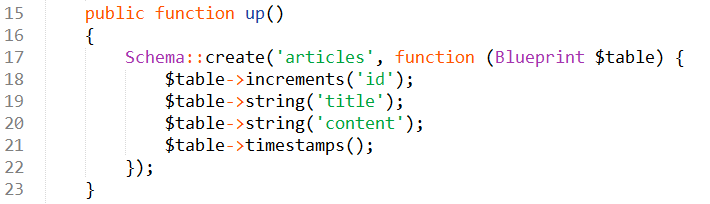
*Migration file to create an articles table (the timestamp may vary)*

**Step 2: Modifying the migration file**

In the migration file, we have two functions

* function up()
  + Describes what to do when we **run** the migration
* function down()
  + Describes what to do when we **undo** the migration

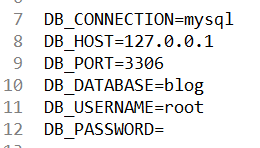
Let’s say we want to create a table articles with columns: **id, title, content** and a **timestamp**. We modify our migration file:



For more details about data types: <https://laravel.com/docs/5.3/migrations>

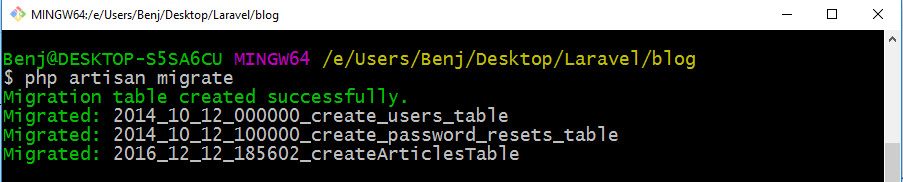
**Step 3: Create a database**

You can use PhpMyAdmin to create your database. Make sure MySQL is running and your **.env file** has the correct values.

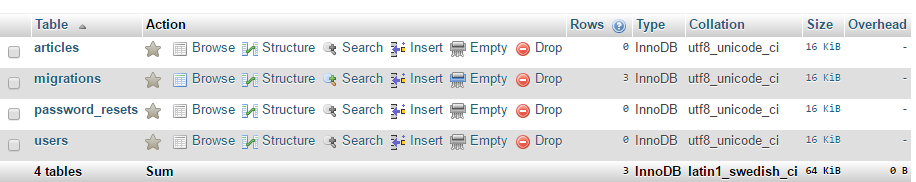


**Step 4: Run the migration**

Type ***php artisan migrate***



This will create tables in our database

****

Tables created

* **Articles** - comes from our created migration file
* **Users** and **password\_resets table** - provided by Laravel by default
* **Migrations** - list of migrations that are already done. Laravel keeps track of migrations that are already done and which ones are not.

To undo a previous migration

* Type **php artisan migrate:rollback**

**Benefits:** Our database tables are automatically created from our migration files.

# Lesson 7: Database Query Builder

Laravel allows us to access our database using its query builder. It has functions for SELECT, UPDATE, INSERT, and DELETE. We can also add additional clauses like WHERE, HAVING, etc.

<https://laravel.com/docs/5.3/queries>

# 

# 

# Lesson 8: Eloquent Model

<https://laravel.com/docs/5.3/eloquent>

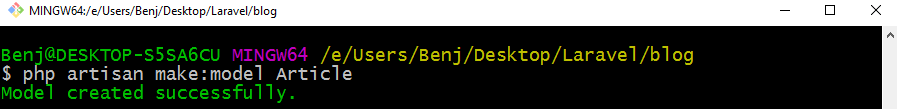
This concept will be fairly new to you… This is where we will apply our OOP lessons previously.

Think of our website (blogsite) that it will contain objects like articles, comments and users. We want our objects to be created, retrieved, modified, and deleted in our database with ease. Here is where our model will be useful.

To create a model, type

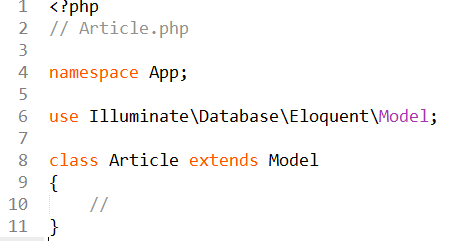
***php artisan make:model <model name>****.*

Models are stored in the **app** **directory**. As a convention, start with an uppercase and use singular form.



If we want to create a migration file along with the model, we add the --migration option

***php artisan make:model <model name>******--migration***

******

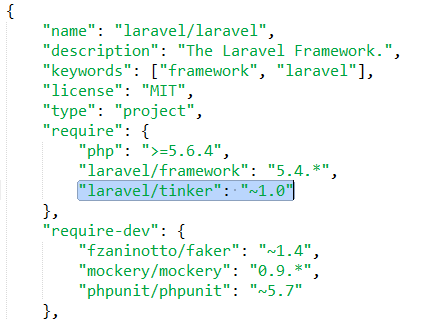
*app/Article.php*

## Tinker (See changes in version 5.4)

## Tinker

Tinker is a tool to run PHP scripts without having to type them in a PHP file. Tinker is a PHP console where we can test and execute our models. What we type here can be also used in our controllers.

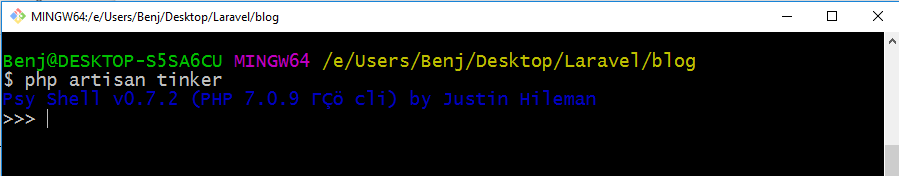
Make sure you have included the tinker in your composer.json file



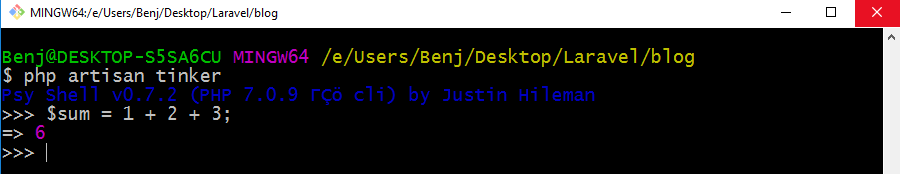
Then type *composer update* in your command line to include tinker

To run tinker

***php artisan tinker***

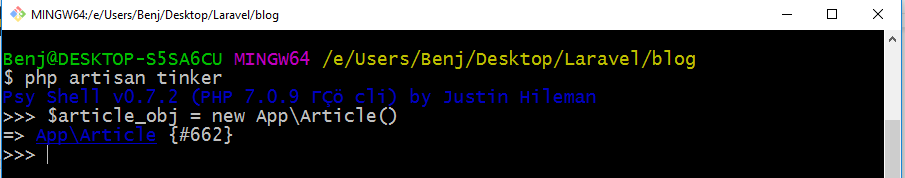


We can run PHP lines

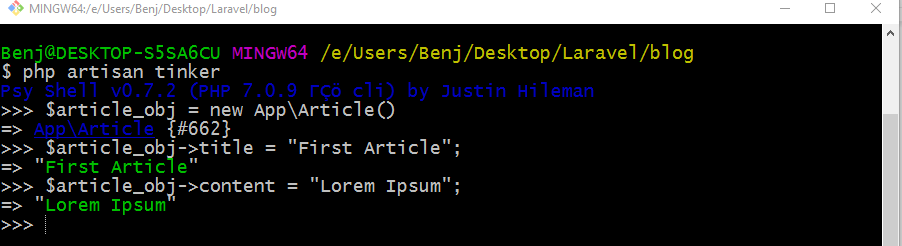


One characteristic of tinker is it immediately prints the result of our last expression

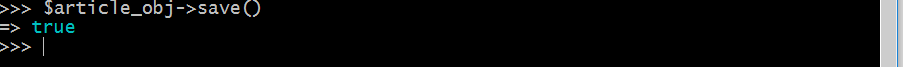
We can now create our article object using **new** keyword



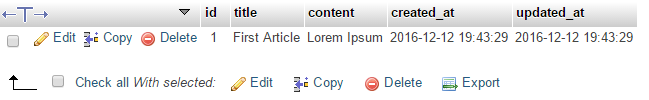
Set the *title* and *content* properties



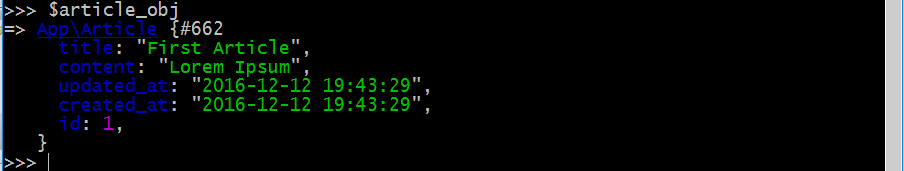
And we can save our object using **save()**



And our object is saved automatically in the database. No need for INSERT statements



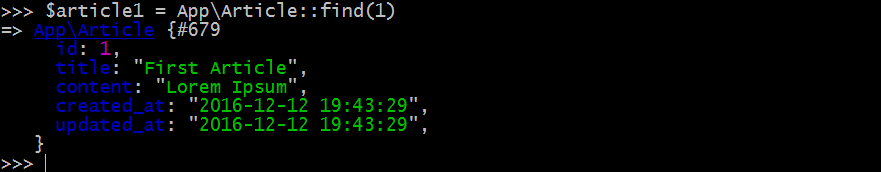
To view our article object, we can type the variable name

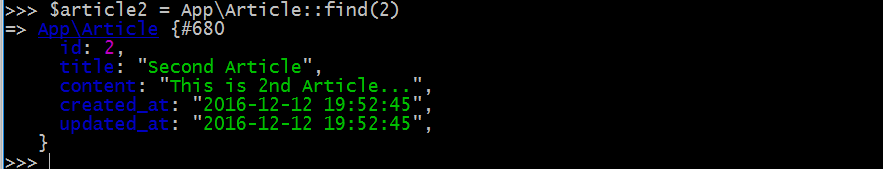


EXERCISE: Add more articles in the database using Tinker

## Retrieving objects from database

To retrieve an object from the database, we can use its primary key column id and the **static method find()**





We can now retrieve the attributes such as $article1->title

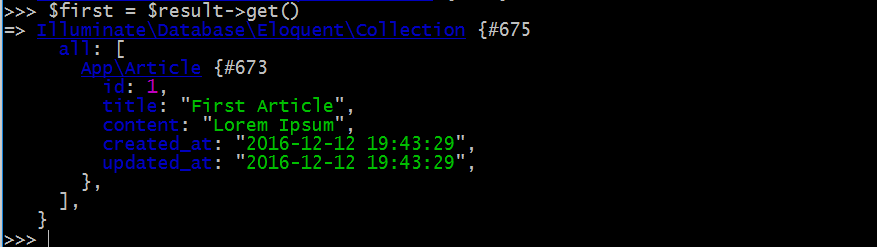


To retrieve using another column like title, we can use the **where()** method

****

**This will give us a LIST of objects.**

To get a single result, we can use the **->get()**



To retrieve all Articles: we use the method **all()**



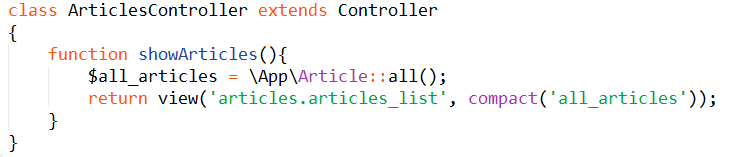
This will give us a **LIST of Article objects.**

# 

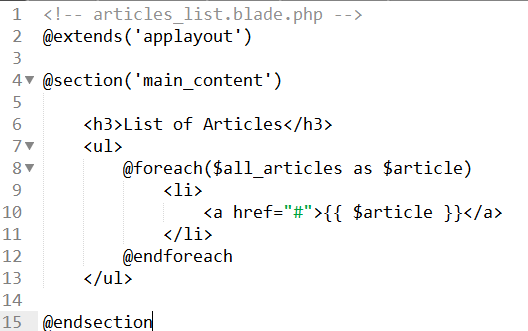
# 

# Lesson 9: Route-Controller-Model-View

After setting up each component (route, controller, model and view), it is now time to integrate all those components.



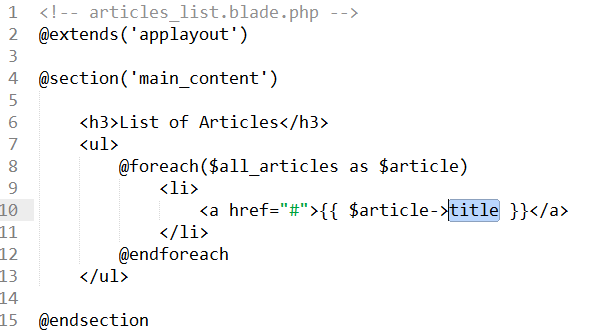
We update our ArticlesController to retrieve all articles from the database

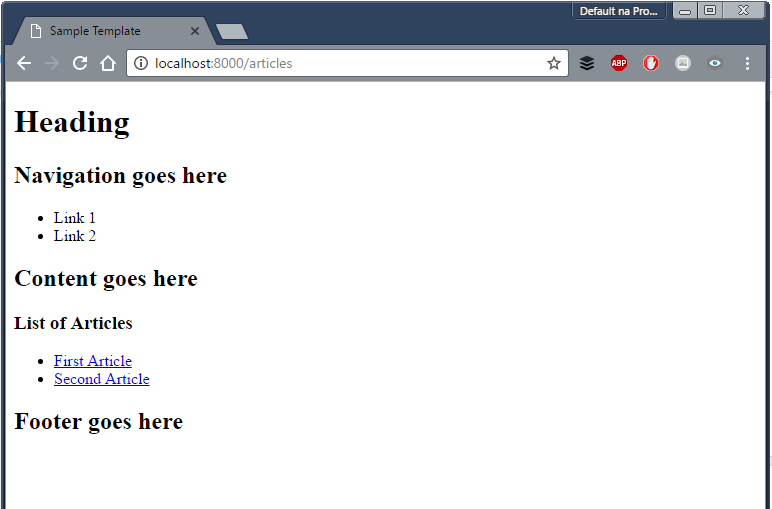


Then we use @foreach to loop in our collection of Article objects. Then we check our browser



Notice that it prints the whole Article object instead of the title only. We update our articles\_list.blade.php to show the title only.





# 

# 

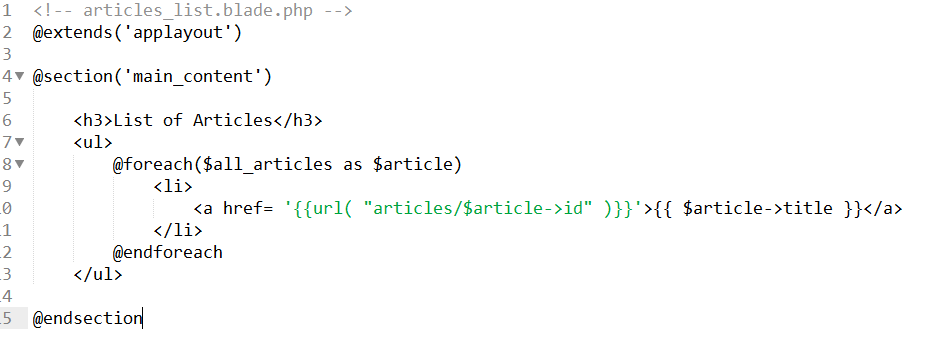
# Lesson 10: GET Request

Making our link to go to the specific page

Let’s say we want when we click the first link, we want to go to the

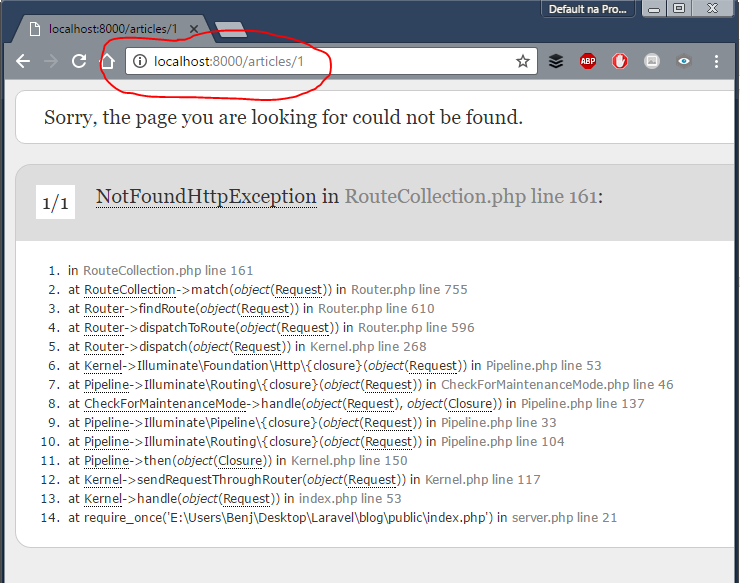
* First Article -> localhost:8000/articles/1
* Second Article -> localhost:8000/articles/2

We’ll update our href tags:

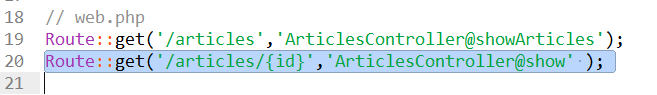


We add the {{ url() }} function to generate the url path of the href tag for us. The parameter of the url(), **“articles/$article->id”** means that we wanted to go to “localhost:8000/**articles/1**” for the first article and “localhost:8000/**articles/2**” for the second article.

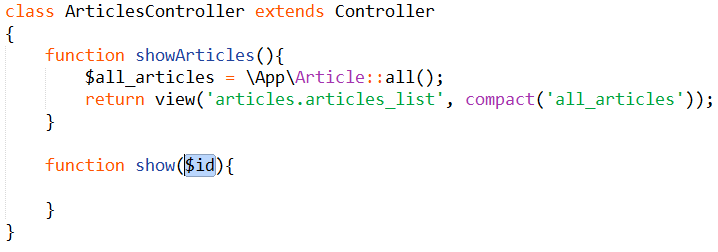
Upon clicking the link, we would actually go to the address we expected



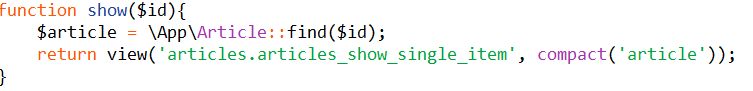
We receive a **NotFoundHttpExecption** error because our route is not yet defined. We update our routes and add an additional line

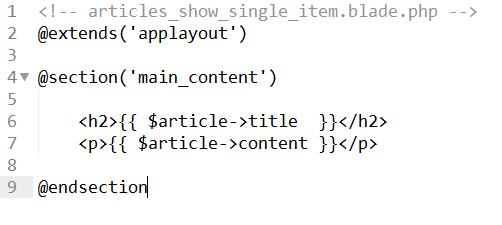


We define a new route where we define a “wildcard” character ***id*** after the articles/ path. We place our ***id*** inside a curly brace to denote that it will be a parameter in our **show()** method in our ArticlesController

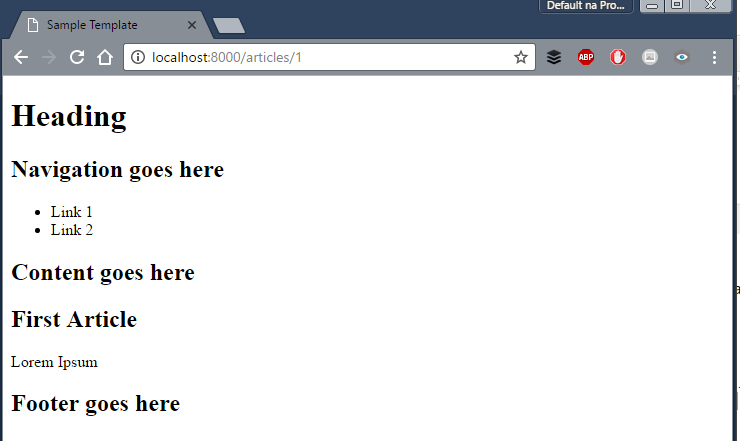


We retrieve the article object with the given id and pass it to our new view: **articles\_show\_single\_item.blade.php**.



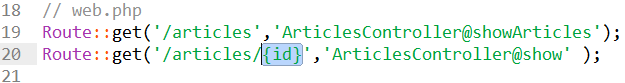


Our page now works

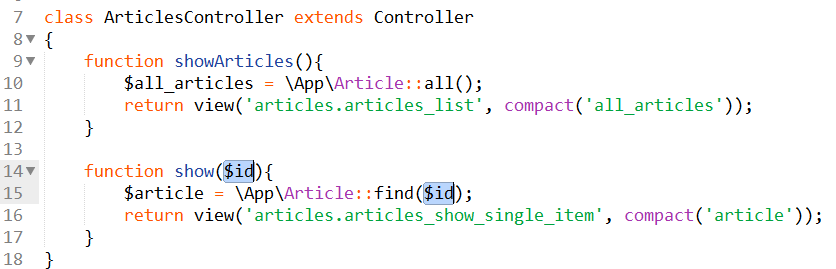


**Summary**: We can submit our GET variables in the path:

1. Adding a ***wildcard*** in our route using the curly brace



1. Adding a parameter in our controller function



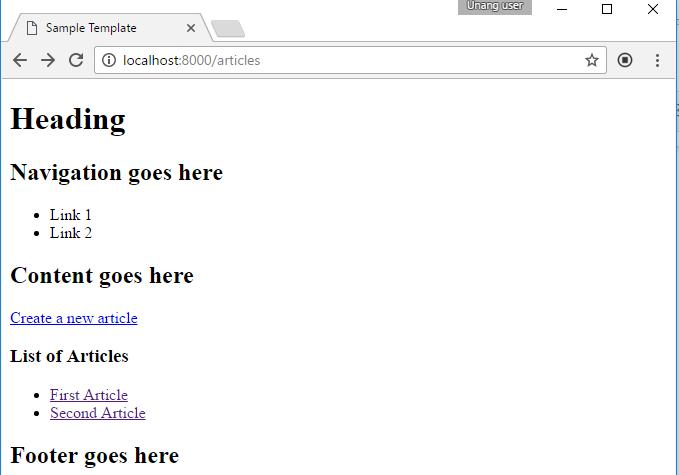
# 

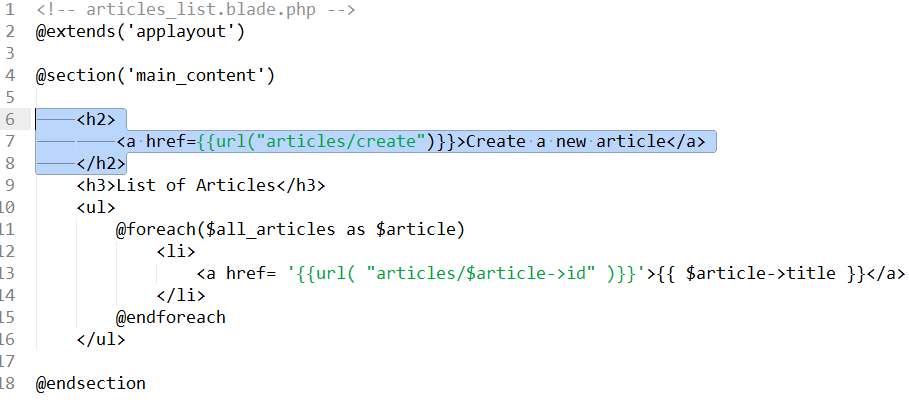
# 

# Lesson 11: POST Request and Form Validation

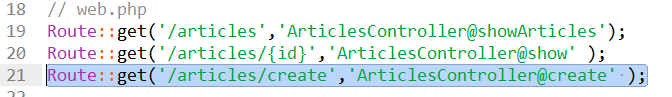
### Creating a form

We modify our **articles\_list.blade.php** to include a new link to create a new form

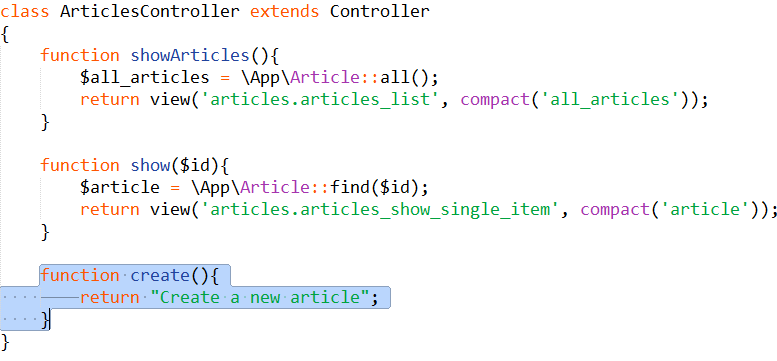




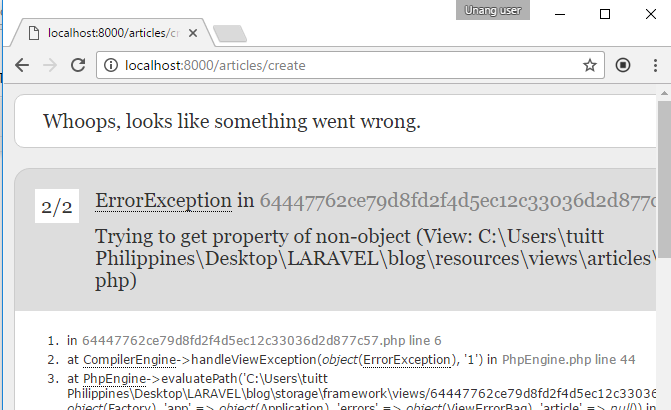
We also update the web.php route file



And add a function create in ArticlesController:



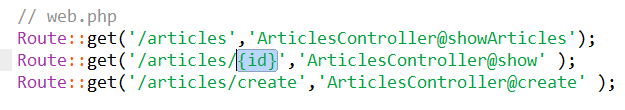
We test our output:



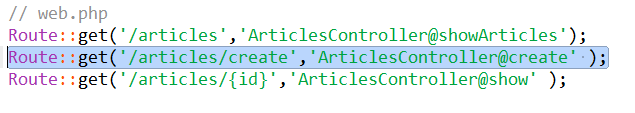
We get **property of non-object** error..

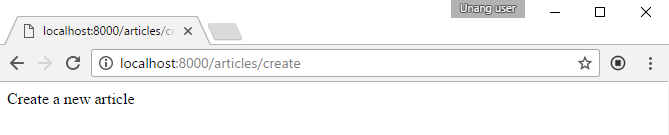
Explanation:

What is really happening is when we go to localhost:8000**/articles/create**, the “create” keyword is treated as the **{id}** in the previous route.



To solve this, **we change the order** of the two routes:

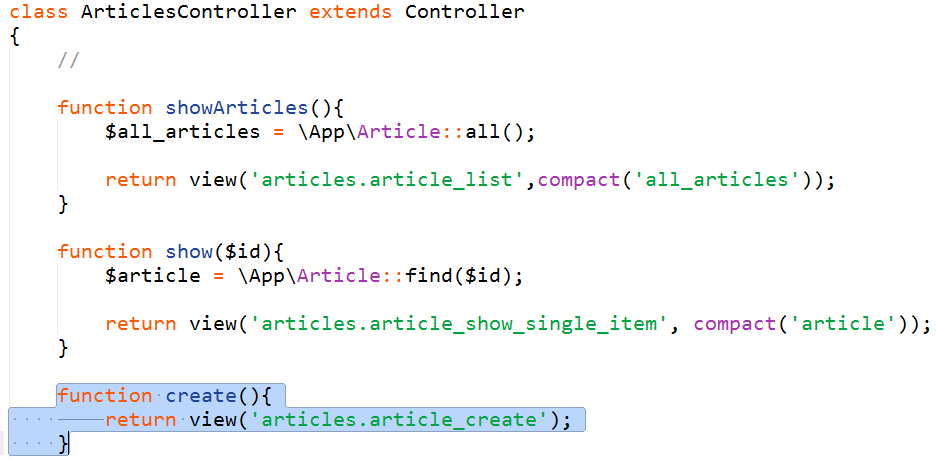




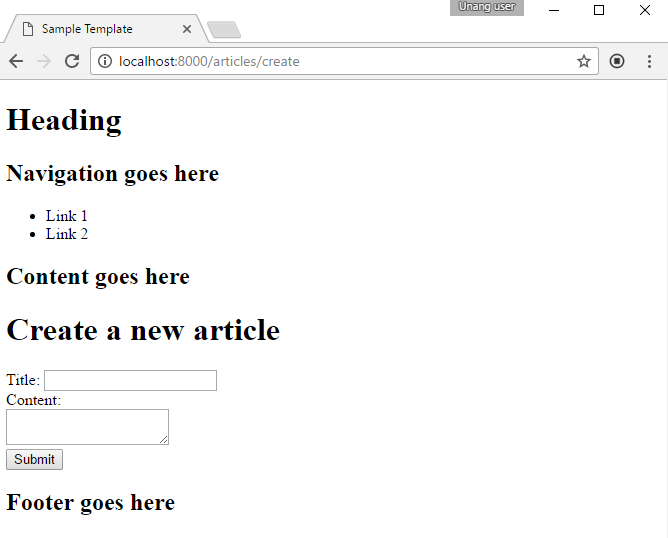
Next is we create a View for our form. We save it as **articles\_create.blade.php** in the articles directory.



We update our ArticlesController to return our view:



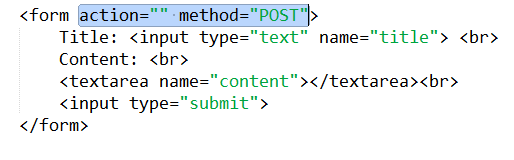
View the page: localhost:8000/create



CONGRATULATIONS ON REACHING THIS PART! :)

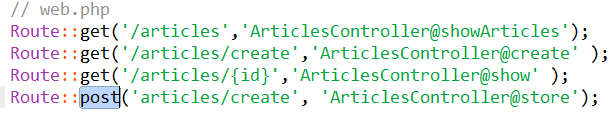
### Defining submit behavior

Our next step is to specify our action and method properties in our form.

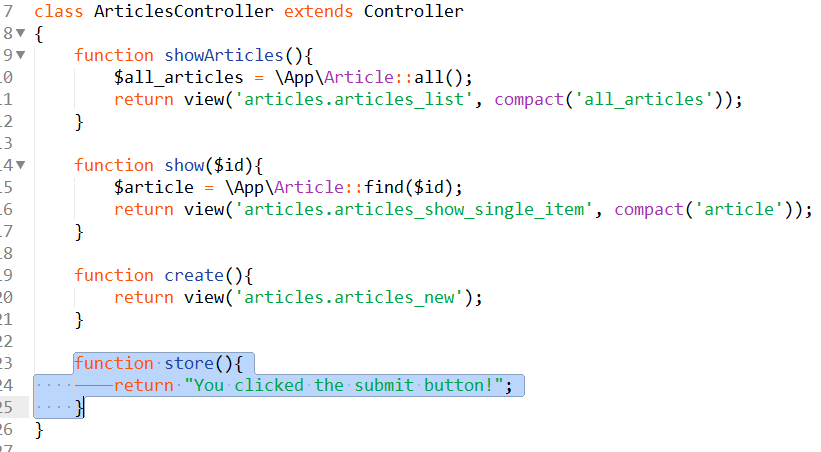


We set the **action** method of our form to be empty (means it will post in the same url: localhost:8000/articles/new) and our **method** to be POST.

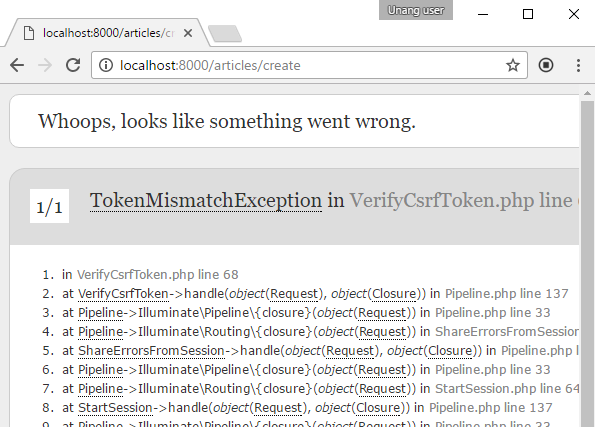
We update our **routes** to handle the POST request to localhost:8000/articles/new and create a function ***store()*** in ArticlesController



Notice that instead of using Route::get(), we used **Route::post()** to describe that we want the ArticlesController to use the store() method when we have a POST request.



Then we try to click the submit button in our form:

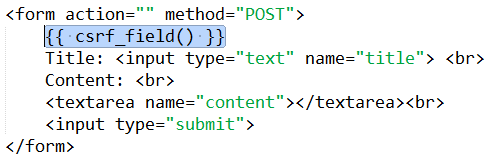


We get a TokenMismatchException error.

Laravel needs a CSRF token to handle form data. Check <https://laravel.com/docs/5.3/csrf>

For more details.

We add a **{{ csrf\_field() }}** in our form. This will create a hidden input with name “\_token” in our form.



And we try to submit again:

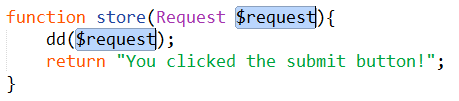


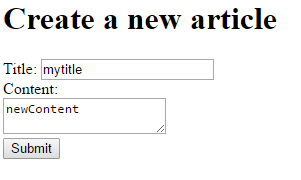
Our form is now accepted and our controller gives us the proper response.

**IMPORTANT** : We will receive a form when we typed **localhost:8000/articles/create** in the address bar since it is a GET request and does not come from a form.

### Getting the values from the form.

We add a **Request objec**t parameter in our store() function in **ArticlesController** and dd() (dd-stands for ‘dump and die’) to print the request object:

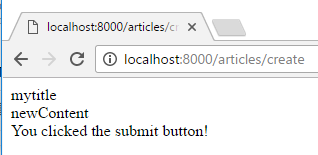
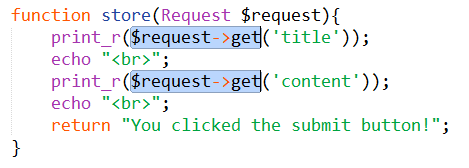




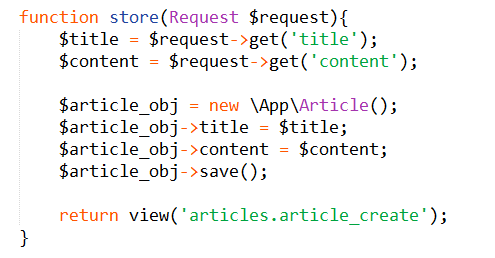
And we can see the values we passed from the **$request** object.

Furthermore, we can use the **$request->get(‘’)** method to extract the values we passed from the form

creat

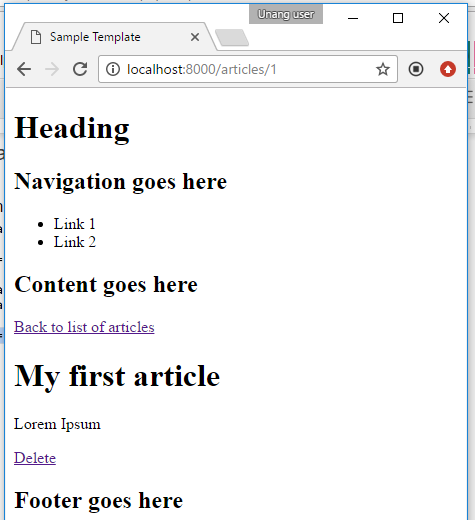


We update the store function to save the article. Then we display again the form after saving our article

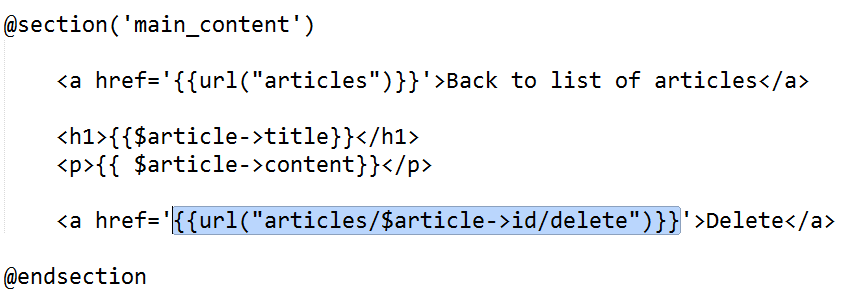


### Deleting an article

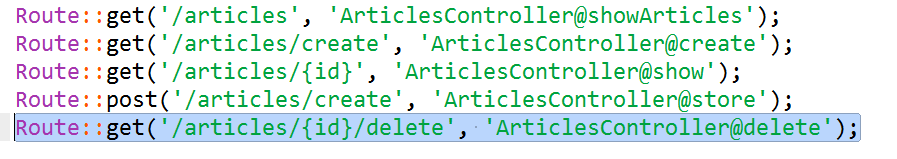
We add a link to delete the article in **articles\_show\_single\_item.blade.php**



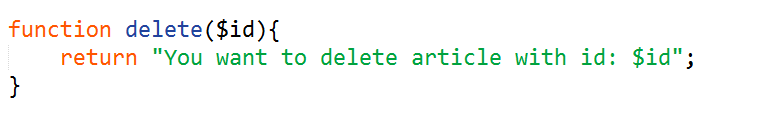
And set the link go to the **localhost:8000/articles/{id}/delete**



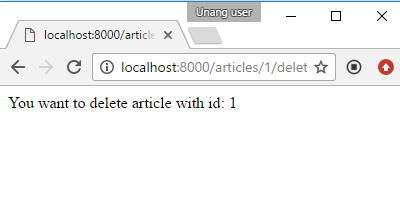
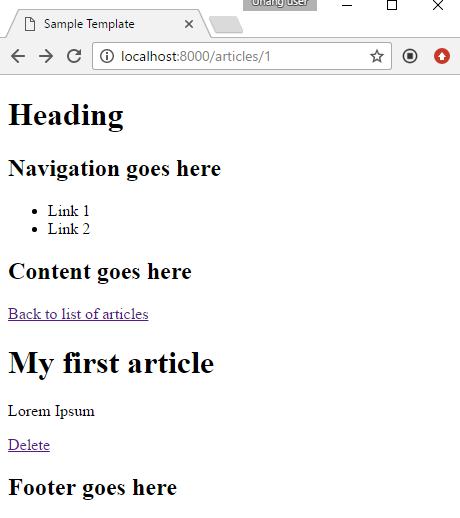
We create a new **route** for the delete and a function **delete()** in our controller:



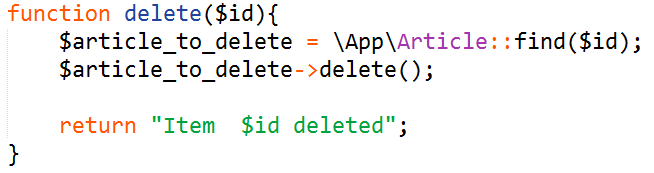
**ArticlesController.php:**

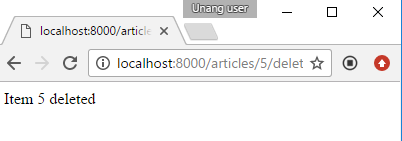


Clicking delete will give us this result:



Once we are already receive the id of the article that we wanted to delete, we update our **delete()** function in **ArticlesController**.



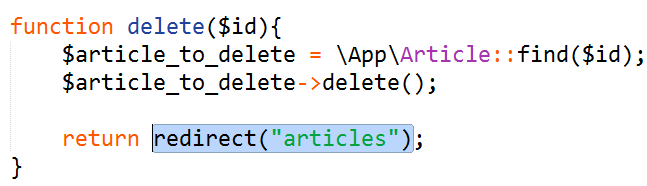


# 

# 

# Lesson 12: Redirects

Suppose we want to redirect to a new url after some code in our controller, we can add a redirect in our delete function in the previous topic



This will redirect us to localhost:8000/articles.

For more information about redirects: <https://laravel.com/docs/5.3/redirects>

# 

# 

# Lesson 13: Sessions

<https://laravel.com/docs/5.3/session>

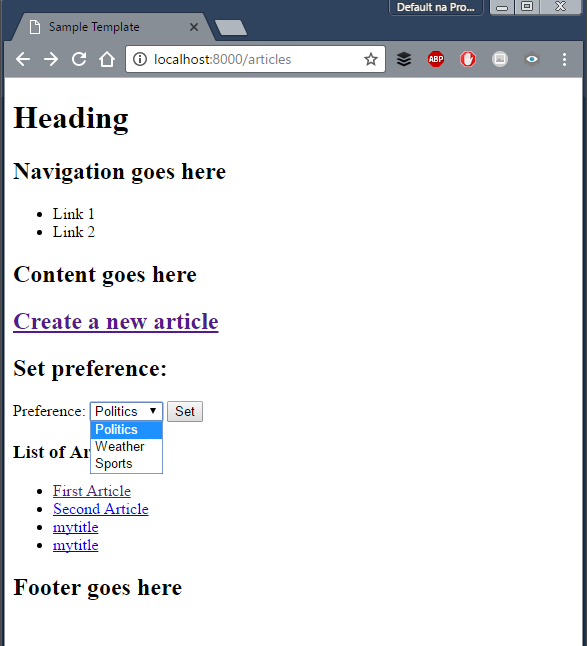
Session variables are variables that can be accessed globally by any webpage within the website. In plain PHP, we use the $\_SESSION superglobal variable to access the variable. In laravel, accessing the session variables and keys are similar.

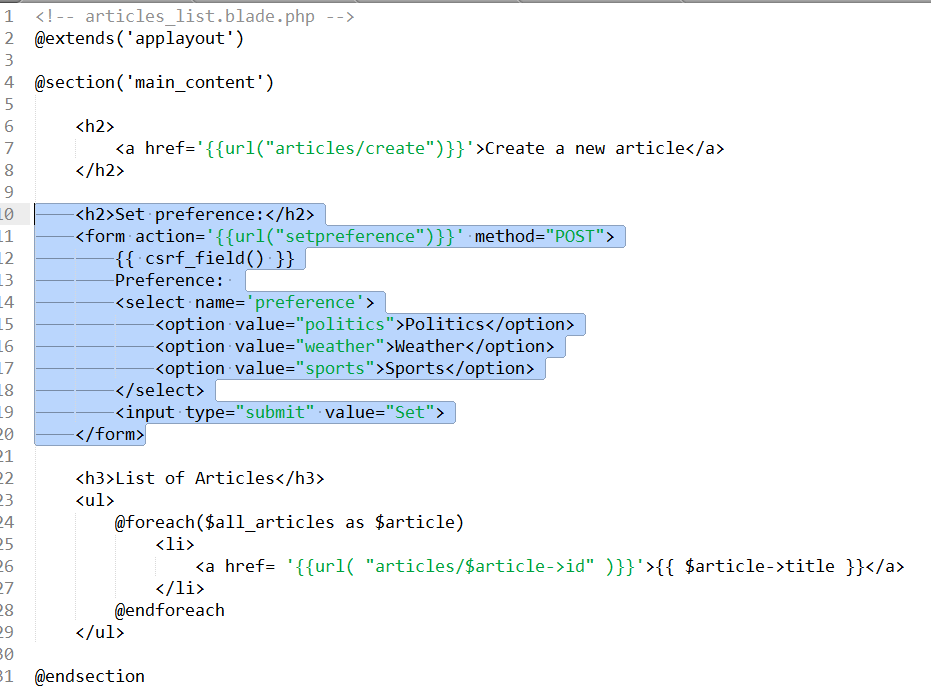
|  |  |  |
| --- | --- | --- |
| Action | Plain PHP | Laravel |
| Storing data | $\_SESSION[‘key’] = $value | $request->session()->put(‘key’, $value) |
| Retrieving data | $var = $\_SESSION[‘key’] | $var = $request->session()->get(‘key’, ‘default’) |
| Deleting data | unset($\_SESSION[‘key’]) | $request->session()->forget(‘key’) |
| Deleting all data | session\_destroy() | $request->session()->flush() |
| Checking if data exists | isset($\_SESSION[‘key’]) | $request->session()->has(‘key’) |

\*- the ‘default’ in session()->get() specifies the default value returned when the data does not exist

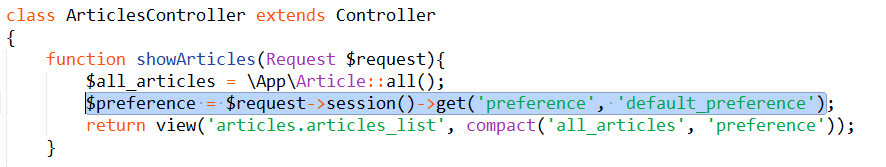
Coding Example: Suppose we want to set a “preference” value in our session that will be available anywhere in our website.

Let us update our **article\_list.blade.php** to include a form to set our preference.

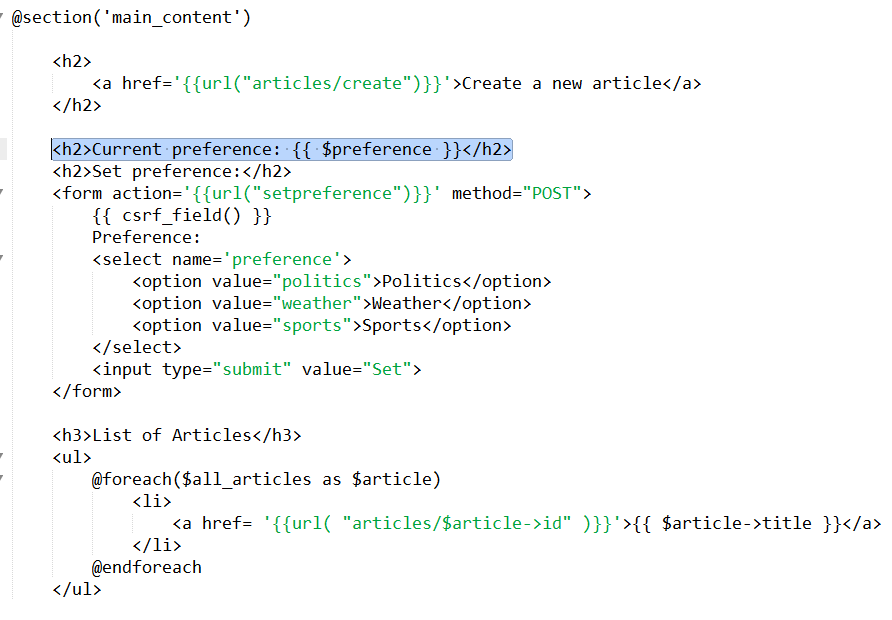


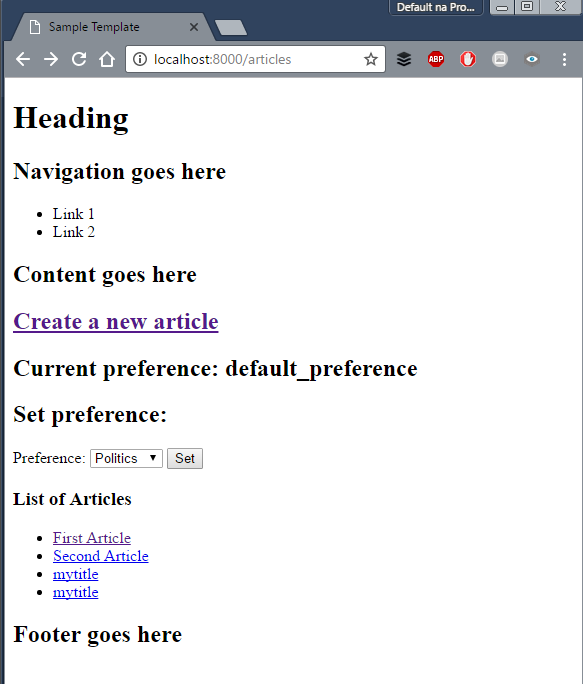


Then, we want to show the preference in the articels\_list.blade.php. We edit our showArticles() in **ArticlesController.php** to load the preference from the session.

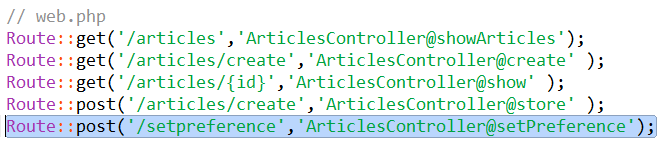


And we print it in **articles\_list.blade.php**

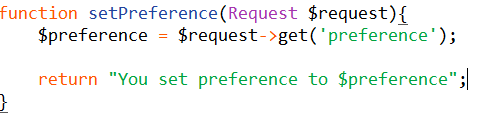




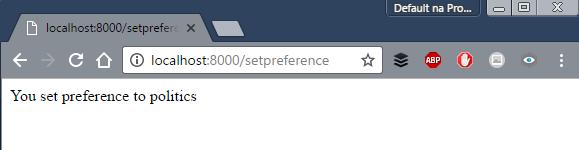
Our form will send a POST request to localhost:8000/**setpreference** therefore we need to setup a route there.



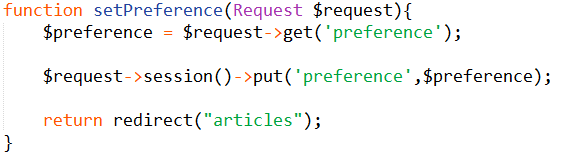
And we update our **ArticlesController.php** to add the setPreference() function



And we try to set a new preference:



Now we are ready to set a values to our session. Retrieve the preference using $request object and set the session key ‘preference’ in **ArticlesController.php**



# 

# 

# Lesson 14: Validation

On the previous lesson, our controller does not validate the contents of the form when we passed the values. Suppose we wanted the following constraints on our title and content on articles

For the title field

* Cannot be empty
* Minimum of 3 characters and maximum of 10
* Must be alpha-numeric (numbers and letters only) No special chars like !,@,# etc

For the content

* Cannot be empty

## Creating Validation Rules

We add the following **rules** in our store() in **ArticlesController.php.** We specify our rules in an associative array with the keys as the fields and values as the rules.



.

For more validation rules:

<https://laravel.com/docs/5.3/validation#available-validation-rules>

Then we add the validation of the POST request



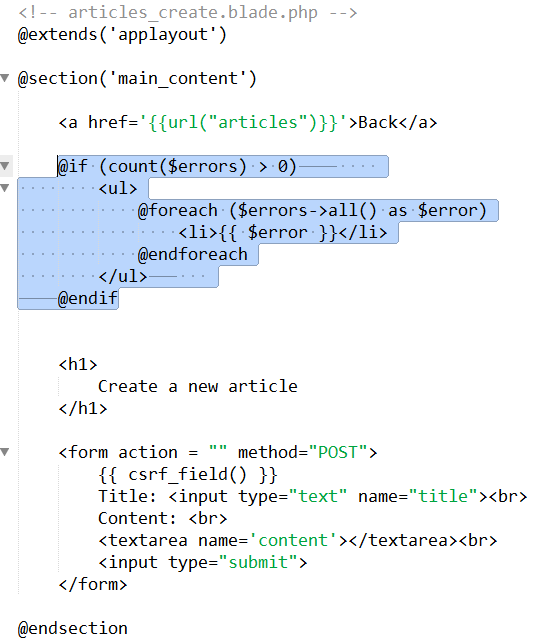
If the validation is **passed**, the execution continues (lines after line #32 will be executed)

If the validation **fails**, the execution stops at line #32 and goes back to the previous page.

## Displaying Error Messages

When the validation fails, the error messages is sent to the SESSION using **flash** data using the **$errors** variable. Unlike regular session variables, flash data are **immediately removed once retrieved**.

We add the following code in **articles\_create.blade.php**



And the validation error appears.



Since the error messages are stored as **flash data** in our session, the error messages disappear when we refresh the page.

# 

# 

# Lesson 15: User Authentication

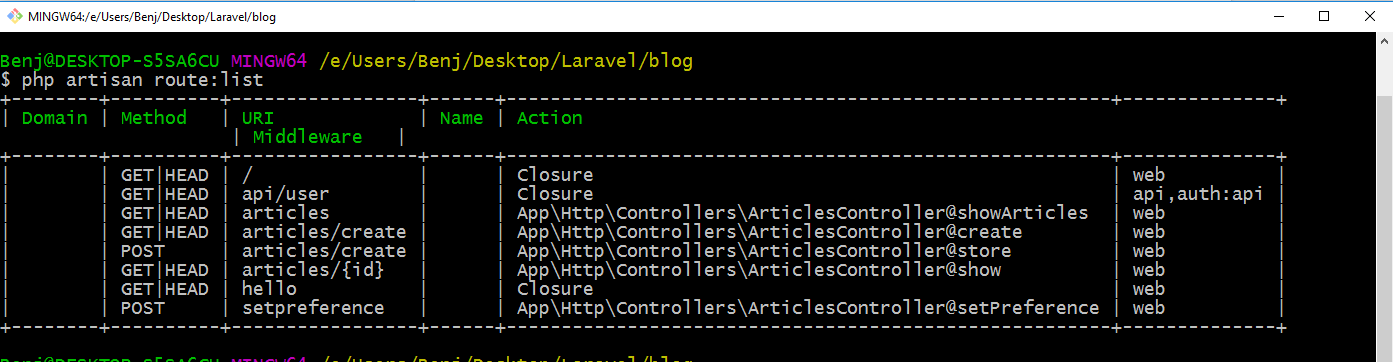
<https://laravel.com/docs/5.3/authentication>

Now we want to create users for our blog. By default, Laravel provides files (controllers, models, migration) for the user login. Laravel will can create a ‘template’ that will allow us to:

* Register new users
* Login user

Before we start, we check our current routes by typing

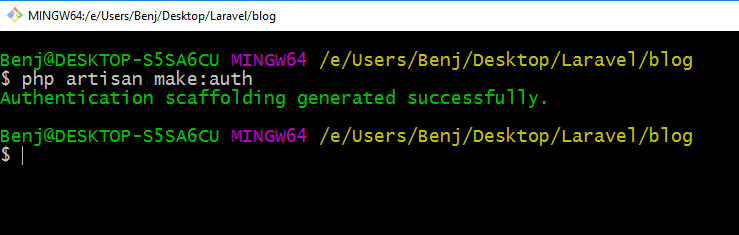
*php artisan route:list*



We’ll see here the current routes that are available we created from the **web.php** routes file.

## The *make:auth* command

To create other files related to user login and authentication, we’ll type **php artisan make:auth**



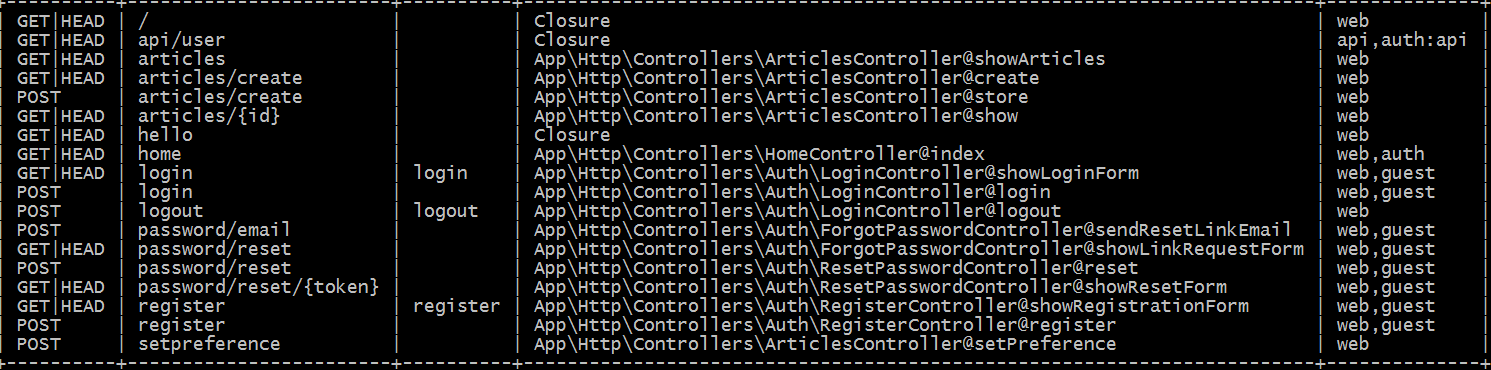
Now this command will create the authentication scaffolding in our files. We’ll discuss the changes made here.

1. For our **Route** changes

Laravel will update our **web.php** and add the following line:



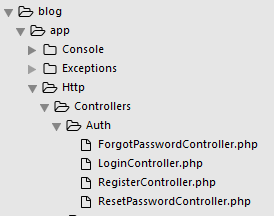
So what do they do? We check the our routes:list to view the new routes.   
Type **php artisan route:list again**



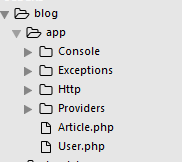
We’ll focus on the some new routes added

* localhost:8000/login
  + This will direct us to the showLoginForm() method of LoginController
* localhost:8000/register
  + This will direct us to the showRegistratonForm() method of Registerontroller
* localhost:8000/home
  + This will direct us to the index() method of HomeController

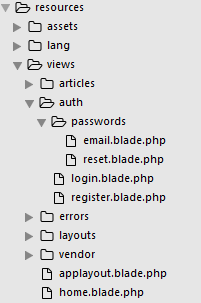
1. For **controllers**: Under the Auth directory, there are



1. For the **Eloquent Model**, we have the **User**.php

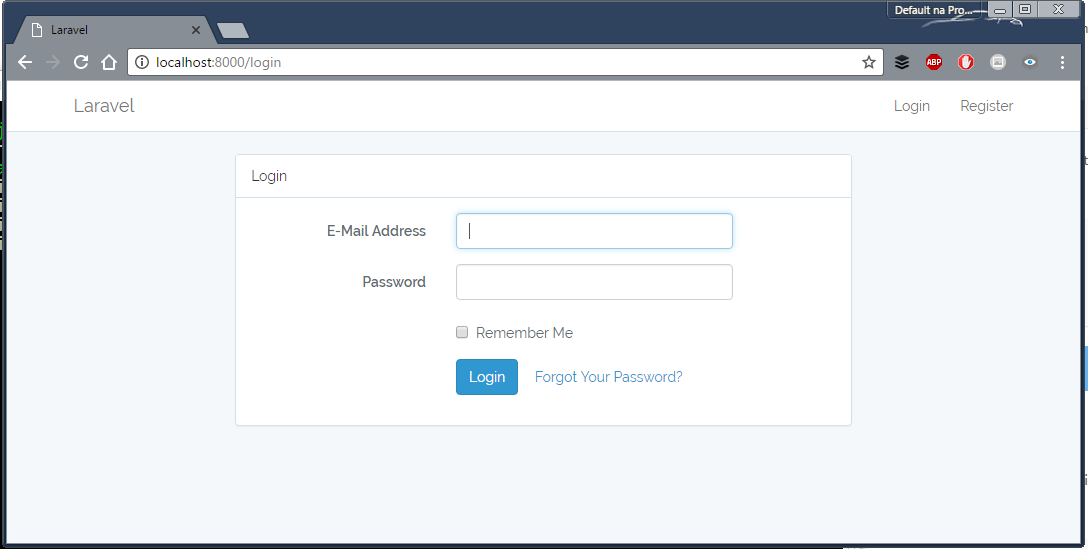


1. For **views**, they added additional views under auth directory,
   * email.blade.php
   * reset.blade.php
   * login.blade.php
   * register.blade.php
   * applayout.blade.php
   * home.blade.php



## Viewing the template for user signup

To view our created template. We can go to localhost:8000/login:

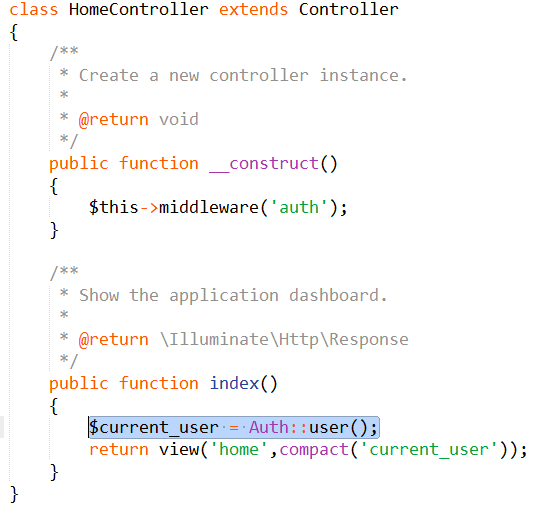


From here, we can create new user, login as the newly created user and logout.

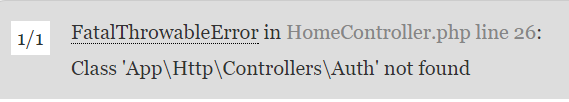
## Accessing the currently logged in user

To access the current logged in user, we use the **Auth::user()** facade.

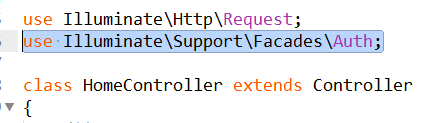
To demonstrate, we’ll use the HomeController.php to add our code



If this error appears:

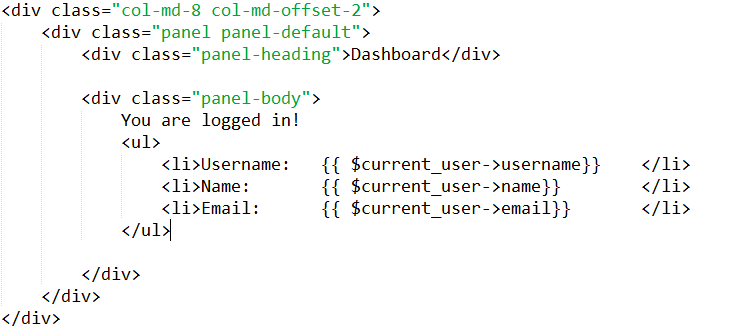


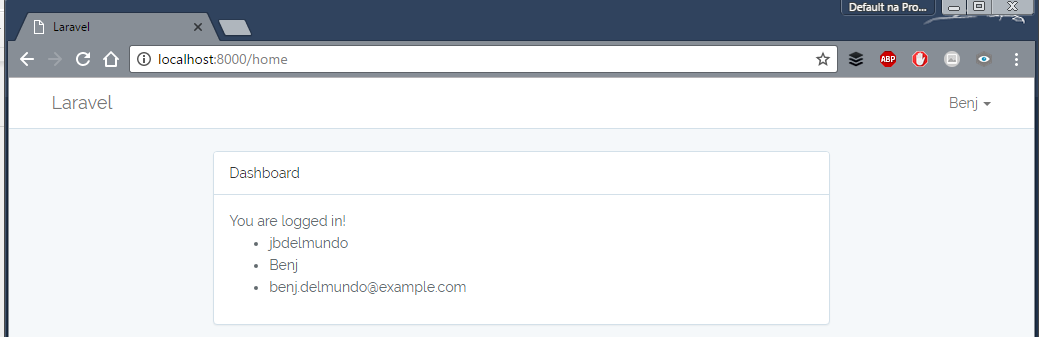
Add the following at the beginning of the controller

****

Then we use the **$current\_user** variable as an eloquent object

In **home.blade.php**:





## Protecting the access to controllers

If we want only the logged in user can access a specific controller, we add the auth middleware to the constructor of our controller

# 

Here, we can only access the method in HomeController only when we are logged in.

# Lesson 15 B: Customizing the User fields

Let’s say you want to add a new attribute to our user (ex. **nickname**), change the **name** attribute to **username** and make it as the login credential instead of the email.

**IT IS VERY IMPORTANT THAT YOU UNDERSTAND HOW THINGS ARE WORKING BEFORE YOU CAN CUSTOMIZE. :)**

To assess yourself, if you truly understand,

* Which functions in the controller is used in **creating a new user**.
* Which functions in the controller is used when a **user logs in**
* How the **users table** in the database is created
* How migrations work
* Difference of GET and POST routes

Ask assistance from your instructor if you need clarifications.

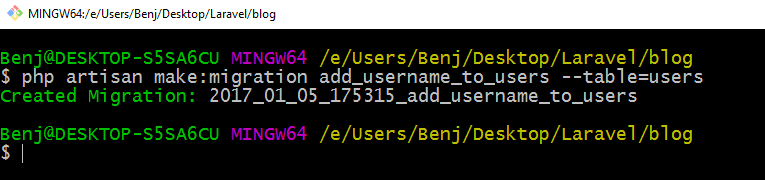
## Adding a new attribute to User model and changing the primary login credential.

Suppose we want to

1. Add a **username** field to the user
2. Use the username attribute to login instead of the email.

### Step 1: Adding the username column in the database

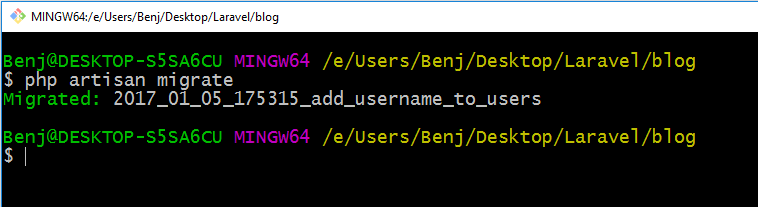
If the users table in the database is already created, we need to create a new migration file to add columns to our users table. We add the **--table=users** option to specify that we want to refer to the users table



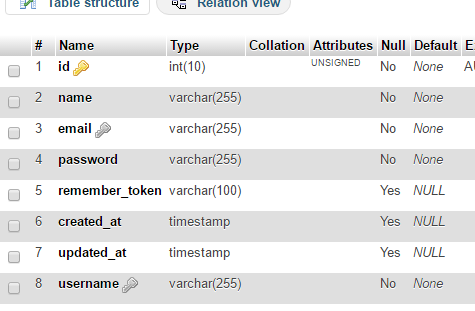
Then we edit the migration file. We add the **username** column as varchar and we want to make it **unique**.



And run the migration



We’ll see the changes in phpmyadmin



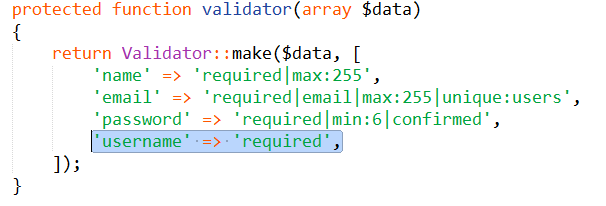
### Step 2: Updating our User.php model

Add the **username** in the **$fillable** array to be included in the list of attributes for our model

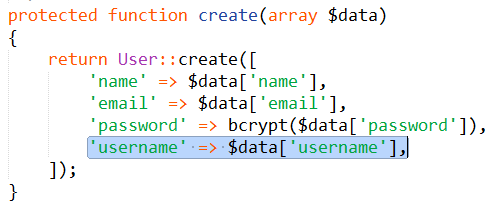


### Step 3: Updating our **RegisterController.php**

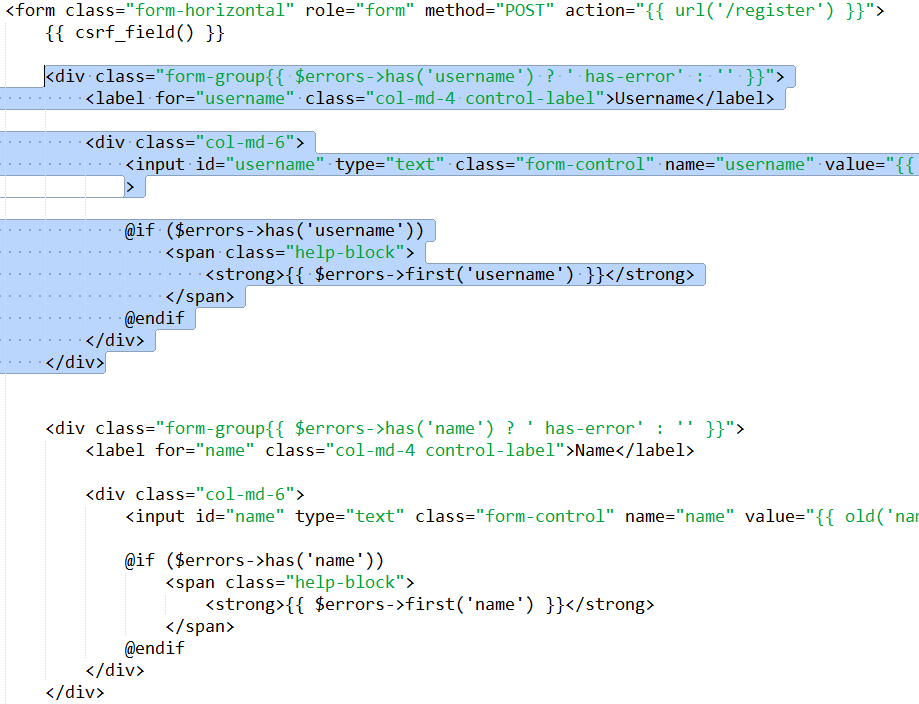
We add validation rules for our username field

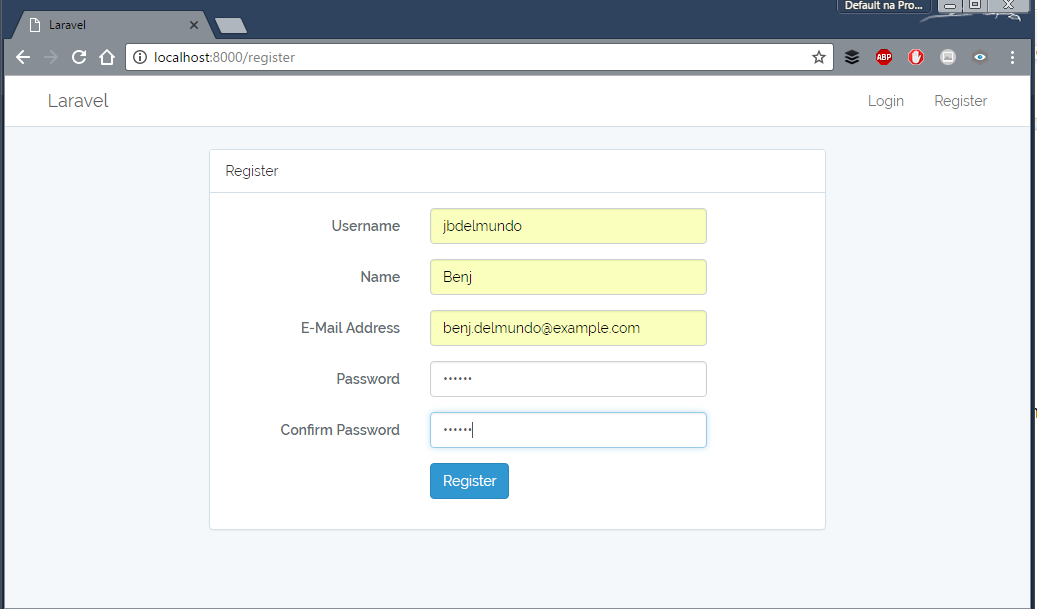


And add the username upon creating a user



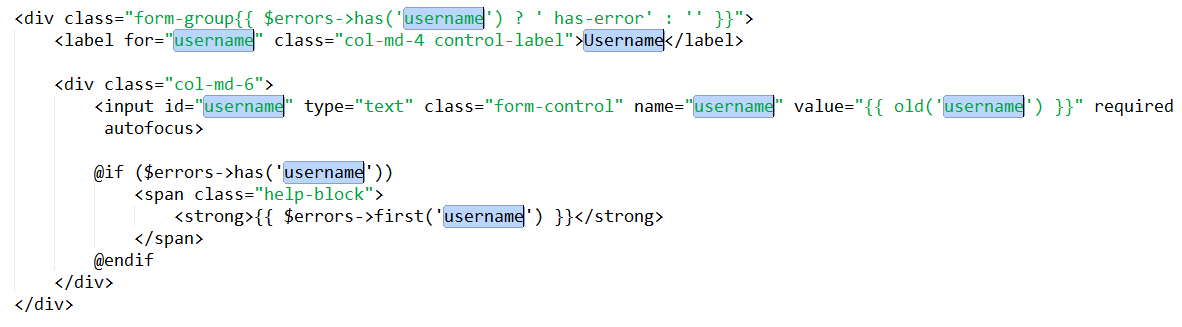
### Step 4: Updating our **register.blade.php** to add the username field in our form

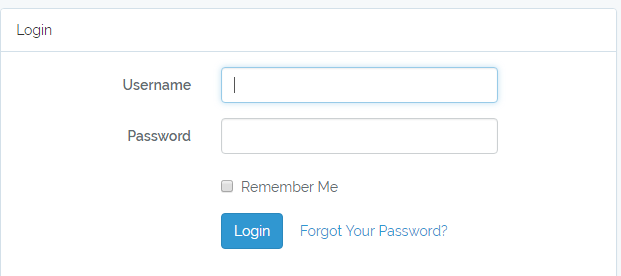
****

****

We have successfully added a new attribute to our user.

### Step 5: Updating our **login.blade.php** to change the **email** to **username** field in our form

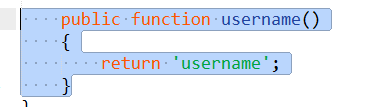
****

****

### Step 6: Updating our **LoginController.php**

### 

We override a method **username()**. This function tells us that we want to use the username attribute as the login credential instead of the email address.



# 

# 

# 

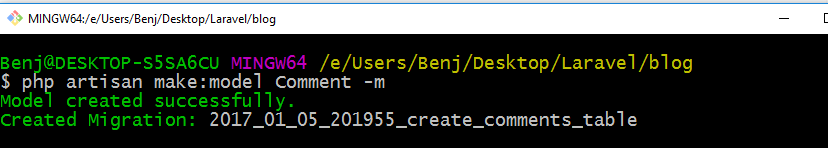
# Lesson 16: Eloquent Relationships

## Part 1: Many-to-One Relationship

### Creating a Comment Model

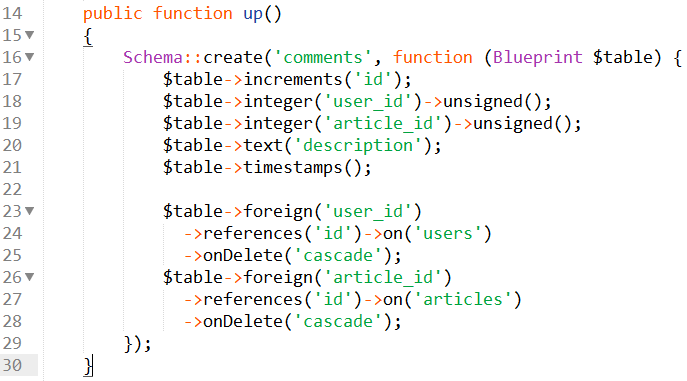
In this lesson we’ll add a new model Comment in addition to the our Articles.

Create a model and migration for Comment



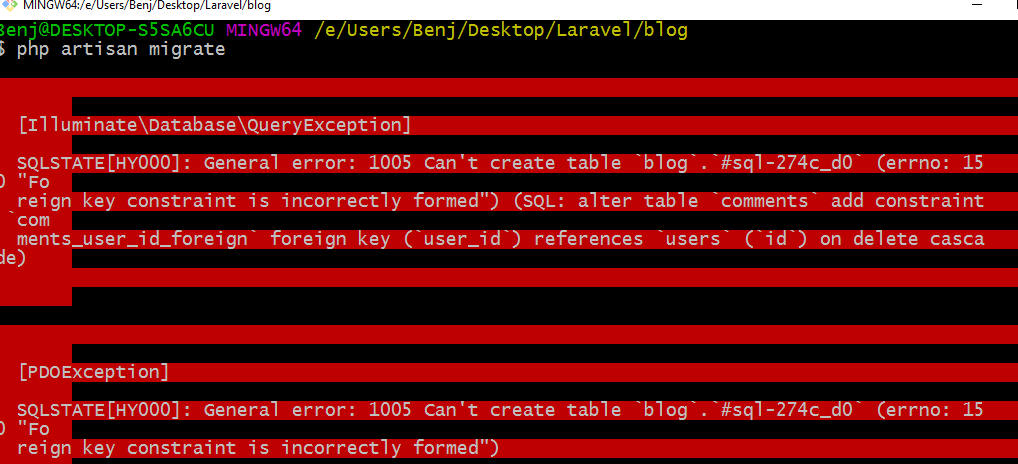
And on the migration file, we add

* user\_id (INTEGER), it is important that we make it **unsigned**!
* article\_id (INTEGER), it is important that we make it **unsigned**!
* description (TEXT)



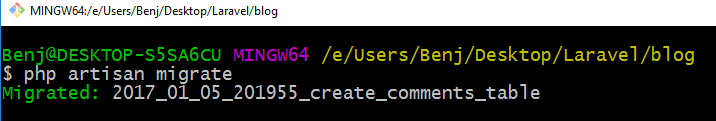
The lines 23-25 says that the column user\_id is a **foreign key** that references the id column on the users table.

Then we migrate:

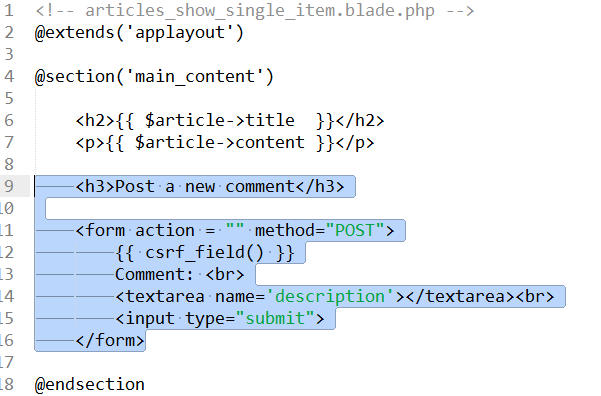


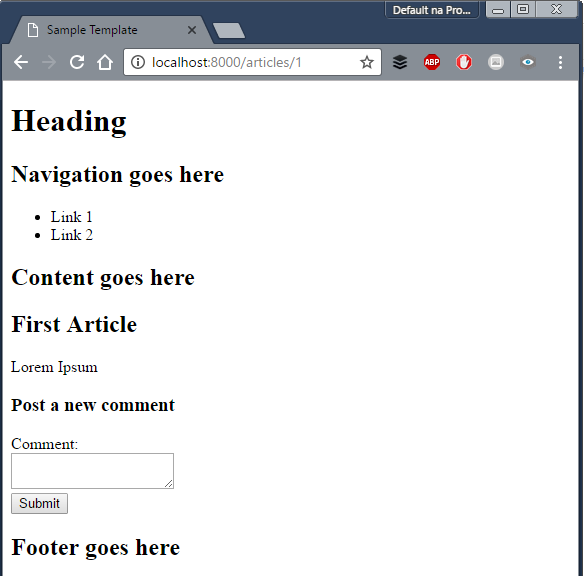
If we encounter **“Foreign key constraint incorrectly formed”** error in the future, check the following

* The table and column that the foreign key refers to exists
* The foreign key has unsigned integer data type



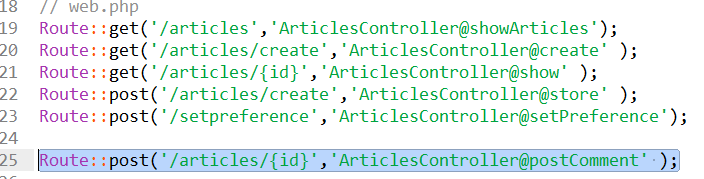
Then we create the form to add a new comment in **articles\_show\_single\_item.blade.php**:



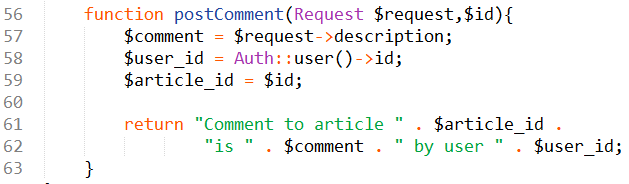


When we click submit, this will send a POST request to the current path*(/articles/{id})*

So we create a new route:

****

And add postComment() function in **ArticlesController.php**



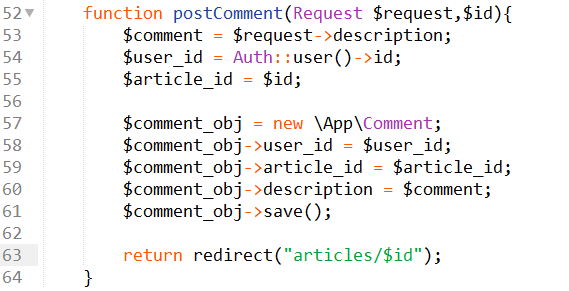
IMPORTANT: Line # 53 will give Trying to get property of non-object if there is no logged in user since ***Auth::user()*** is null.



Now we have 3 important things to store,

1. **id** of the current user
2. **id** of the article we want comment on
3. Body of the **comment**

Then we can save the 3 items on our database.

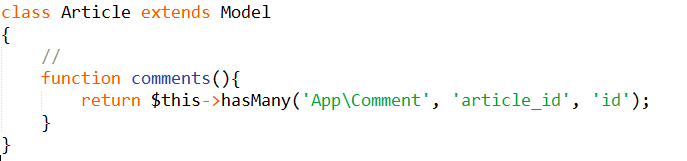


### Using Relationship with Comment Model

Now we can create relationship between **Article** and **Comment** models.

* An article can have many Comments
* A Comment belongs to only one Article

Then we add the following to **Article.php**



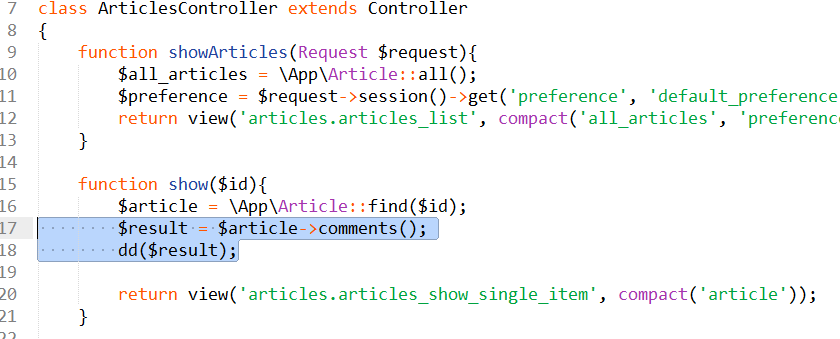
The hasMany() function has 3 parameters:

1. Model of the object
2. Foreign key (optional)
3. Local key (optional)

In our example, the 3rd parameter (*id*) will be matched in the foreign key (*article\_id*) in the *comments* table (table of the Comment model)

Now, any object that is an **Article** model object can use the **relationship** with **Comment** object.

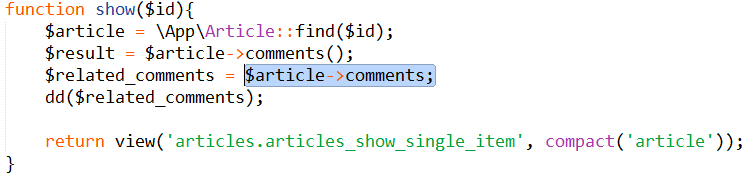
To test, we modify **temporarily** our ArticlesControler.php. We’ll use the dd() function to print variables.



When we print the result, we get a **HasMany relationship object** that is related to the *Comment* object.

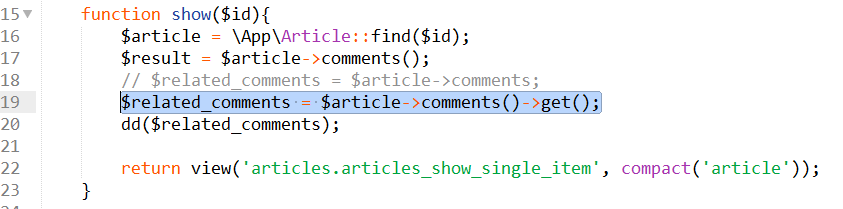


To get the **Collection of Comment objects** related to this article, we remove the parenthesis





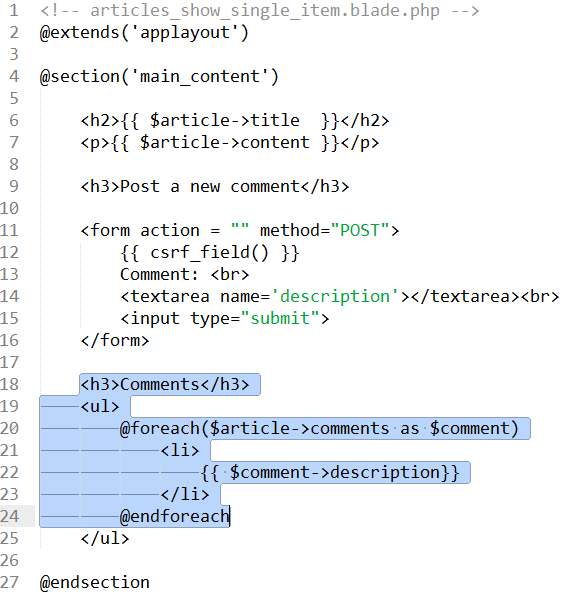
Or use the get() on the relationship



Both are equivalent: They return a Collection of Comment objects

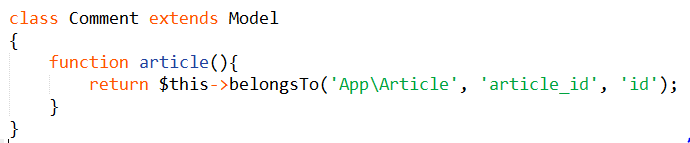
* $article->comments
* $article->comments()->get()

We can also use the relationship in **articles\_show\_single\_item.blade.php**



## Part 2: One-to-Many Relationship

For one-to-many, we use **belongsTo()**



The rest is the same.

## Part 3: One-to-One Relationship

For one-to-many, we use **hasOne()**

## Part 4: Many-Many Relationship

For one-to-many, we need a intermediate table to join the two models, then we use **belongsToMany()**

<https://laravel.com/docs/5.3/eloquent-relationships#many-to-many>

<https://laravel.com/docs/5.3/eloquent-relationships#updating-many-to-many-relationships>