LARAVEL BLOG GUIDE

How to create a blog site using Laravel 5.3

What is Laravel

Components of Laravel

- Routes
- View
- Controller
- Model

Installation of Laravel

Requirements:

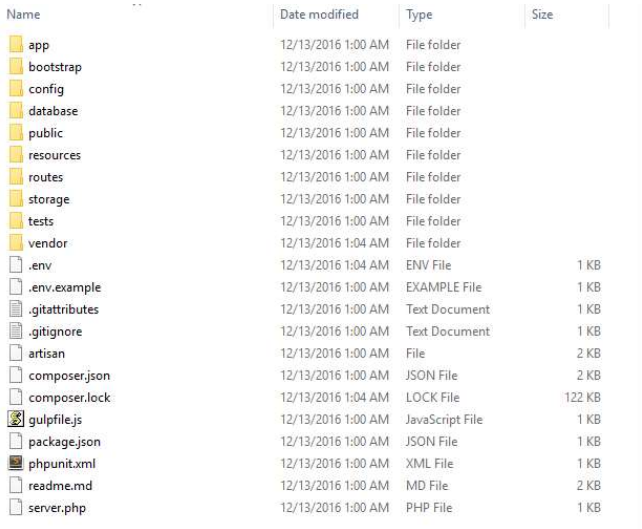
- Composer: getcomposer.org
- PHP 5.6.4 onwards (for Laravel 5.3 onwards)

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...

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



Name	Date modified	Type	Size
app	12/13/2016 1:00 AM	File folder	
bootstrap	12/13/2016 1:00 AM	File folder	
config	12/13/2016 1:00 AM	File folder	
database	12/13/2016 1:00 AM	File folder	
public	12/13/2016 1:00 AM	File folder	
resources	12/13/2016 1:00 AM	File folder	
routes	12/13/2016 1:00 AM	File folder	
storage	12/13/2016 1:00 AM	File folder	
tests	12/13/2016 1:00 AM	File folder	
vendor	12/13/2016 1:04 AM	File folder	
.env	12/13/2016 1:04 AM	ENV File	1 KB
.env.example	12/13/2016 1:00 AM	EXAMPLE File	1 KB
.gitattributes	12/13/2016 1:00 AM	Text Document	1 KB
.gitignore	12/13/2016 1:00 AM	Text Document	1 KB
artisan	12/13/2016 1:00 AM	File	2 KB
composer.json	12/13/2016 1:00 AM	JSON File	2 KB
composer.lock	12/13/2016 1:04 AM	LOCK File	122 KB
gulpfile.js	12/13/2016 1:00 AM	JavaScript File	1 KB
package.json	12/13/2016 1:00 AM	JSON File	1 KB
phpunit.xml	12/13/2016 1:00 AM	XML File	1 KB
readme.md	12/13/2016 1:00 AM	MD File	2 KB
server.php	12/13/2016 1:00 AM	PHP File	1 KB

Laracasts: Laravel 5 Fundamentals

If you need video tutorials
https://laracasts.com/series/laravel-5-fundamentals

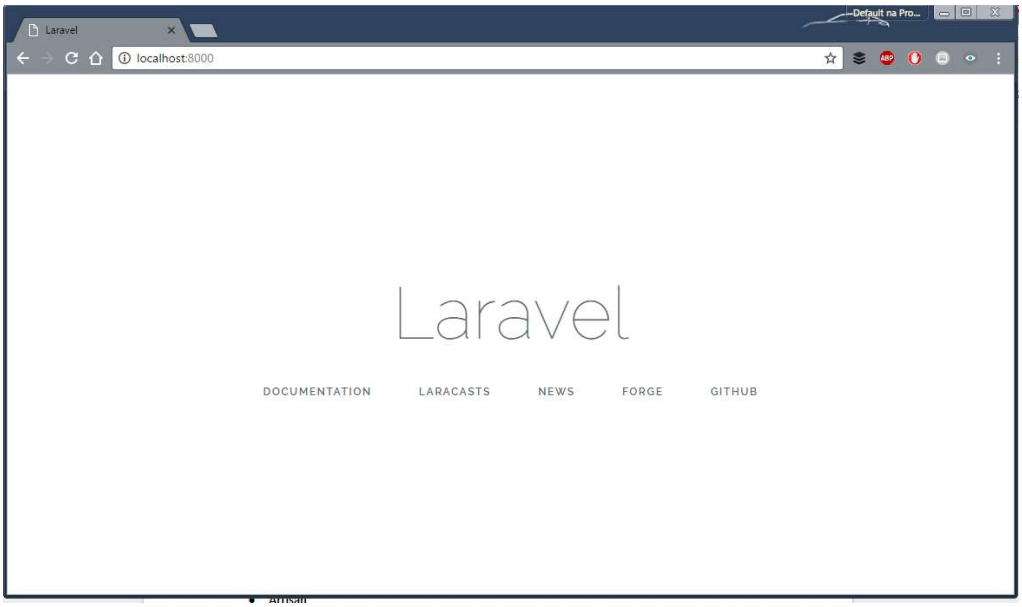
Lesson 1: Accessing your first Laravel website

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

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



Lesson 2: Routes

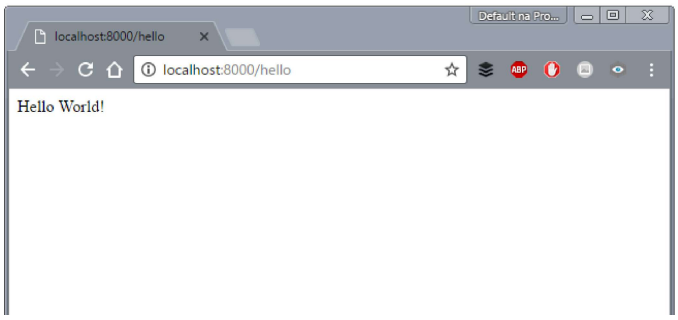
The routes file determine what to do upon receiving a particular URL.
For version 5.3, the routes file is located in **routes/web.php**

```php
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| This file is where you may define all of the routes that are handled
| by your application. Just tell Laravel the URIs it should respond
| to using a Closure or controller method. Build something great!
|
*/

Route::get('/', function () {
    return view('welcome');
});

```

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.

```php
*/

Route::get('/', function () {
    return view('welcome');
});


Route::get('/hello', function () {
    return "Hello World!";
});

```



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.php

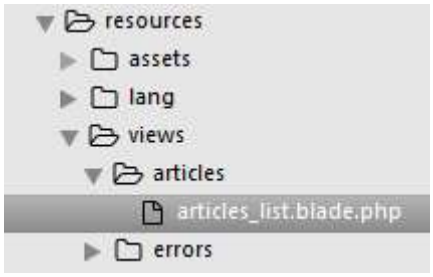
```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>Laravel</title>

        <!-- Fonts -->
        <link href="https://fonts.googleapis.com/css?family=Raleway:100,600" rel="
        stylesheet" type="text/css">

        <!-- Styles -->
        <style>
            html, body {
                background-color: #fff;
                color: #636b6f;
                font-family: 'Raleway', sans-serif;
                font-weight: 100;
                height: 100vh;
                margin: 0;
            }

            .full-height {
                height: 100vh;
```

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

```
1  <!-- articles_list.blade.php -->
2
3  <h3>List of Articles</h3>
```

Contents of article_list.blade.php

```
18 // web.php
19 Route::get('/articles', function () {
20     return view('articles.articles_list');
21 });
22
```

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

```
18 // web.php
19 Route::get('/articles', function () {
20     $article1 = 'Tutorial';
21     $article2 = 'Getting started';
22     return view('articles.articles_list', compact('article1', 'article2'));
23 });
```

Printing variables in views

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

```
1  <!-- articles_list.blade.php -->
2
3  <h3>List of Articles</h3>
4  <ul>
5      <li>{{ $article1 }}</li>
6      <li>{{ $article2 }}</li>
7  </ul>
```

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

```blade
<!-- applayout.blade.php -->
<!DOCTYPE html>
<html>
<head>
    <title>Sample Template</title>
</head>
<body>
    <h1>Heading</h1>
    <div>
        <h2>Navigation goes here</h2>
        <ul>
            <li>Link 1</li>
            <li>Link 2</li>
        </ul>
    </div>

    <div>
        <h2>Content goes here</h2>
        @yield("main_content")
    </div>

    <div>
        <h2>Footer goes here</h2>
    </div>

</body>
</html>
```

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.

```blade
<!-- articles_list.blade.php -->
@extends('applayout')

@section('main_content')

    <h3>List of Articles</h3>
    <ul>
        <li>{{ $article1 }}</li>
        <li>{{ $article2 }}</li>
    </ul>

@endsection
```

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

```
18  // web.php
19  Route::get('/articles','ArticlesController@showArticles');
```

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

2. 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)

```
1  <!-- ArticlesController.php -->
2  <?php
3
4  namespace App\Http\Controllers;
5
6  use Illuminate\Http\Request;
7
8  class ArticlesController extends Controller
9  {
10     function showArticles(){
11         $article1 = 'Tutorial';
12         $article2 = 'Getting started';
13         return view('articles.articles_list', compact('article1', 'article2'));
14     }
15 }
```

Lesson 5: Environment and Configurations

The environment and configuration files are:

- **config/database.php**

```php
'connections' => [

    'sqlite' => [
        'driver' => 'sqlite',
        'database' => env('DB_DATABASE', database_path('database.sqlite')),
        'prefix' => '',
    ],

    'mysql' => [
        'driver' => 'mysql',
        'host' => env('DB_HOST', 'localhost'),
        'port' => env('DB_PORT', '3306'),
        'database' => env('DB_DATABASE', 'forge'),
        'username' => env('DB_USERNAME', 'forge'),
        'password' => env('DB_PASSWORD', ''),
        'charset' => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix' => '',
        'strict' => true,
        'engine' => null,
    ],

    'pgsql' => [
```

- **.env file**

```
APP_ENV=local
APP_KEY=base64:aNRmig6agm/OdGiAxfqBpAn2alCAHV9qzDHeWTjV9Yc=
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=blog
DB_USERNAME=root
DB_PASSWORD=

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379
```

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

```php
<?php
// 2016_12_12_185602_createArticlesTable.php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateArticlesTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('articles', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('articles');
    }
}
```

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:

```
15      public function up()
16      {
17          Schema::create('articles', function (Blueprint $table) {
18              $table->increments('id');
19              $table->string('title');
20              $table->string('content');
21              $table->timestamps();
22          });
23      }
```

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

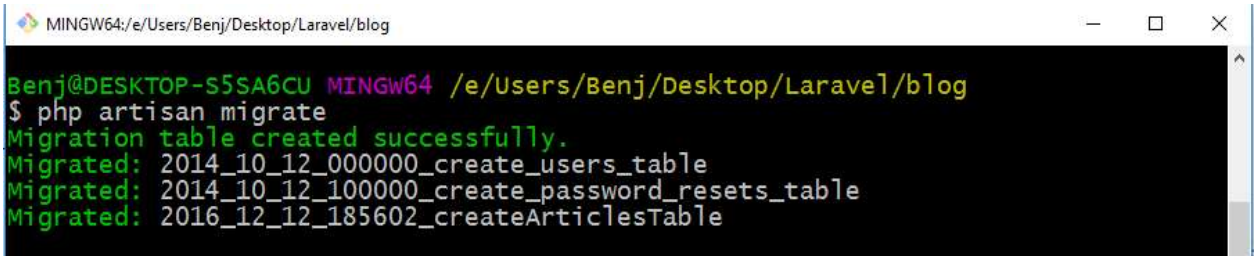
Step 3: Create a database

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

```
7   DB_CONNECTION=mysql
8   DB_HOST=127.0.0.1
9   DB_PORT=3306
10  DB_DATABASE=blog
11  DB_USERNAME=root
12  DB_PASSWORD=
```

Step 4: Run the migration

Type ***php artisan migrate***



This will create tables in our database

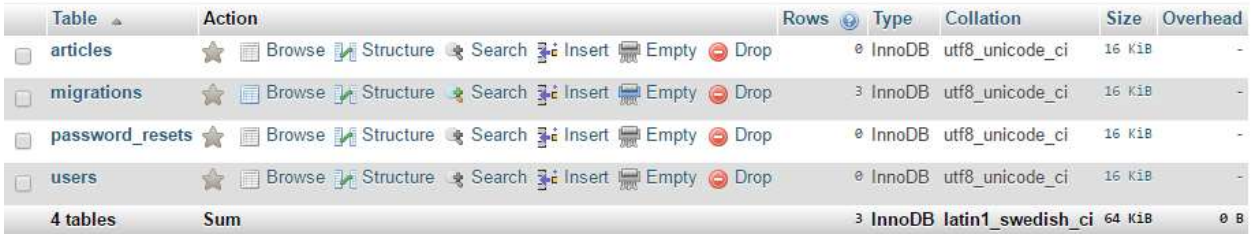


Table	Action	Rows	Type	Collation	Size	Overhead
articles	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	16 KiB	-
migrations	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8_unicode_ci	16 KiB	-
password_resets	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	16 KiB	-
users	Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	16 KiB	-
4 tables	Sum	3	InnoDB	latin1_swedish_ci	64 KiB	0 B

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

```php
<?php
// Article.php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    //
}
```

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

```
{
    "name": "laravel/laravel",
    "description": "The Laravel Framework.",
    "keywords": ["framework", "laravel"],
    "license": "MIT",
    "type": "project",
    "require": {
        "php": ">=5.6.4",
        "laravel/framework": "5.4.*",
        "laravel/tinker": "~1.0"
    },
    "require-dev": {
        "fzaninotto/faker": "~1.4",
        "mockery/mockery": "0.9.*",
        "phpunit/phpunit": "~5.7"
    },
```

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

To run tinker

php artisan tinker

We can run PHP lines

```
MINGW64:/e/Users/Benj/Desktop/Laravel/blog
Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$ php artisan tinker
Psy Shell v0.7.2 (PHP 7.0.9 rçö cli) by Justin Hileman
>>> $sum = 1 + 2 + 3;
=> 6
>>>
```

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

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

```
MINGW64:/e/Users/Benj/Desktop/Laravel/blog
Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$ php artisan tinker
Psy Shell v0.7.2 (PHP 7.0.9 rçö cli) by Justin Hileman
>>> $article_obj = new App\Article()
=> App\Article {#662}
>>>
```

Set the *title* and *content* properties

```
MINGW64:/e/Users/Benj/Desktop/Laravel/blog
Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$ php artisan tinker
Psy Shell v0.7.2 (PHP 7.0.9 rçö cli) by Justin Hileman
>>> $article_obj = new App\Article()
=> App\Article {#662}
>>> $article_obj->title = "First Article";
=> "First Article"
>>> $article_obj->content = "Lorem Ipsum";
=> "Lorem Ipsum"
>>>
```

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

```
>>> $article_obj->save()
=> true
>>>
```

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

←T→	id	title	content	created_at	updated_at
Edit Copy Delete	1	First Article	Lorem Ipsum	2016-12-12 19:43:29	2016-12-12 19:43:29

Check all *With selected:* Edit Copy Delete Export

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

```
>>> $article_obj
=> App\Article {#662
     title: "First Article",
     content: "Lorem Ipsum",
     updated_at: "2016-12-12 19:43:29",
     created_at: "2016-12-12 19:43:29",
     id: 1,
   }
>>>
```

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()**

```
>>> $article1 = App\Article::find(1)
=> App\Article {#679
     id: 1,
     title: "First Article",
     content: "Lorem Ipsum",
     created_at: "2016-12-12 19:43:29",
     updated_at: "2016-12-12 19:43:29",
   }
>>>
```

```
>>> $article2 = App\Article::find(2)
=> App\Article {#680
     id: 2,
     title: "Second Article",
     content: "This is 2nd Article...",
     created_at: "2016-12-12 19:52:45",
     updated_at: "2016-12-12 19:52:45",
   }
>>>
```

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

```
>>> $title1 = $article1->title
=> "First Article"
>>>
```

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

```
>>> $result = App\Article::where('title','First Article')
=> Illuminate\Database\Eloquent\Builder {#671}
>>>
```

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.

```
class ArticlesController extends Controller
{
    function showArticles(){
        $all_articles = \App\Article::all();
        return view('articles.articles_list', compact('all_articles'));
    }
}
```

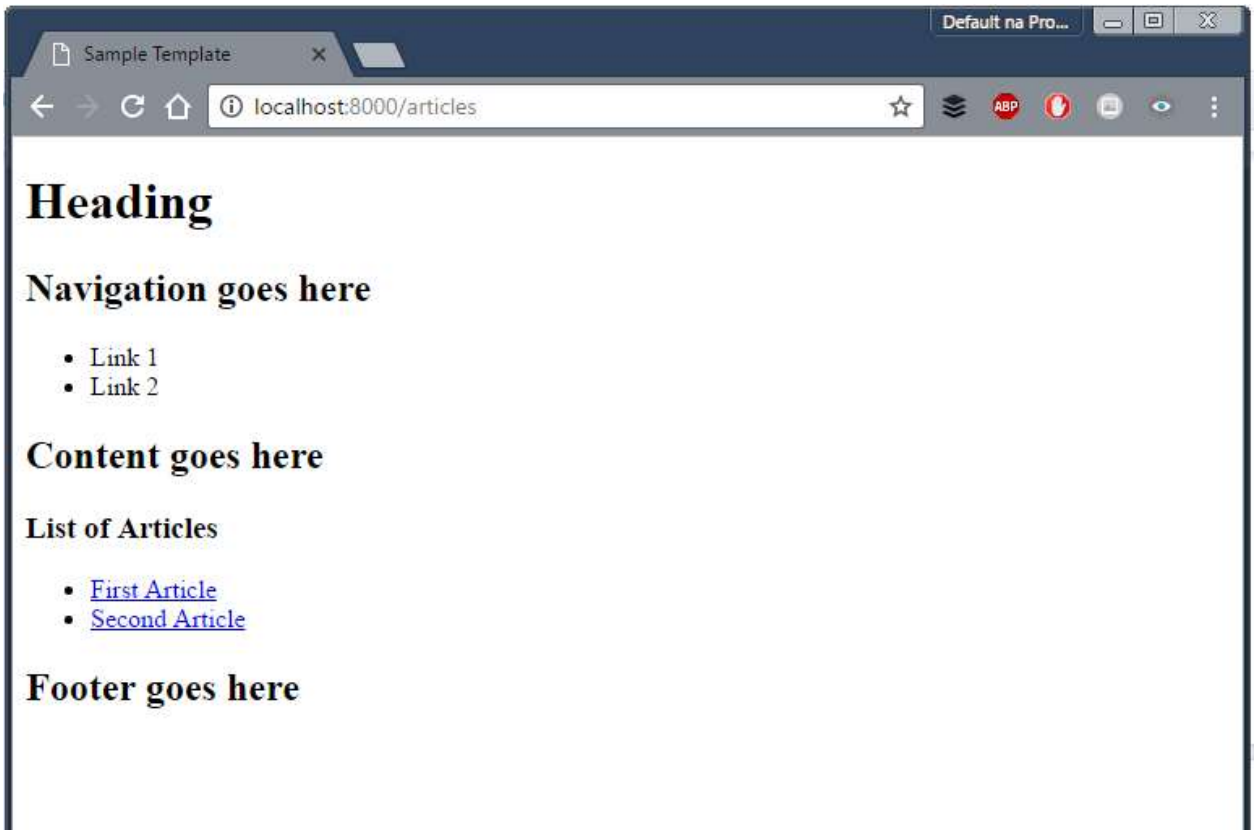
We update our ArticlesController to retrieve all articles from the database

```
1  <!-- articles_list.blade.php -->
2  @extends('applayout')
3
4  @section('main_content')
5
6      <h3>List of Articles</h3>
7      <ul>
8          @foreach($all_articles as $article)
9              <li>
10                 <a href="#">{{ $article }}</a>
11             </li>
12         @endforeach
13     </ul>
14
15 @endsection
```

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.

```
1  <!-- articles_list.blade.php -->
2  @extends('applayout')
3
4  @section('main_content')
5
6      <h3>List of Articles</h3>
7      <ul>
8          @foreach($all_articles as $article)
9              <li>
10                 <a href="#">{{ $article->title }}</a>
11             </li>
12         @endforeach
13     </ul>
14
15 @endsection
```


Default na Pro...
Sample Template
localhost:8000/articles
Heading
Navigation goes here
Link 1
Link 2
Content goes here
List of Articles
First Article
Second Article
Footer goes here

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

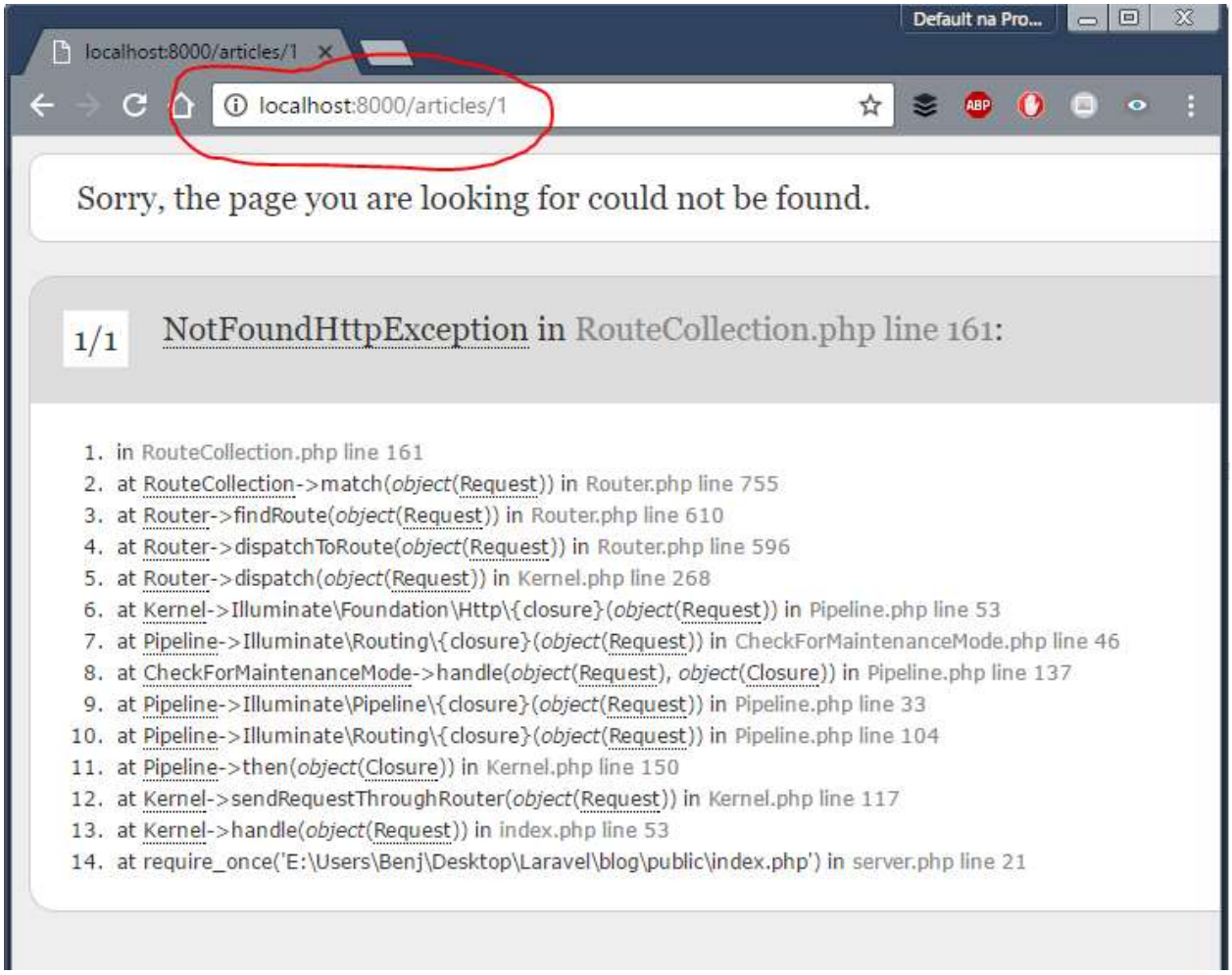
- First Article -> localhost:8000/articles/1
- Second Article -> localhost:8000/articles/2

We'll update our href tags:

```
<!-- articles_list.blade.php -->
@extends('applayout')

@section('main_content')

    <h3>List of Articles</h3>
    <ul>
        @foreach($all_articles as $article)
            <li>
                <a href= '{{url( "articles/$article->id" )}}'>{{ $article->title }}</a>
            </li>
        @endforeach
    </ul>

@endsection
```

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

```
18  // web.php
19  Route::get('/articles','ArticlesController@showArticles');
20  Route::get('/articles/{id}','ArticlesController@show');
21
```

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

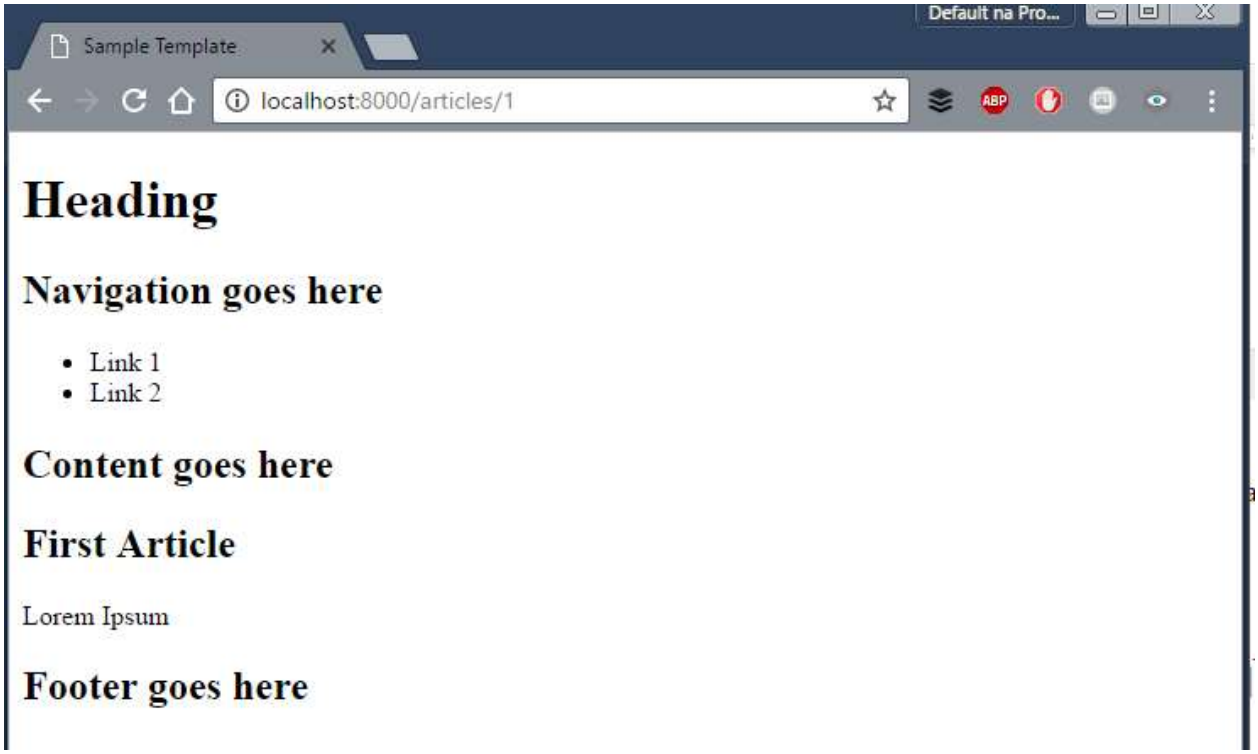
```
class ArticlesController extends Controller
{
    function showArticles(){
        $all_articles = \App\Article::all();
        return view('articles.articles_list', compact('all_articles'));
    }

    function show($id){

    }
}
```

We retrieve the article object with the given id and pass it to our new view: **articles_show_single_item.blade.php.**

```
function show($id){
    $article = \App\Article::find($id);
    return view('articles.articles_show_single_item', compact('article'));
}
```

```
1  <!-- articles_show_single_item.blade.php -->
2  @extends('applayout')
3
4  @section('main_content')
5
6      <h2>{{ $article->title  }}</h2>
7      <p>{{ $article->content }}</p>
8
9  @endsection
```

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

```
18  // web.php
19  Route::get('/articles','ArticlesController@showArticles');
20  Route::get('/articles/{id}','ArticlesController@show' );
21
```

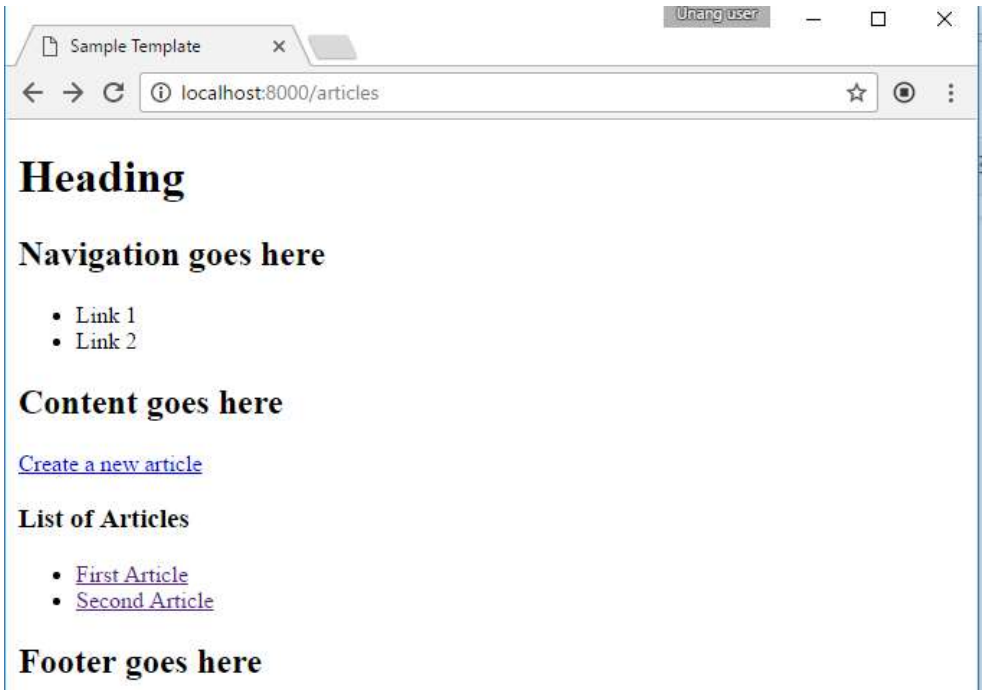
2. Adding a parameter in our controller function

```
7  class ArticlesController extends Controller
8  {
9      function showArticles(){
10         $all_articles = \App\Article::all();
11         return view('articles.articles_list', compact('all_articles'));
12     }
13
14     function show($id){
15         $article = \App\Article::find($id);
16         return view('articles.articles_show_single_item', compact('article'));
17     }
18 }
```

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




```
<!-- articles_list.blade.php -->
@extends('applayout')

@section('main_content')

    <h2>
        <a href={{url("articles/create")}}>Create a new article</a>
    </h2>
    <h3>List of Articles</h3>
    <ul>
        @foreach($all_articles as $article)
            <li>
                <a href= '{{url( "articles/$article->id" )}}'>{{ $article->title }}</a>
            </li>
        @endforeach
    </ul>

@endsection
```

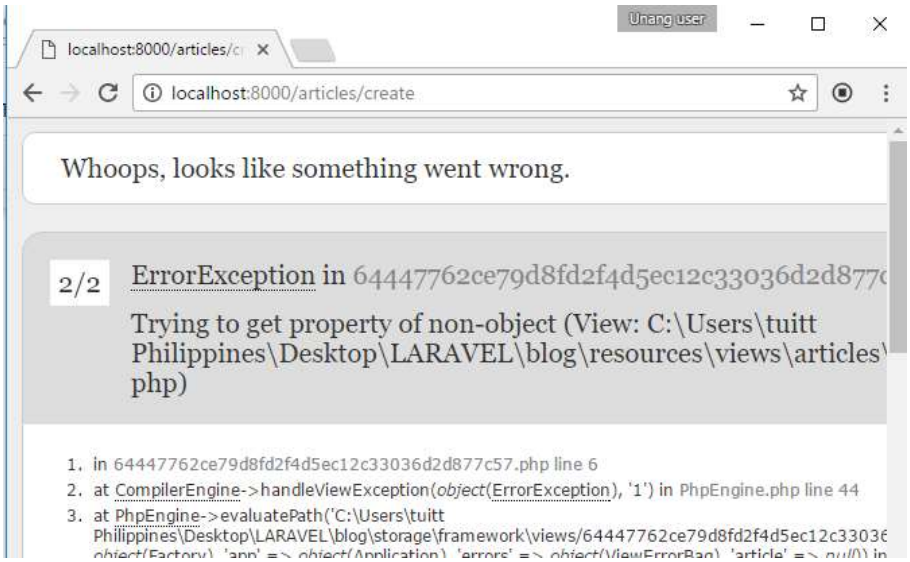
We also update the web.php route file

```
18  // web.php
19  Route::get('/articles','ArticlesController@showArticles');
20  Route::get('/articles/{id}','ArticlesController@show' );
21  Route::get('/articles/create','ArticlesController@create' );
```

And add a function create in ArticlesController:

```
class ArticlesController extends Controller
{
    function showArticles(){
        $all_articles = \App\Article::all();
        return view('articles.articles_list', compact('all_articles'));
    }

    function show($id){
        $article = \App\Article::find($id);
        return view('articles.articles_show_single_item', compact('article'));
    }

    function create(){
        return "Create a new article";
    }
}
```

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.

```
// web.php
Route::get('/articles','ArticlesController@showArticles');
Route::get('/articles/{id}','ArticlesController@show' );
Route::get('/articles/create','ArticlesController@create' );
```

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

```
// web.php
Route::get('/articles','ArticlesController@showArticles');
Route::get('/articles/create','ArticlesController@create' );
Route::get('/articles/{id}','ArticlesController@show' );
```

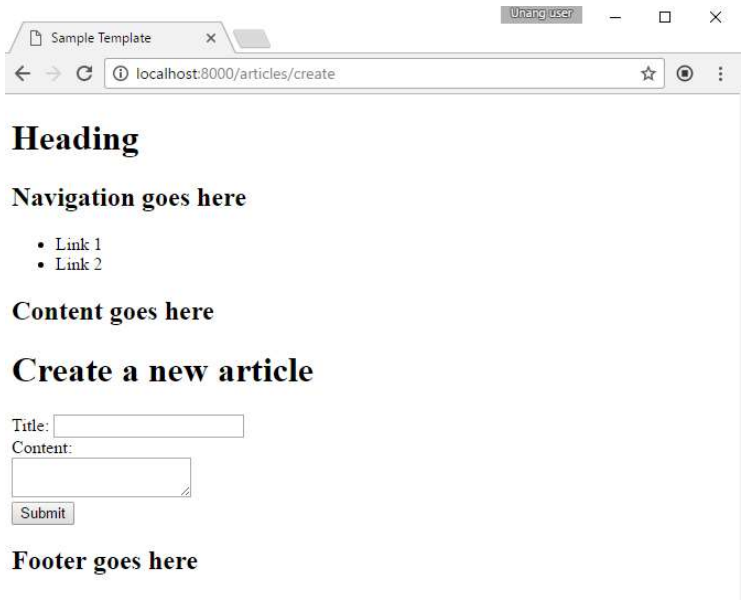
Next is we create a View for our form. We save it as **articles_create.blade.php** in the articles directory.

```
@extends('applayout')

@section('main_content')

    <h1>
        Create a new article
    </h1>

    <form>
        Title: <input type="text" name="title"> <br>
        Content: <br>
        <textarea name="content"></textarea><br>
        <input type="submit">
    </form>

@endsection
```

We update our ArticlesController to return our view:

```php
class ArticlesController extends Controller
{
    //

    function showArticles(){
        $all_articles = \App\Article::all();

        return view('articles.article_list',compact('all_articles'));
    }

    function show($id){
        $article = \App\Article::find($id);

        return view('articles.article_show_single_item', compact('article'));
    }

    function create(){
        return view('articles.article_create');
    }
```

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.

```
<form action="" method="POST">
    Title: <input type="text" name="title"> <br>
    Content: <br>
    <textarea name="content"></textarea><br>
    <input type="submit">
</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.

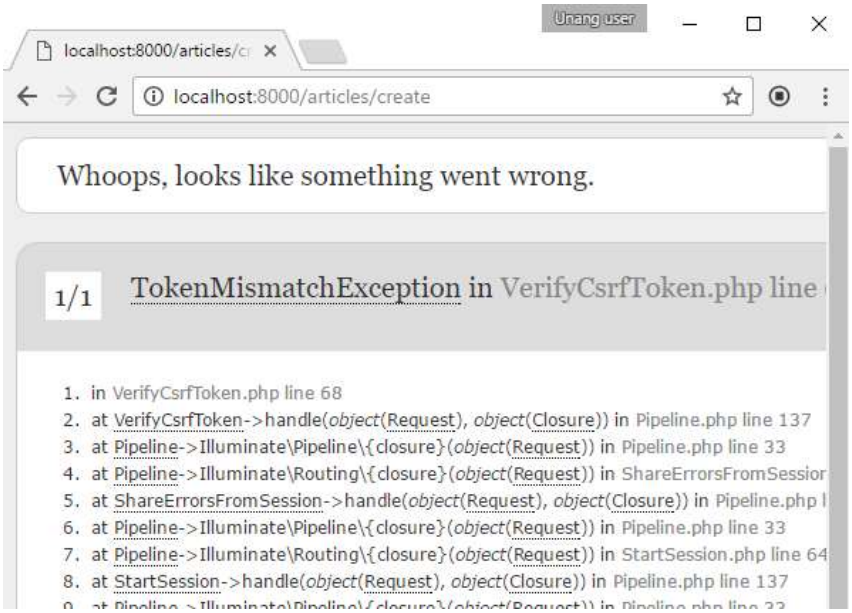
We update our **routes** to handle the POST request to localhost:8000/articles/new and create a function ***store()*** in ArticlesController

```
// web.php
Route::get('/articles','ArticlesController@showArticles');
Route::get('/articles/create','ArticlesController@create' );
Route::get('/articles/{id}','ArticlesController@show' );
Route::post('articles/create', 'ArticlesController@store');
```

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.

```
class ArticlesController extends Controller
{
    function showArticles(){
        $all_articles = \App\Article::all();
        return view('articles.articles_list', compact('all_articles'));
    }

    function show($id){
        $article = \App\Article::find($id);
        return view('articles.articles_show_single_item', compact('article'));
    }

    function create(){
        return view('articles.articles_new');
    }

    function store(){
        return "You clicked the submit button!";
    }
}
```

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.

```
<form action="" method="POST">
    {{ csrf_field() }}
    Title: <input type="text" name="title"> <br>
    Content: <br>
    <textarea name="content"></textarea><br>
    <input type="submit">
</form>
```

And we try to submit again:

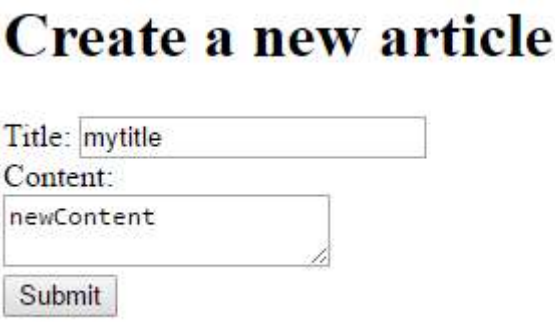
Our form is now accepted and our controller gives us the proper response.

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:

```
function store(Request $request){
    dd($request);
    return "You clicked the submit button!";
}
```



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

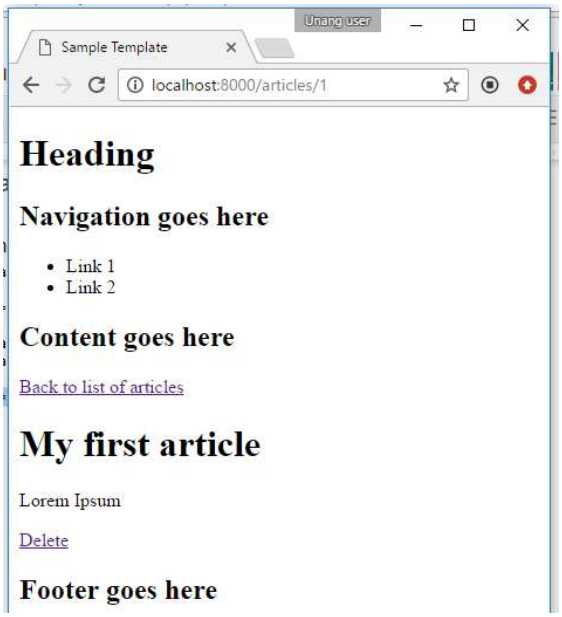
```php
function store(Request $request){
    print_r($request->get('title'));
    echo "<br>";
    print_r($request->get('content'));
    echo "<br>";
    return "You clicked the submit button!";
}
```

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

```php
function store(Request $request){
    $title = $request->get('title');
    $content = $request->get('content');

    $article_obj = new \App\Article();
    $article_obj->title = $title;
    $article_obj->content = $content;
    $article_obj->save();

    return view('articles.article_create');
}
```

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**

```
@section('main_content')

    <a href='{{url("articles")}}'>Back to list of articles</a>

    <h1>{{$article->title}}</h1>
    <p>{{ $article->content}}</p>

    <a href='{{url("articles/$article->id/delete")}}'>Delete</a>

@endsection
```

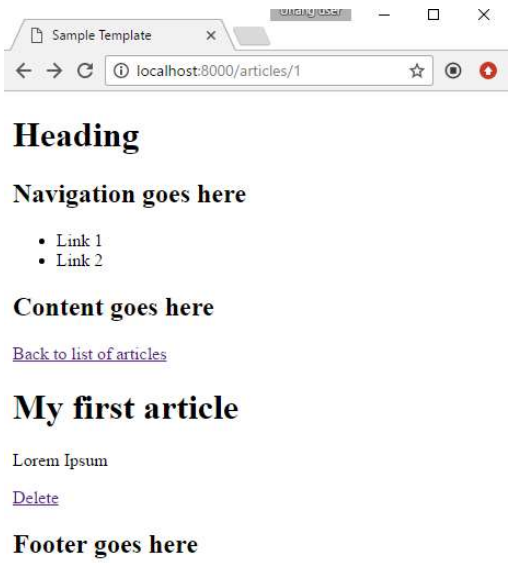
We create a new **route** for the delete and a function **delete()** in our controller:

```
Route::get('/articles', 'ArticlesController@showArticles');
Route::get('/articles/create', 'ArticlesController@create');
Route::get('/articles/{id}', 'ArticlesController@show');
Route::post('/articles/create', 'ArticlesController@store');
Route::get('/articles/{id}/delete', 'ArticlesController@delete');
```

ArticlesController.php:

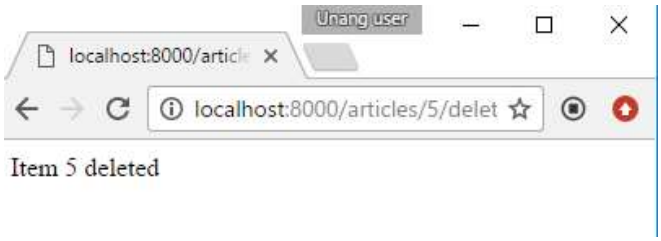
```
function delete($id){
    return "You want to delete article with id: $id";
}
```

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.**

```
function delete($id){
    $article_to_delete = \App\Article::find($id);
    $article_to_delete->delete();

    return "Item  $id deleted";
}
```



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

```
function delete($id){
    $article_to_delete = \App\Article::find($id);
    $article_to_delete->delete();

    return redirect("articles");
}
```

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

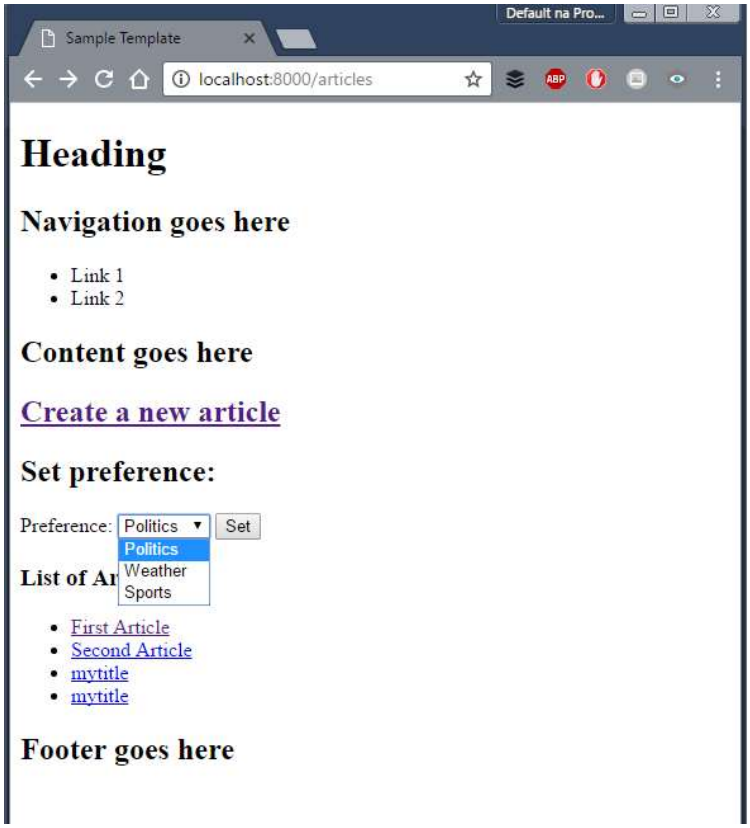
Session variables are variables that can be accessed globally by any webpage within the website. In plain PHP, we use the $_SESSION superglobal variable to access the variable. In laravel, accessing the session variables and keys are similar.

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.


Default na Pro...
Sample Template
localhost:8000/articles
Heading
Navigation goes here
Link 1
Link 2
Content goes here
Create a new article
Set preference:
Preference: Politics Set
Politics
Weather
Sports
List of Ar
First Article
Second Article
mytitle
mytitle
Footer goes here

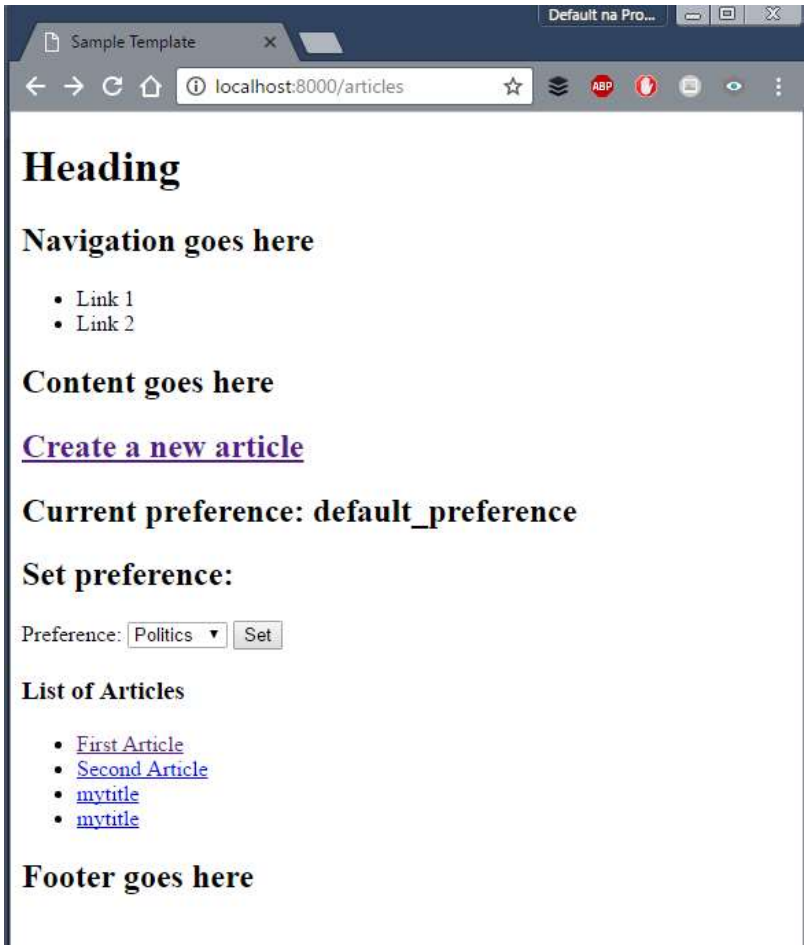
```blade
<!-- articles_list.blade.php -->
@extends('applayout')

@section('main_content')

    <h2>
        <a href='{{url("articles/create")}}'>Create a new article</a>
    </h2>

    <h2>Set preference:</h2>
    <form action='{{url("setpreference")}}' method="POST">
        {{ csrf_field() }}
        Preference:
        <select name='preference'>
            <option value="politics">Politics</option>
            <option value="weather">Weather</option>
            <option value="sports">Sports</option>
        </select>
        <input type="submit" value="Set">
    </form>

    <h3>List of Articles</h3>
    <ul>
        @foreach($all_articles as $article)
            <li>
                <a href= '{{url( "articles/$article->id" )}}'>{{ $article->title }}</a>
            </li>
        @endforeach
    </ul>

@endsection
```

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.

```php
class ArticlesController extends Controller
{
    function showArticles(Request $request){
        $all_articles = \App\Article::all();
        $preference = $request->session()->get('preference', 'default_preference');
        return view('articles.articles_list', compact('all_articles', 'preference'));
    }
```

And we print it in **articles_list.blade.php**

```
@section('main_content')

    <h2>
        <a href='{{url("articles/create")}}'>Create a new article</a>
    </h2>

    <h2>Current preference: {{ $preference }}</h2>
    <h2>Set preference:</h2>
    <form action='{{url("setpreference")}}' method="POST">
        {{ csrf_field() }}
        Preference:
        <select name='preference'>
            <option value="politics">Politics</option>
            <option value="weather">Weather</option>
            <option value="sports">Sports</option>
        </select>
        <input type="submit" value="Set">
    </form>

    <h3>List of Articles</h3>
    <ul>
        @foreach($all_articles as $article)
            <li>
                <a href= '{{url( "articles/$article->id" )}}'>{{ $article->title }}</a>
            </li>
        @endforeach
    </ul>
```



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

```
// web.php
Route::get('/articles','ArticlesController@showArticles');
Route::get('/articles/create','ArticlesController@create' );
Route::get('/articles/{id}','ArticlesController@show' );
Route::post('/articles/create','ArticlesController@store' );
Route::post('/setpreference','ArticlesController@setPreference');
```

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

```
function setPreference(Request $request){
    $preference = $request->get('preference');

    return "You set preference to $preference";
}
```

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**

```php
function setPreference(Request $request){
    $preference = $request->get('preference');

    $request->session()->put('preference',$preference);

    return redirect("articles");
}
```

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.

```
function store(Request $request){
    $title = $request->get('title');
    $content = $request->get('content');

    $rules = array(
        'title' => 'required | min:3 | max:10 | alpha_num',
        'content' => 'required'
    );


    $article_obj = new \App\Article();
    $article_obj->title = $title;
    $article_obj->content = $content;
    $article_obj->save();

    return view('articles.article_create');
}
```

.

For more validation rules:
https://laravel.com/docs/5.3/validation#available-validation-rules

Then we add the validation of the POST request

```
24      function store(Request $request){
25          $title = $request->get('title');
26          $content = $request->get('content');
27
28          $rules = array(
29              'title' => 'required | min:3 | max:10 | alpha_num',
30              'content' => 'required'
31          );
32          $this->validate($request,$rules);
33
34
35          $article_obj = new \App\Article();
36          $article_obj->title = $title;
37          $article_obj->content = $content;
38          $article_obj->save();
39
40          return view('articles.article_create');
41      }
```

If the validation is **passed**, the execution continues (**l**ines 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**

```blade
<!-- articles_create.blade.php -->
@extends('applayout')

@section('main_content')

    <a href='{{url("articles")}}'>Back</a>

    @if (count($errors) > 0)
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    @endif


    <h1>
        Create a new article
    </h1>

    <form action = "" method="POST">
        {{ csrf_field() }}
        Title: <input type="text" name="title"><br>
        Content: <br>
        <textarea name='content'></textarea><br>
        <input type="submit">
    </form>

@endsection
```

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

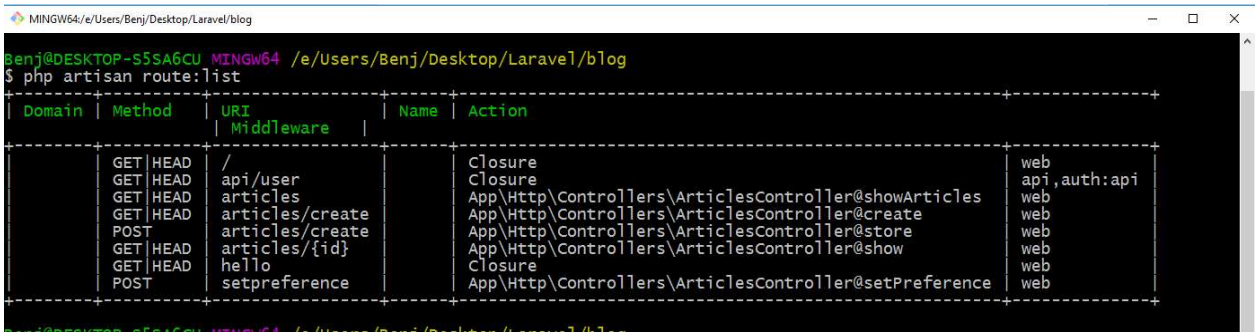
https://laravel.com/docs/5.3/authentication

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

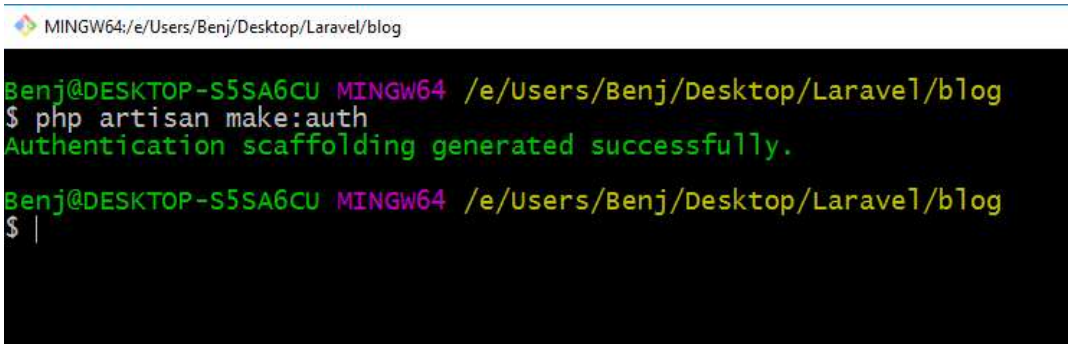
php artisan route:list



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:

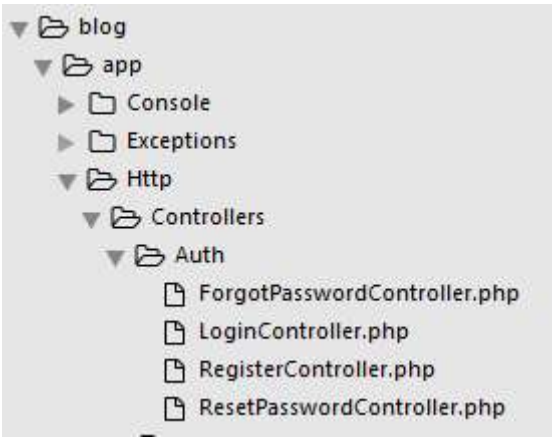
```php
Route::get('/', function () {
    return view('welcome');
});

// web.php
Route::get('/articles','ArticlesController@showArticles');
Route::get('/articles/create','ArticlesController@create' );
Route::get('/articles/{id}','ArticlesController@show' );
Route::post('/articles/create','ArticlesController@store' );
Route::post('/setpreference','ArticlesController@setPreference');


Route::get('/hello', function () {
    return "Hello World!";
});

Auth::routes();

Route::get('/home', 'HomeController@index');

```

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

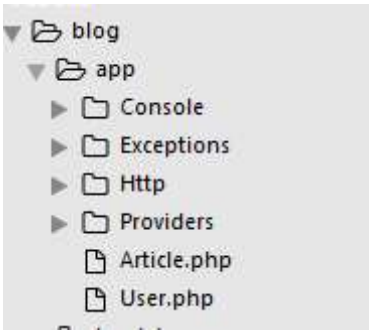
```
+----------+-----------------------+----------+-----------------------------------------------------------------+---------------+
| GET|HEAD | /                     |          | Closure                                                         | web           |
| GET|HEAD | api/user              |          | Closure                                                         | api,auth:api  |
| GET|HEAD | articles              |          | App\Http\Controllers\ArticlesController@showArticles            | web           |
| GET|HEAD | articles/create       |          | App\Http\Controllers\ArticlesController@create                  | web           |
| POST     | articles/create       |          | App\Http\Controllers\ArticlesController@store                   | web           |
| GET|HEAD | articles/{id}         |          | App\Http\Controllers\ArticlesController@show                    | web           |
| GET|HEAD | hello                 |          | Closure                                                         | web           |
| GET|HEAD | home                  |          | App\Http\Controllers\HomeController@index                       | web,auth      |
| GET|HEAD | login                 | login    | App\Http\Controllers\Auth\LoginController@showLoginForm         | web,guest     |
| POST     | login                 |          | App\Http\Controllers\Auth\LoginController@login                 | web,guest     |
| POST     | logout                | logout   | App\Http\Controllers\Auth\LoginController@logout                | web           |
| POST     | password/email        |          | App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail | web,guest |
| GET|HEAD | password/reset        |          | App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm | web,guest |
| POST     | password/reset        |          | App\Http\Controllers\Auth\ResetPasswordController@reset         | web,guest     |
| GET|HEAD | password/reset/{token} |         | App\Http\Controllers\Auth\ResetPasswordController@showResetForm | web,guest     |
| GET|HEAD | register              | register | App\Http\Controllers\Auth\RegisterController@showRegistrationForm | web,guest   |
| POST     | register              |          | App\Http\Controllers\Auth\RegisterController@register           | web,guest     |
| POST     | setpreference         |          | App\Http\Controllers\ArticlesController@setPreference           | web           |
+----------+-----------------------+----------+-----------------------------------------------------------------+---------------+
```

   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

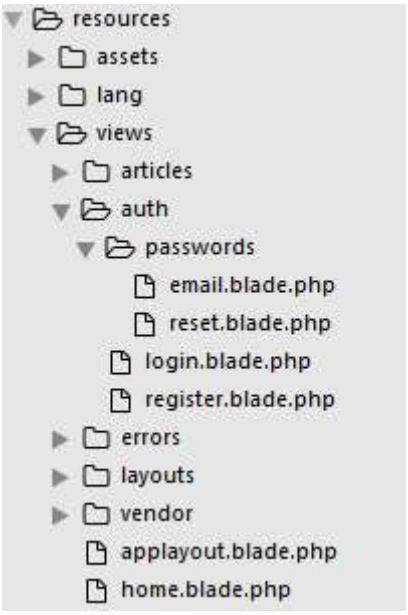
2. For **controllers**: Under the Auth directory, there are



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

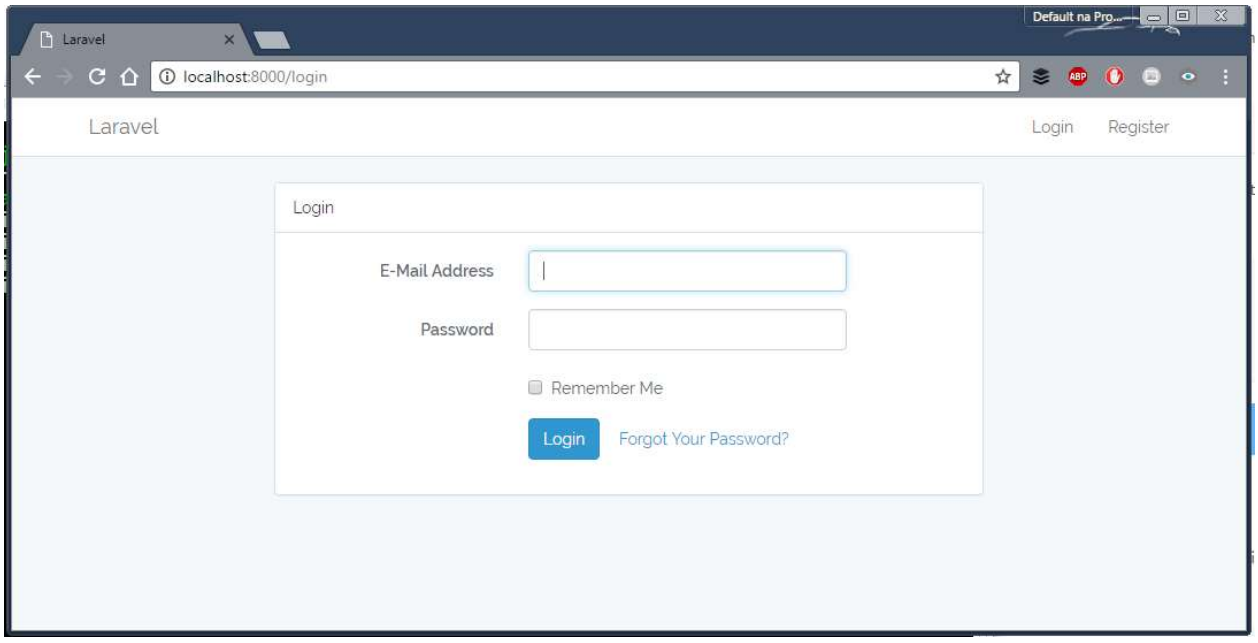


4. For **views**, they added additional views under auth directory,
    - email.blade.php
    - reset.blade.php
    - login.blade.php
    - register.blade.php
    - applayout.blade.php
    - home.blade.php



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

```php
class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }

    /**
     * Show the application dashboard.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $current_user = Auth::user();
        return view('home',compact('current_user'));
    }
}
```

If this error appears:

Add the following at the beginning of the controller

```php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;

class HomeController extends Controller
{
```

Then we use the **`$current_user`** variable as an eloquent object

In **home.blade.php**:

```
<div class="col-md-8 col-md-offset-2">
    <div class="panel panel-default">
        <div class="panel-heading">Dashboard</div>

        <div class="panel-body">
            You are logged in!
            <ul>
                <li>Username:   {{ $current_user->username}}    </li>
                <li>Name:       {{ $current_user->name}}        </li>
                <li>Email:      {{ $current_user->email}}       </li>
            </ul>

        </div>
    </div>
</div>
```

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

```
class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }
```

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

```
MINGW64:/e/Users/Benj/Desktop/Laravel/blog

Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$ php artisan make:migration add_username_to_users --table=users
Created Migration: 2017_01_05_175315_add_username_to_users

Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$
```

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

```php
class AddUsernameToUsers extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('username')->unique();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('username');
        });
    }
}
```

And run the migration

```
MINGW64:/e/Users/Benj/Desktop/Laravel/blog

Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$ php artisan migrate
Migrated: 2017_01_05_175315_add_username_to_users

Benj@DESKTOP-S5SA6CU MINGW64 /e/Users/Benj/Desktop/Laravel/blog
$
```

We'll see the changes in phpmyadmin

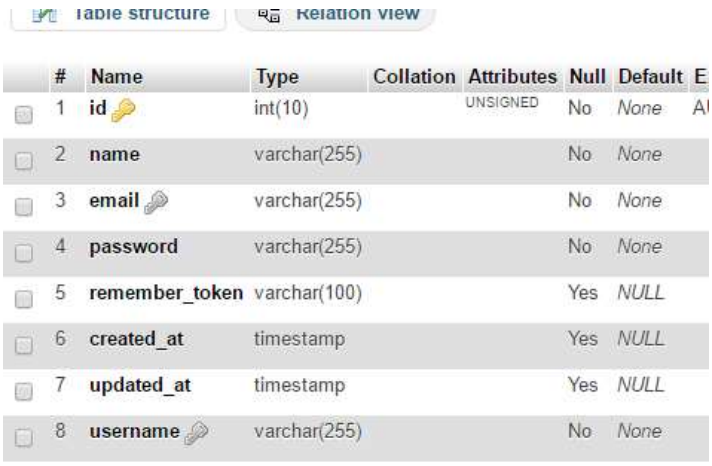


Table structure | Relation view

#	Name	Type	Collation	Attributes	Null	Default	E
1	id	int(10)		UNSIGNED	No	None	A
2	name	varchar(255)			No	None	
3	email	varchar(255)			No	None	
4	password	varchar(255)			No	None	
5	remember_token	varchar(100)			Yes	NULL	
6	created_at	timestamp			Yes	NULL	
7	updated_at	timestamp			Yes	NULL	
8	username	varchar(255)			No	None	

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

```
class User extends Authenticatable
{
    use Notifiable;

    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password','username'
    ];

    /**
     * The attributes that should be hidden for arrays.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];
}
```

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

We add validation rules for our username field

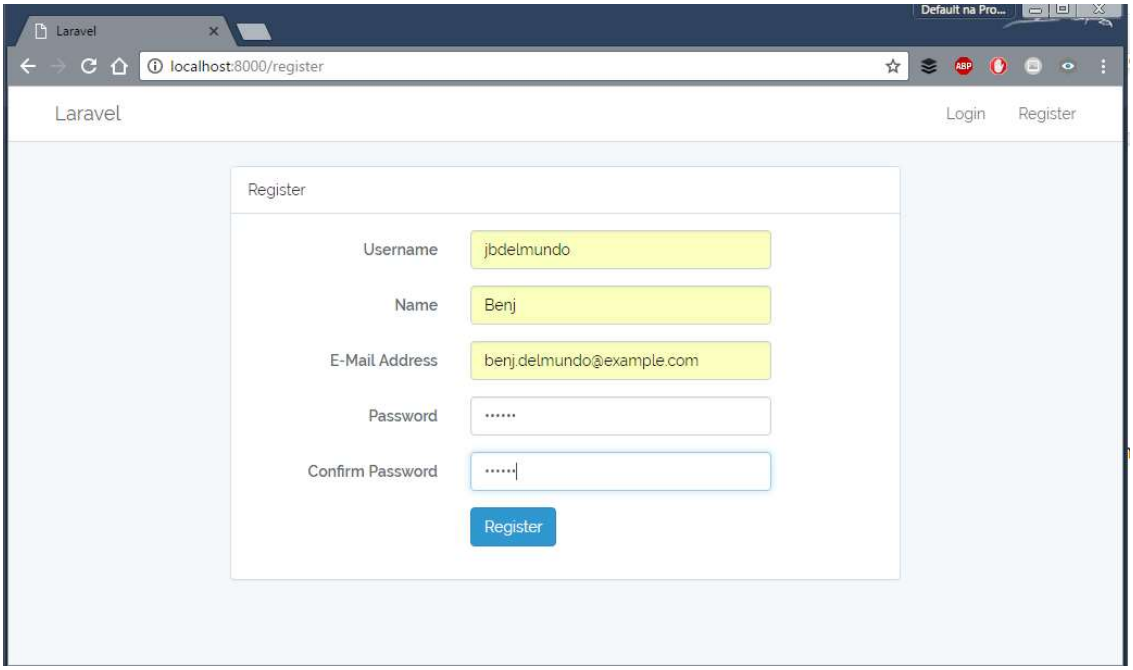
```php
protected function validator(array $data)
{
    return Validator::make($data, [
        'name' => 'required|max:255',
        'email' => 'required|email|max:255|unique:users',
        'password' => 'required|min:6|confirmed',
        'username' => 'required',
    ]);
}
```

And add the username upon creating a user

```php
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
        'username' => $data['username'],
    ]);
}
```

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

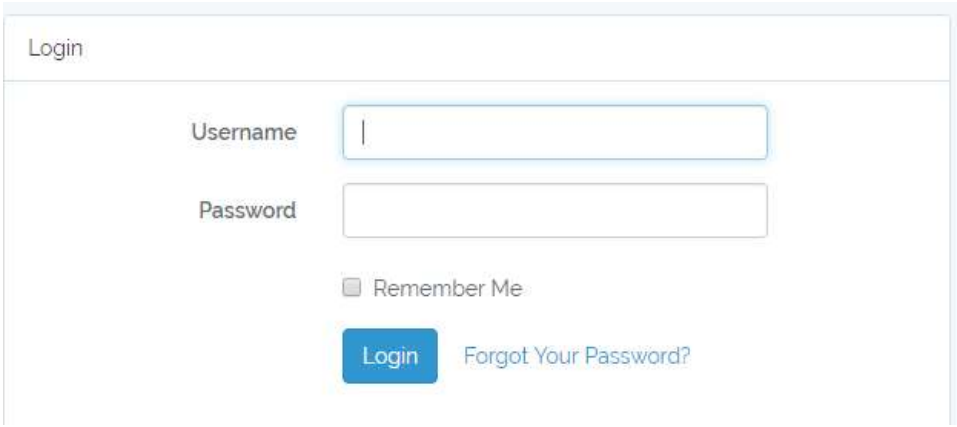
```blade
<form class="form-horizontal" role="form" method="POST" action="{{ url('/register') }}">
    {{ csrf_field() }}

    <div class="form-group{{ $errors->has('username') ? ' has-error' : '' }}">
        <label for="username" class="col-md-4 control-label">Username</label>

        <div class="col-md-6">
            <input id="username" type="text" class="form-control" name="username" value="{{
            >

            @if ($errors->has('username'))
                <span class="help-block">
                    <strong>{{ $errors->first('username') }}</strong>
                </span>
            @endif
        </div>
    </div>


    <div class="form-group{{ $errors->has('name') ? ' has-error' : '' }}">
        <label for="name" class="col-md-4 control-label">Name</label>

        <div class="col-md-6">
            <input id="name" type="text" class="form-control" name="name" value="{{ old('nar

            @if ($errors->has('name'))
                <span class="help-block">
                    <strong>{{ $errors->first('name') }}</strong>
                </span>
            @endif
        </div>
    </div>
```



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

```
<div class="form-group{{ $errors->has('username') ? ' has-error' : '' }}">
    <label for="username" class="col-md-4 control-label">Username</label>

    <div class="col-md-6">
        <input id="username" type="text" class="form-control" name="username" value="{{ old('username') }}" required
         autofocus>

        @if ($errors->has('username'))
            <span class="help-block">
                <strong>{{ $errors->first('username') }}</strong>
            </span>
        @endif
    </div>
</div>
```



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.

```
    public function username()
    {
        return 'username';
    }
}
```

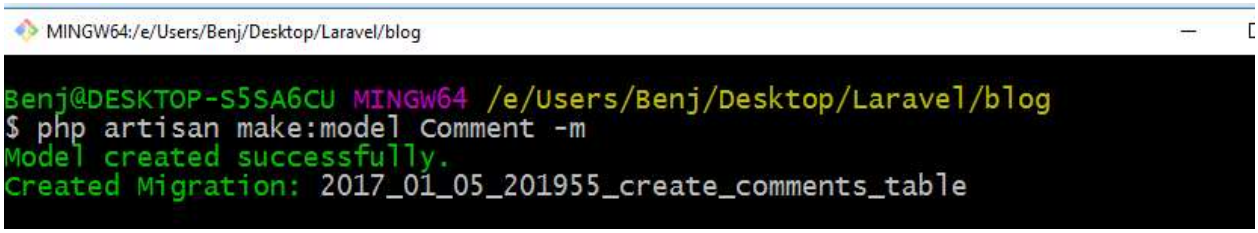
Lesson 16: Eloquent Relationships

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



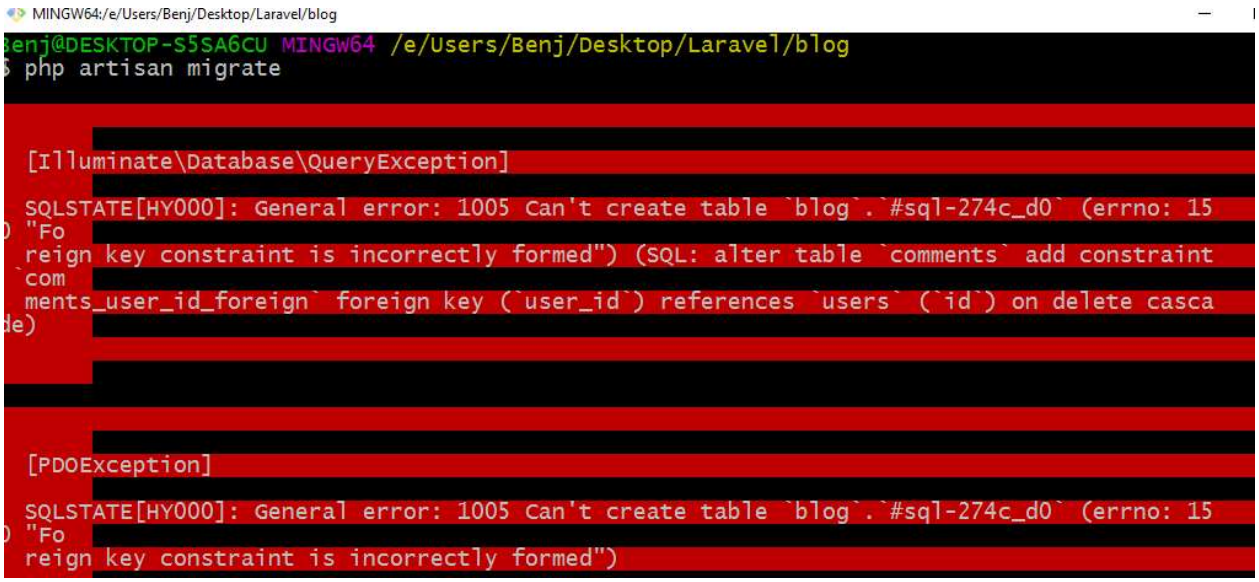
And on the migration file, we add

- user_id (INTEGER), it is important that we make it **unsigned**!
- article_id (INTEGER), it is important that we make it **unsigned**!
- description (TEXT)

```php
14      public function up()
15      {
16          Schema::create('comments', function (Blueprint $table) {
17              $table->increments('id');
18              $table->integer('user_id')->unsigned();
19              $table->integer('article_id')->unsigned();
20              $table->text('description');
21              $table->timestamps();
22
23              $table->foreign('user_id')
24                ->references('id')->on('users')
25                ->onDelete('cascade');
26              $table->foreign('article_id')
27                ->references('id')->on('articles')
28                ->onDelete('cascade');
29          });
30      }
```

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:



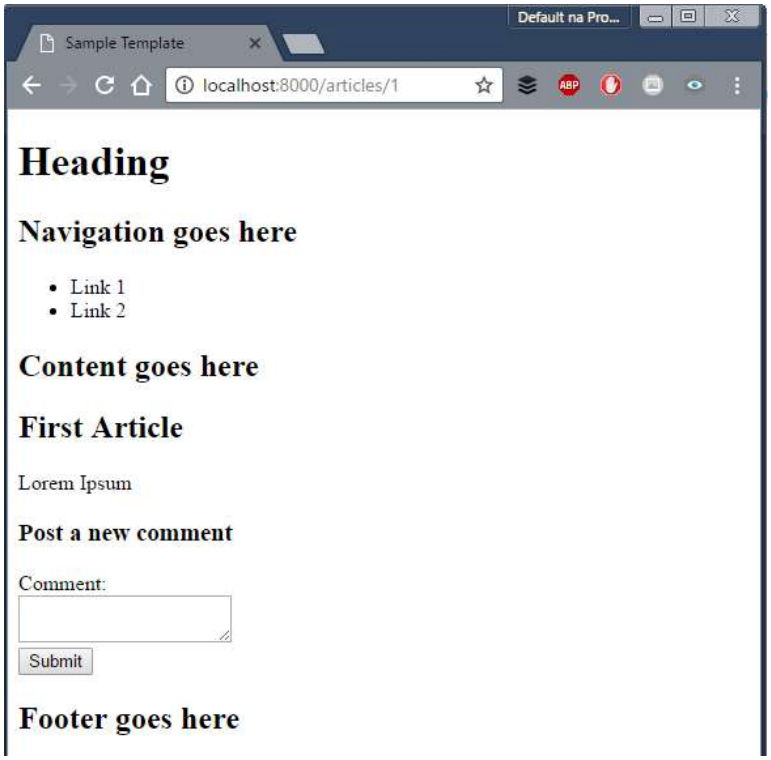
If we encounter **"Foreign key constraint incorrectly formed"** error in the future, check the following

- 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**:

```
<!-- articles_show_single_item.blade.php -->
@extends('applayout')

@section('main_content')

    <h2>{{ $article->title  }}</h2>
    <p>{{ $article->content }}</p>

    <h3>Post a new comment</h3>

    <form action = "" method="POST">
        {{ csrf_field() }}
        Comment: <br>
        <textarea name='description'></textarea><br>
        <input type="submit">
    </form>

@endsection
```



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

So we create a new route:

```
18  // web.php
19  Route::get('/articles','ArticlesController@showArticles');
20  Route::get('/articles/create','ArticlesController@create' );
21  Route::get('/articles/{id}','ArticlesController@show' );
22  Route::post('/articles/create','ArticlesController@store' );
23  Route::post('/setpreference','ArticlesController@setPreference');
24
25  Route::post('/articles/{id}','ArticlesController@postComment' );
```

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

```
56      function postComment(Request $request,$id){
57          $comment = $request->description;
58          $user_id = Auth::user()->id;
59          $article_id = $id;
60
61          return "Comment to article " . $article_id .
62                  "is " . $comment . " by user " . $user_id;
63      }
```

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.

```
52    function postComment(Request $request,$id){
53        $comment = $request->description;
54        $user_id = Auth::user()->id;
55        $article_id = $id;
56
57        $comment_obj = new \App\Comment;
58        $comment_obj->user_id = $user_id;
59        $comment_obj->article_id = $article_id;
60        $comment_obj->description = $comment;
61        $comment_obj->save();
62
63        return redirect("articles/$id");
64    }
```

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**

```php
class Article extends Model
{
    //
    function comments(){
        return $this->hasMany('App\Comment', 'article_id', 'id');
    }
}
```

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.

```
 7  class ArticlesController extends Controller
 8  {
 9      function showArticles(Request $request){
10          $all_articles = \App\Article::all();
11          $preference = $request->session()->get('preference', 'default_preference
12          return view('articles.articles_list', compact('all_articles', 'preferenc
13      }
14
15      function show($id){
16          $article = \App\Article::find($id);
17          $result = $article->comments();
18          dd($result);
19
20          return view('articles.articles_show_single_item', compact('article'));
21      }
```

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

```
function show($id){
    $article = \App\Article::find($id);
    $result = $article->comments();
    $related_comments = $article->comments;
    dd($related_comments);

    return view('articles.articles_show_single_item', compact('article'));
}
```

Or use the get() on the relationship

```
15▼     function show($id){
16          $article = \App\Article::find($id);
17          $result = $article->comments();
18          // $related_comments = $article->comments;
19          $related_comments = $article->comments()->get();
20          dd($related_comments);
21
22          return view('articles.articles_show_single_item', compact('article'));
23      }
```

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**

```
1  <!-- articles_show_single_item.blade.php -->
2  @extends('applayout')
3
4  @section('main_content')
5
6      <h2>{{ $article->title  }}</h2>
7      <p>{{ $article->content }}</p>
8
9      <h3>Post a new comment</h3>
10
11     <form action = "" method="POST">
12         {{ csrf_field() }}
13         Comment: <br>
14         <textarea name='description'></textarea><br>
15         <input type="submit">
16     </form>
17
18     <h3>Comments</h3>
19     <ul>
20         @foreach($article->comments as $comment)
21             <li>
22                 {{ $comment->description}}
23             </li>
24         @endforeach
25     </ul>
26
27 @endsection
```

Part 2: One-to-Many Relationship

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

```php
class Comment extends Model
{
    function article(){
        return $this->belongsTo('App\Article', 'article_id', 'id');
    }
}
```

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