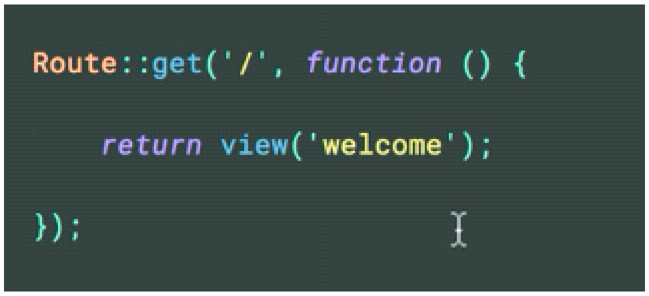
Routes and passing values to views

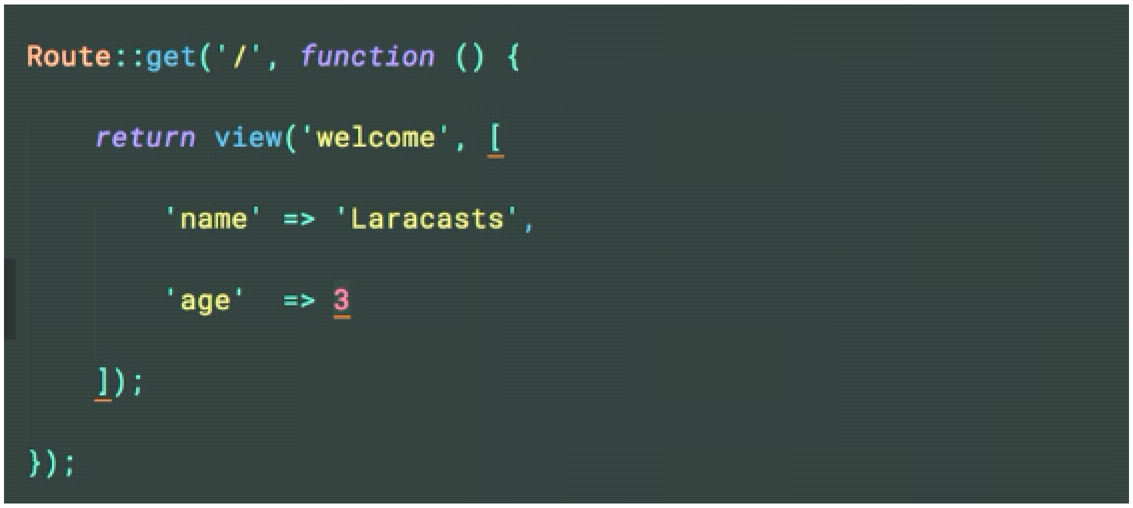
Tuesday, March 28, 2017 3:54 PM

- We can return views without any variables.



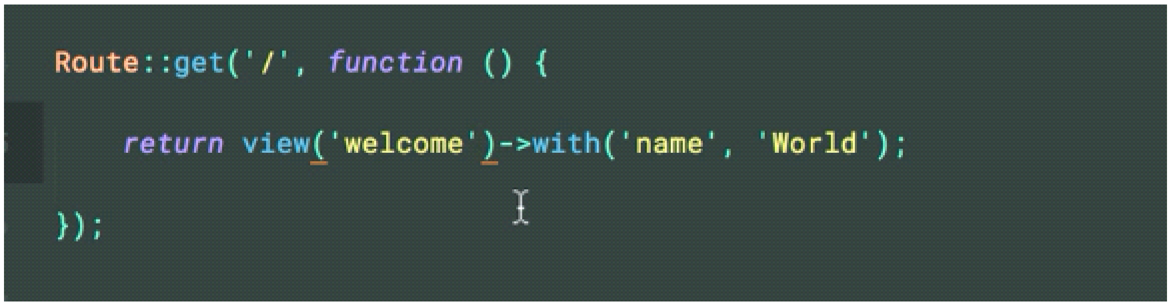
```
Route::get('/', function () {

    return view('welcome');

});
```

- We can Views with associative arrays ,Now we can use index of arrays as variable name in view;



```
Route::get('/', function () {

    return view('welcome', [

        'name' => 'Laracasts',

        'age'  => 3

    ]);

});
```

- We can also pass values using **with(key_name,value_name)** function.
  'key_name', will be used as a name for variable inside view.

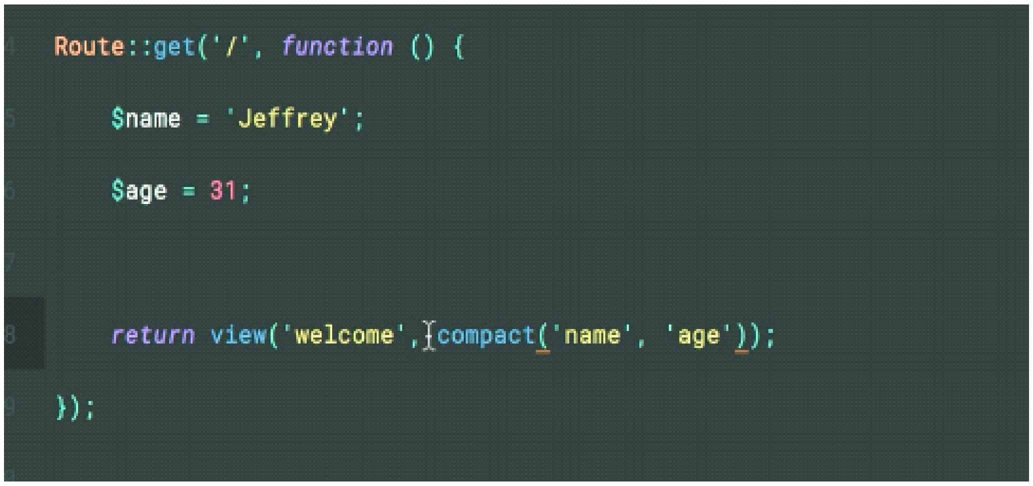


```
Route::get('/', function () {

    return view('welcome')->with('name', 'World');

});
```

COMPACT METHOD

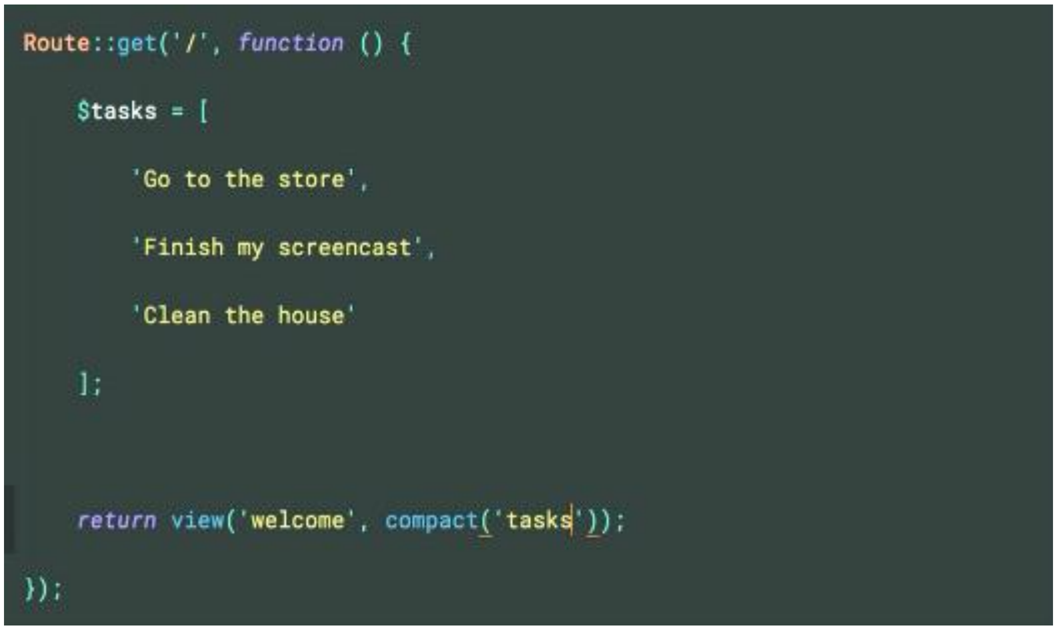
- We can pass all variable using **compact()** method

1- Passing simple variable, now the name of these function will act as name in views.



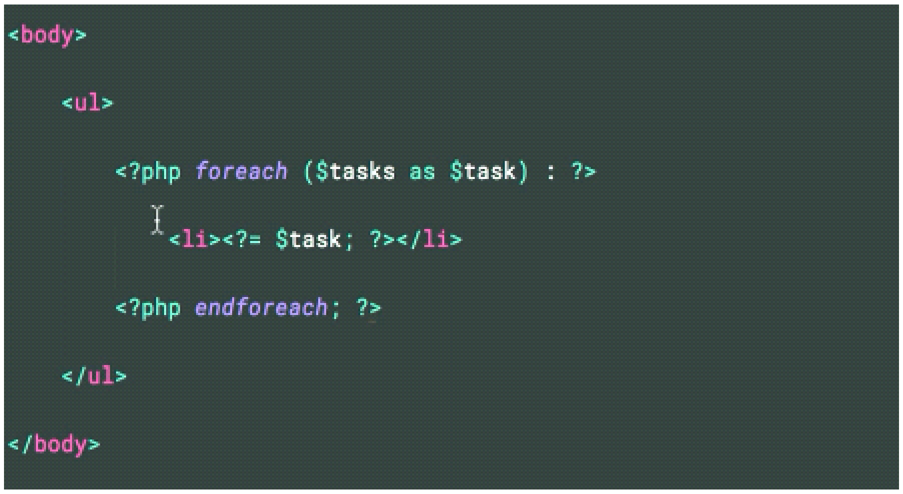
```
Route::get('/', function () {

    $name = 'Jeffrey';

    $age = 31;


    return view('welcome', compact('name', 'age'));

});
```

2- Passing simple array, now the we can access the whole array using array name in view.



```
Route::get('/', function () {

    $tasks = [

        'Go to the store',

        'Finish my screencast',

        'Clean the house'

    ];


    return view('welcome', compact('tasks'));

});
```

- Inside views we can iterate using as this.

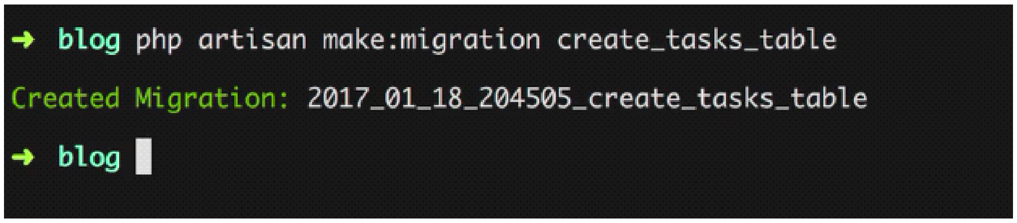


```
<body>

    <ul>

        <?php foreach ($tasks as $task) : ?>

            <li><?= $task; ?></li>

        <?php endforeach; ?>

    </ul>

</body>
```

Query builder

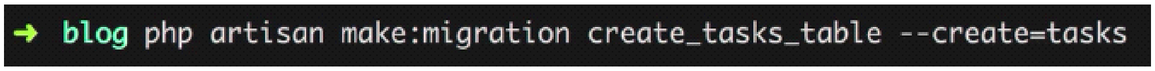
Tuesday, March 28, 2017 4:17 PM

Creating Migrations



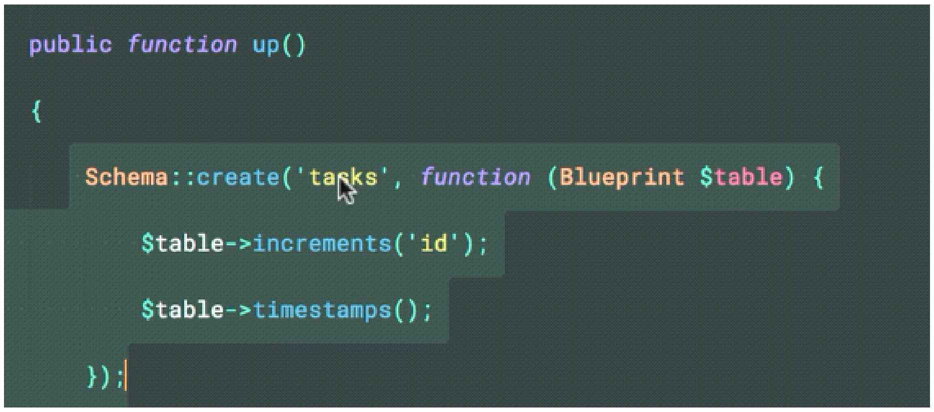
```
➜ blog php artisan make:migration create_tasks_table
Created Migration: 2017_01_18_204505_create_tasks_table
➜ blog
```

- We can also tell artisan , that in this migration we are going to create a table 'task', so it will create necessary template for that table.



```
➜ blog php artisan make:migration create_tasks_table --create=tasks
```

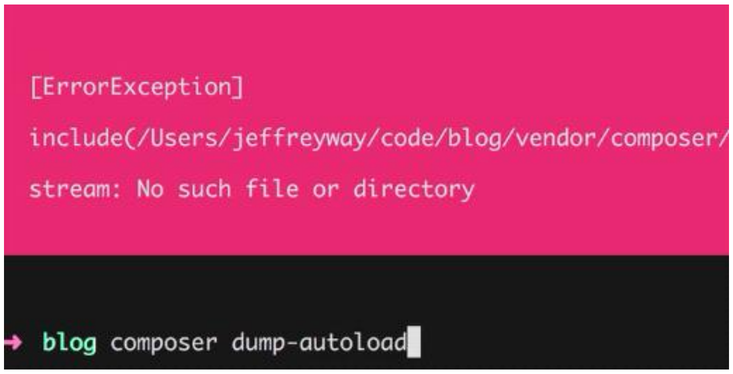
- Template provided inside up() function



```
public function up()
{
    Schema::create('tasks', function (Blueprint $table) {
        $table->increments('id');
        $table->timestamps();
    });
```

- Some times we might get this error if we delete some files and Composer do not know about them, So when it tries to find these files, it could not and gives us error.

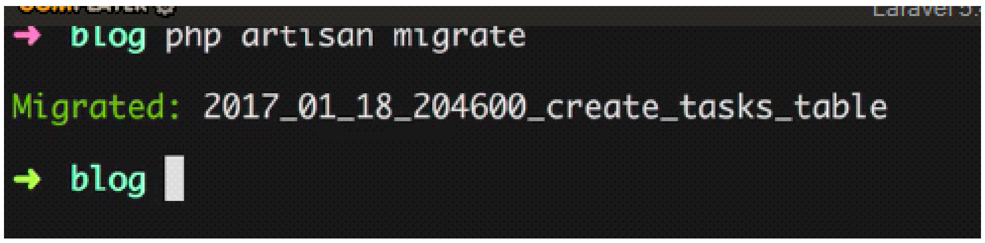
- So we have to tell composer to autoload your directory.
  so we have to use following command



```
[ErrorException]
include(/Users/jeffreyway/code/blog/vendor/composer/
stream: No such file or directory

➜ blog composer dump-autoload
```

Running Migrations

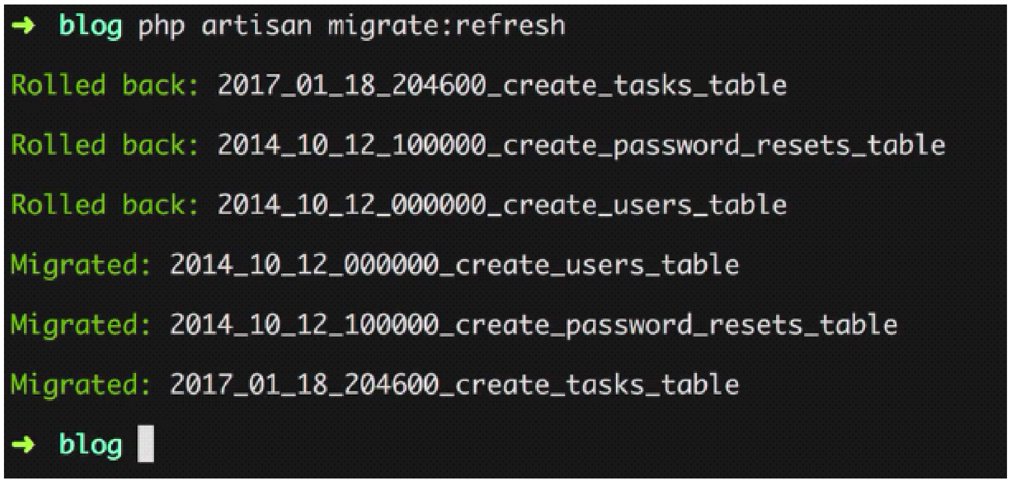
- Now we can tell php to run all migrations we have created.



```
➜  blog php artisan migrate

Migrated: 2017_01_18_204600_create_tasks_table

➜  blog
```

Re-running Migrations

- Sometimes, we do some mistake so we just tell to refresh, this will rollback all migration(run down() function of all migrations) and then migrate(run up() function of all migrations)



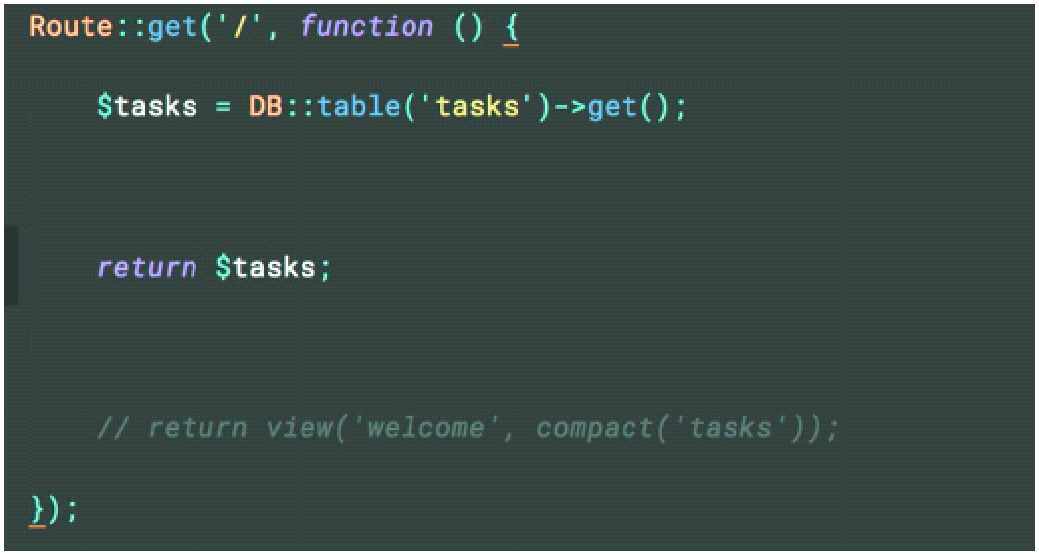
```
➜  blog php artisan migrate:refresh

Rolled back: 2017_01_18_204600_create_tasks_table

Rolled back: 2014_10_12_100000_create_password_resets_table

Rolled back: 2014_10_12_000000_create_users_table

Migrated: 2014_10_12_000000_create_users_table

Migrated: 2014_10_12_100000_create_password_resets_table

Migrated: 2017_01_18_204600_create_tasks_table

➜  blog
```

Running Queries

- There are two methods of Doing Queries
  - Method 1 - using DB class Query Builder
  - Method 2 - using Laravel Eloquent Model

Using DB class

- We can used DB class to build our queries.



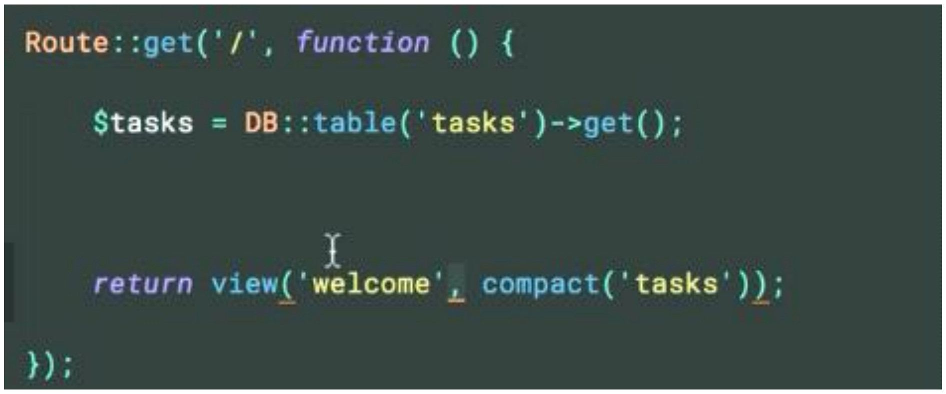
```php
Route::get('/', function () {

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


    return $tasks;


    // return view('welcome', compact('tasks'));

});
```

NOTE- if we every return database result to browser(like above) , laravel automatically convert it
into JSON formate, so we get following result.

```
[{"id":1,"body":"Go to the store","created_at":"2017-01-18 15:48:56","updated_at":
18 15:49:02","updated_at":"2017-01-18 15:49:02"}]
```

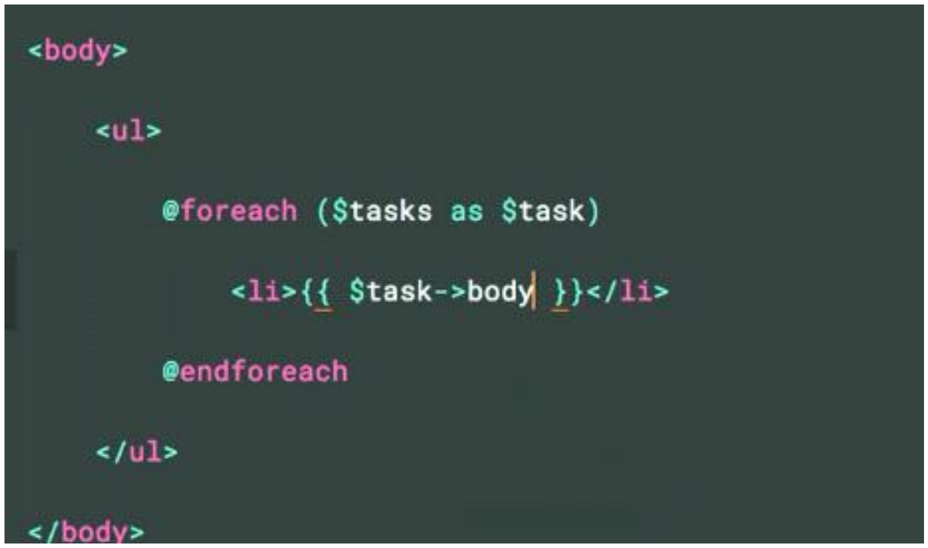
- When we pass 'query result' to a view, it goes in the form of objects(by default), so we have to access using object formatting(using arrows)

- IN ROUTE



```php
Route::get('/', function () {

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


    return view('welcome', compact('tasks'));

});
```

- IN VIEW



```
<body>

    <ul>

        @foreach ($tasks as $task)

            <li>{{ $task->body }}</li>

        @endforeach

    </ul>

</body>
```

- All kind of functionalities like
    - Where()
    - Latest()
    - Orderby()
    - First() - for selecting first record only

      Etc are present and we can see the docs.

Eloquent Model

Tuesday, March 28, 2017 6:18 PM

Introduction

The Eloquent ORM included with Laravel provides a beautiful, simple ActiveRecord implementation for working with your database. Each database table has a corresponding "Model" which is used to interact with that table. Models allow you to query for data in your tables, as well as insert new records into the table.

Defining Models

The easiest way to create a model instance is using the `make:model` Artisan command:

```
php artisan make:model User
```

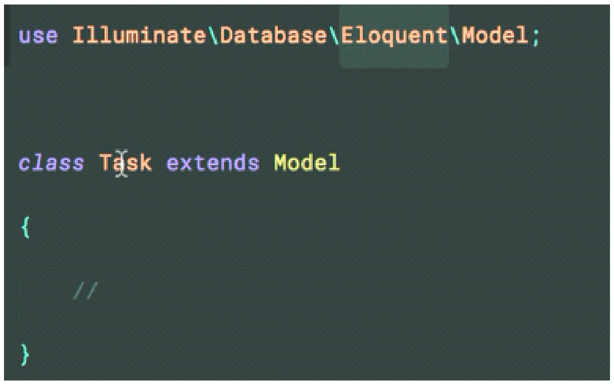
If you would like to generate a database migration when you generate the model, you may use the `--migration` or `-m` option:

```
php artisan make:model User --migration

php artisan make:model User -m
```

Working with database

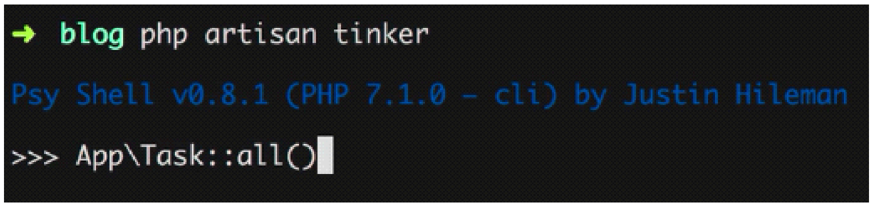
- Once a Eloquent model and its migration is made.
- We can use object of that model to do database queries, instead of using DB class.

Example - here we have a Eloquent Model 'Task' , now we can do various queries on it.(in tinker for demo)



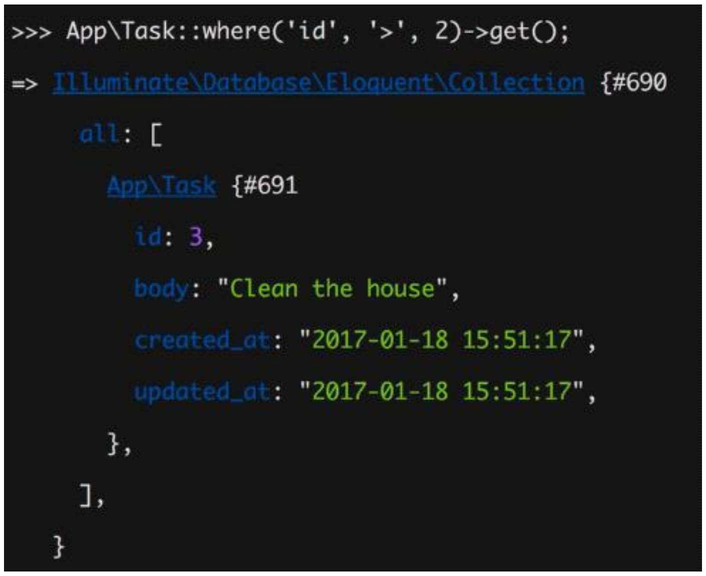
```
use Illuminate\Database\Eloquent\Model;


class Task extends Model

{

    //

}
```

1- For getting all Tasks from database



```
➜ blog php artisan tinker
Psy Shell v0.8.1 (PHP 7.1.0 — cli) by Justin Hileman
>>> App\Task::all()
```

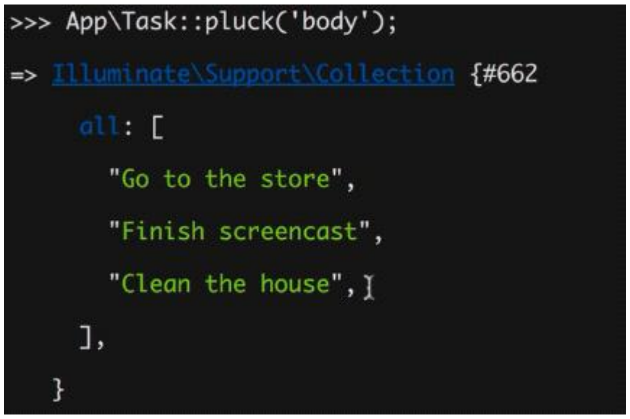
2- For getting all Tasks where id>2



```
>>> App\Task::where('id', '>', 2)->get();
=> Illuminate\Database\Eloquent\Collection {#690
     all: [
       App\Task {#691
         id: 3,
         body: "Clean the house",
         created_at: "2017-01-18 15:51:17",
         updated_at: "2017-01-18 15:51:17",
       },
     ],
   }
```

3- Needing Just a Columns from all records

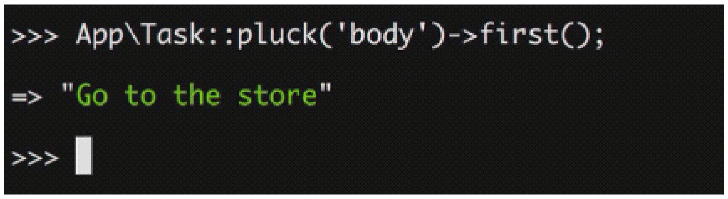
- If we wanted to get just one column from all records we can pluck() them.
- It will come in form of an



```
>>> App\Task::pluck('body');
=> Illuminate\Support\Collection {#662
     all: [
       "Go to the store",
       "Finish screencast",
       "Clean the house",
     ],
   }
```

4- Getting first value from records or values

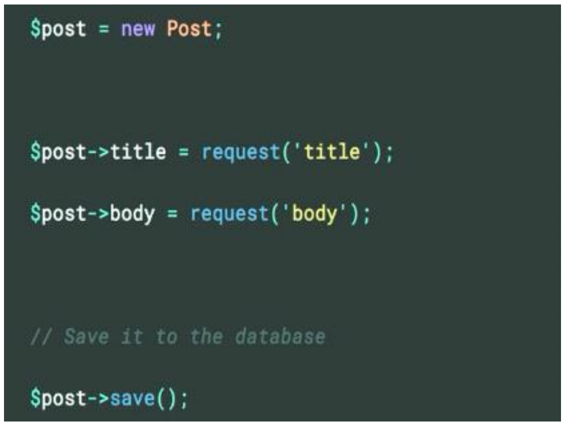
- We can use first() method to do this.



```
>>> App\Task::pluck('body')->first();
=> "Go to the store"
>>> 
```

5- Saving data

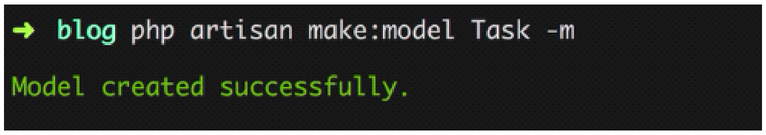
- We can save data in database using save() method from a object of model.



```
$post = new Post;


$post->title = request('title');

$post->body = request('body');


// Save it to the database

$post->save();
```

Creating Migrations along with Model

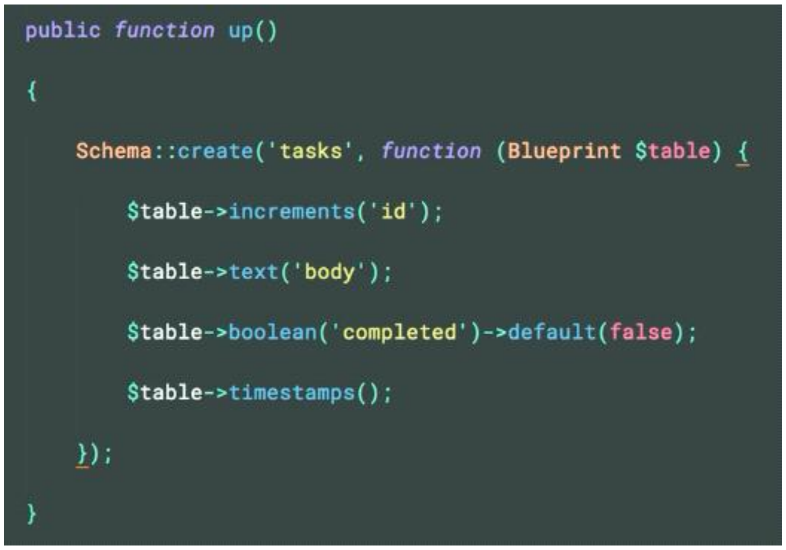
- Most of the time we wanted to create our migration when we create model,
- We can do this by adding **-m** or **--migration** (we can use -c or --controller for creating controller)



```
➜  blog php artisan make:model Task -m
Model created successfully.
```

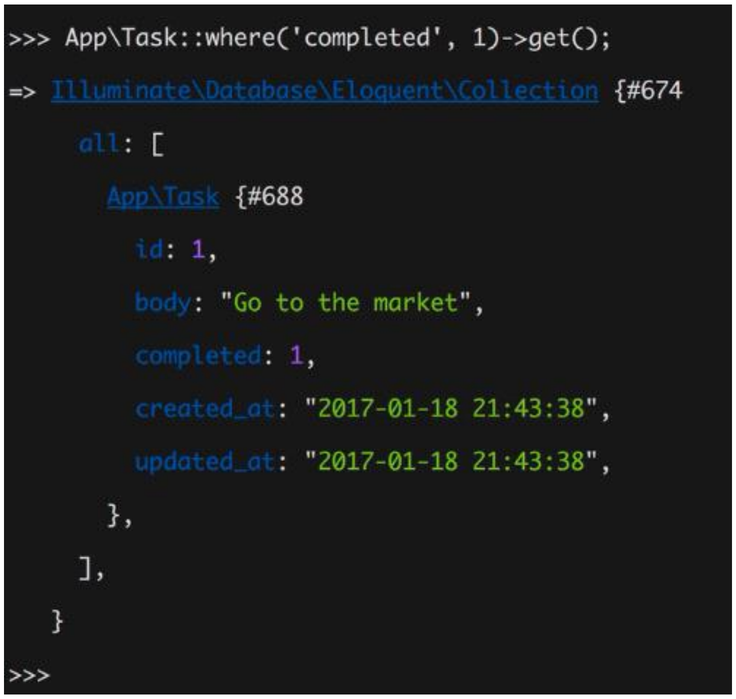
Adding Additional Behaviour to Eloquent Model

- We might wanted to add more function according to our need.
- For example in for 'Task' Model we wanted to get the tasks which are incomplete, for this we have added a boolean 'completed' to Task Migrations.



```
public function up()
{
    Schema::create('tasks', function (Blueprint $table) {
        $table->increments('id');
        $table->text('body');
        $table->boolean('completed')->default(false);
        $table->timestamps();
    });
}
```

- Query to get completed 'Task' (similary we can make for incomplete)

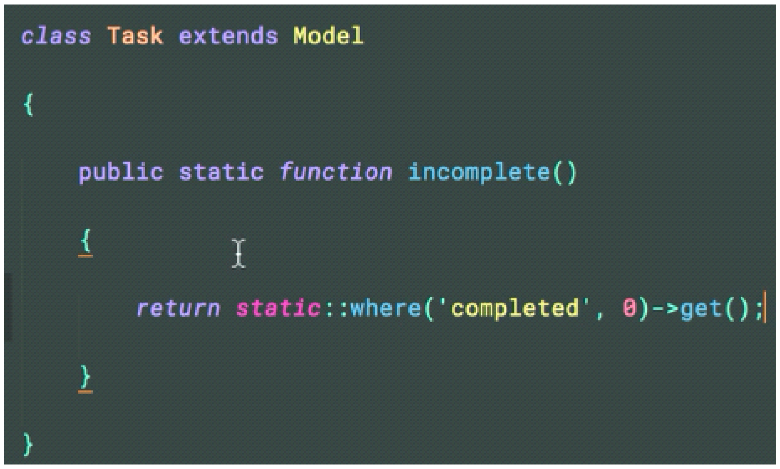

```
>>> App\Task::where('completed', 1)->get();
=> Illuminate\Database\Eloquent\Collection {#674
     all: [
       App\Task {#688
         id: 1,
         body: "Go to the market",
         completed: 1,
         created_at: "2017-01-18 21:43:38",
         updated_at: "2017-01-18 21:43:38",
       },
     ],
   }
>>>
```

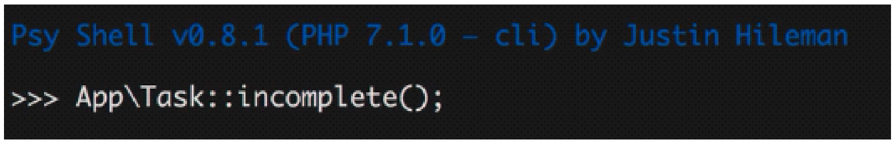
Creating incomplete() function

- There are two ways of defining function
  1- Create static function and return result.
  2- Create function using Query Scopes

METHOD 1 - creating a static function which return records

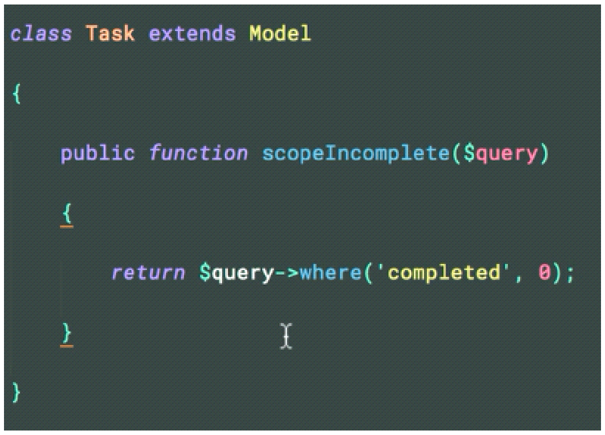
- Now we wanted to create a function for this in 'Task' model, incomplete( )
  Function will be static as we want to call by Class name.
- Problem - this method return records, so can't be used for chaining.


```
class Task extends Model
{
    public static function incomplete()
    {
        return static::where('completed', 0)->get();
    }
}
```

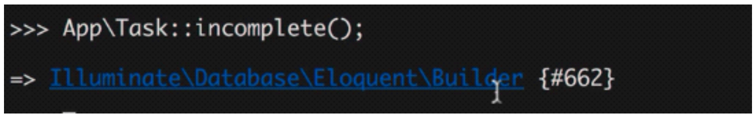

```
Psy Shell v0.8.1 (PHP 7.1.0 — cli) by Justin Hileman
>>> App\Task::incomplete();
```

METHOD 2 - creating a function which return object reference

- This function will be regular public function, having function name prefixed with **'scope'**
  will not be used when calling(just for laravel)

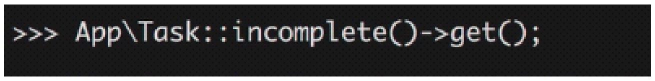


```
class Task extends Model

{

    public function scopeIncomplete($query)

    {

        return $query->where('completed', 0);

    }

}
```



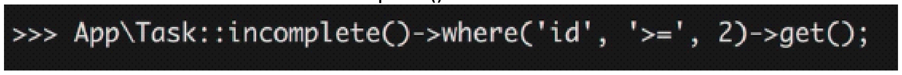
```
>>> App\Task::incomplete();

=> Illuminate\Database\Eloquent\Builder {#662}
```

- This is returning query object, so for getting records we have to use get() or select() or first() etc.



```
>>> App\Task::incomplete()->get();
```

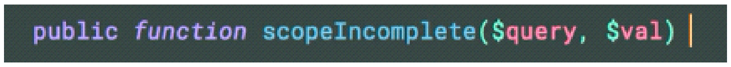
- Now we can chain the method incomplete() as follows



```
>>> App\Task::incomplete()->where('id', '>=', 2)->get();
```

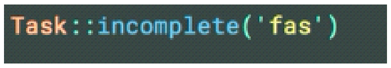
NOTE -

- we a new argument **$val** could be passed along with **$query**($query is used for chaining) ,



```
public function scopeIncomplete($query, $val)
```

Using which we can pass to function while chaining, like



```
Task::incomplete('fas')
```

Controllers

Tuesday, March 28, 2017 9:21 PM

Introduction

Instead of defining all of your request handling logic as Closures in route files, you may wish to organize this behavior using Controller classes. Controllers can group related request handling logic into a single class. Controllers are stored in the `app/Http/Controllers` directory.

Defining Controllers

Below is an example of a basic controller class. Note that the controller extends the base controller class included with Laravel. The base class provides a few convenience methods such as the `middleware` method, which may be used to attach middleware to controller actions:

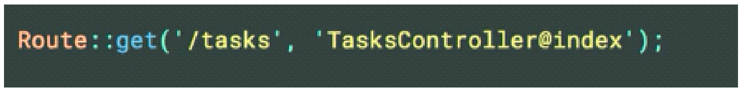
```php
<?php

namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }
}
```

You can define a route to this controller action like so:

```php
Route::get('user/{id}', 'UserController@show');
```

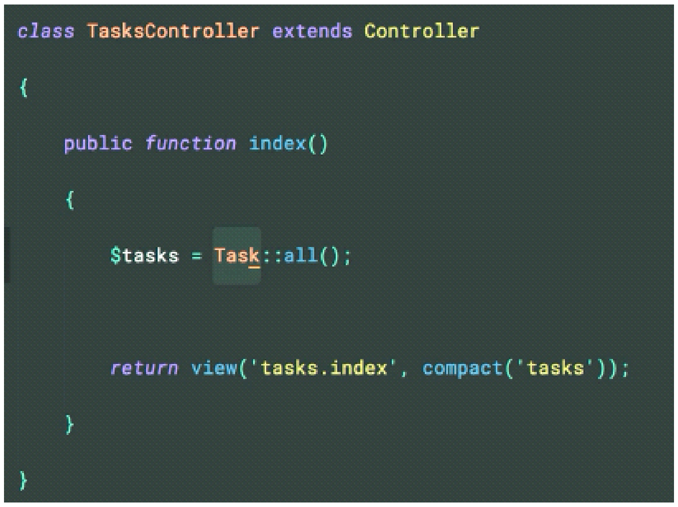
Example-

- Inside route/web.php



```php
Route::get('/tasks', 'TasksController@index');
```

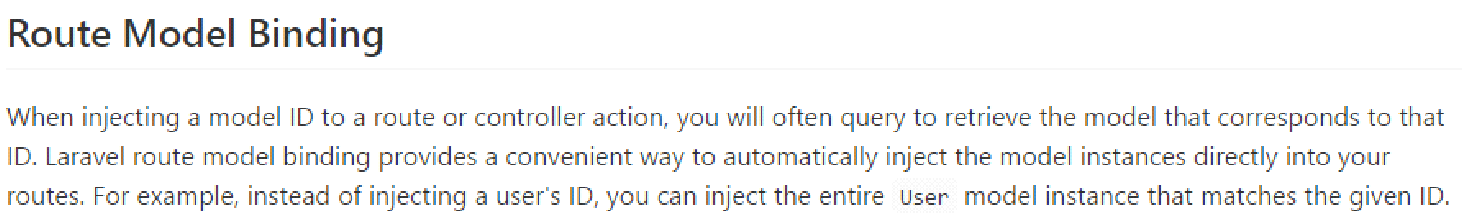
- Inside TasksController



```
class TasksController extends Controller

{

    public function index()

    {

        $tasks = Task::all();


        return view('tasks.index', compact('tasks'));

    }

}
```

Route Model Binding

Tuesday, March 28, 2017 9:36 PM



Route Model Binding

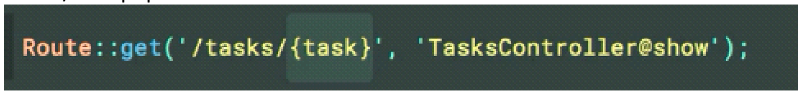
When injecting a model ID to a route or controller action, you will often query to retrieve the model that corresponds to that ID. Laravel route model binding provides a convenient way to automatically inject the model instances directly into your routes. For example, instead of injecting a user's ID,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

- There are two types of binding
  - Implicit
  - Explicit (? Task)

Implicit Binding

- Whenever a wildcard is used in route, like following

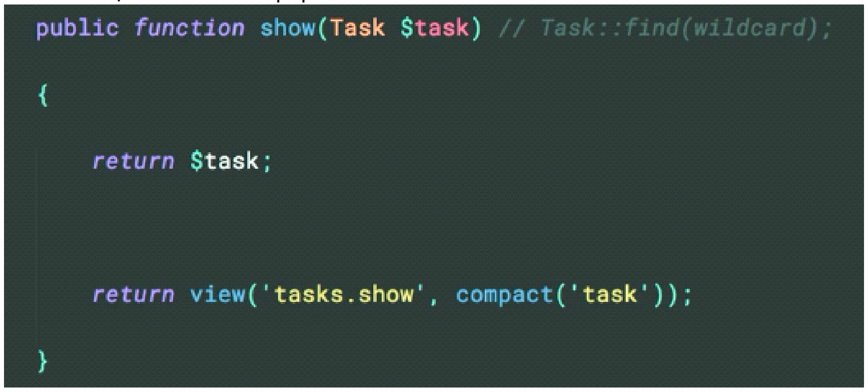
Inside route/web.php



```php
Route::get('/tasks/{task}', 'TasksController@show');
```

- Laravel is takes that wildcard, and fetch Model object for that Controller automatically, for that function(if argument is Model type)
  NAME OF WILD CARD AND OBJECT NAME MUST BE SAME , (here 'task')
- By default wild card will be taken as primary key.(we can change it , will study later)

Inside Controllers/TaskController.php

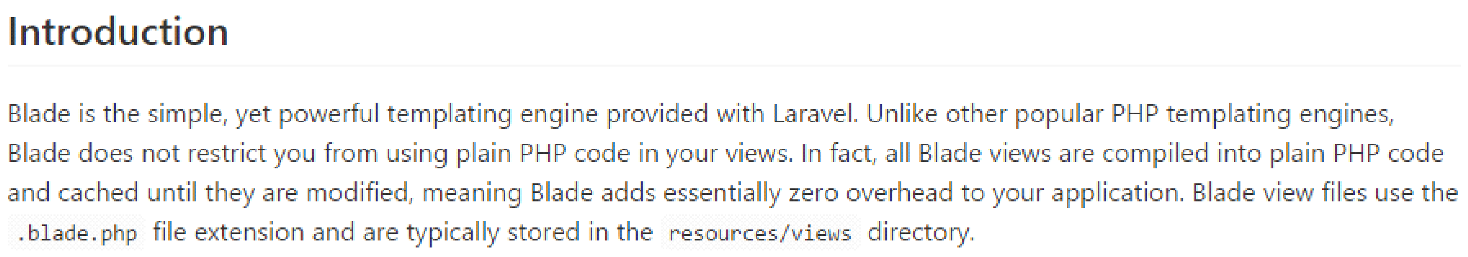


```php
public function show(Task $task) // Task::find(wildcard);

{

    return $task;


    return view('tasks.show', compact('task'));

}
```

- Working - here laravel takes 'task' wild card and in function it see we want 'Task' Model, so fetch the 'Task' Model object using the wild card and put in $task.

Blade Templating

Tuesday, March 28, 2017 9:49 PM



Introduction

Blade is the simple, yet powerful templating engine provided with Laravel. Unlike other popular PHP templating engines, Blade does not restrict you from using plain PHP code in your views. In fact, all Blade views are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade view files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

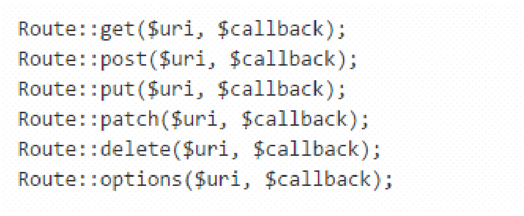
- There are many thing we can do with blade, all php controlls can be accesed with <?php ?> tag
  using @ symbol
  <?php if() ?> can be used as @if() and <? endif; ?> can be used as @endif, similary for all others.

  See Documentation for more

Form Request Data and CSRF

Tuesday, March 28, 2017 10:43 PM

- We can use any of the request in the routing, these are following



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

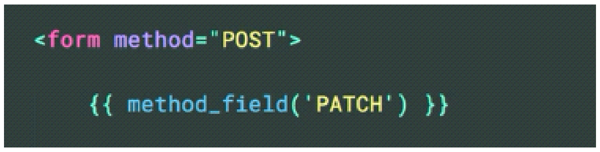
RESTFUL conventions

- restful conventions (**Post** is model)
  1- GET /posts - display all posts
  2- GET /posts/{id} - display post with given id(could be primary key or anyother)

  3- GET /posts/create - create a new post, this will have form
  4- POST /posts - form will be submitted by this.

  5- GET /posts/{id}/edit - shows an edit form (because id is needed for EDITING and /posts/{id} has already been used)
  6- PATCH /posts/{id} - edit form will be submitted to this(id needed but as different type ,no need to give 'edit' here).
  7- DELETE /posts/{id} - delete a post

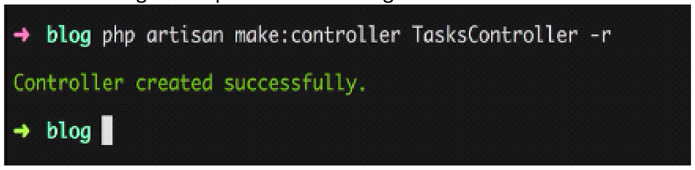
- NOTE- most of the older browser understand only GET and POST request only,
  so if we wanted to give PATCH or DELETE request, we can name method as POST and echo
  method_field() helper method and pass PATCH or DELETE to it.(it will make a hidden input)



```
<form method="POST">

    {{ method_field('PATCH') }}
```

Creating a Resourceful controllers

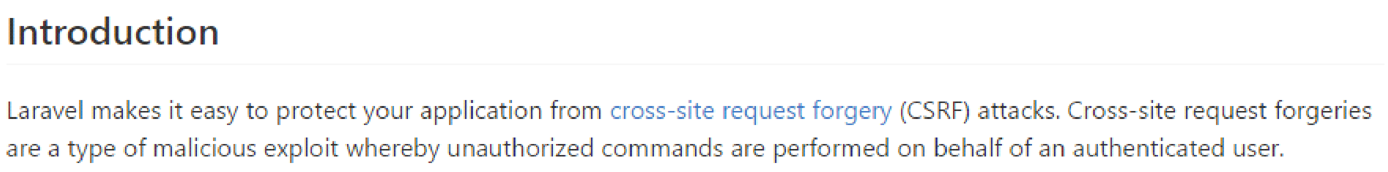
- We can give 'r' option while making controller to make it resourcefull.



```
➜ blog php artisan make:controller TasksController -r
Controller created successfully.
➜ blog █
```

- Now this controller will have template for all functions inside a controller.

Csrf_token



Introduction

Laravel makes it easy to protect your application from cross-site request forgery (CSRF) attacks. Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user.

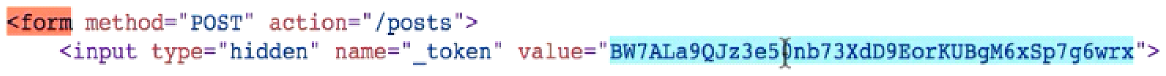
Laravel automatically generates a CSRF "token" for each active user session managed by the application. This token is used to verify that the authenticated user is the one actually making the requests to the application.

Anytime you define a HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the `csrf_field` helper to generate the token field:

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

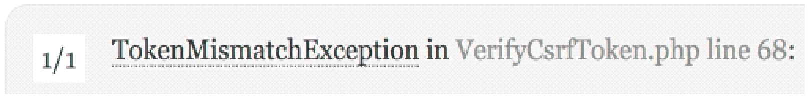
The `VerifyCsrfToken` middleware, which is included in the `web` middleware group, will automatically verify that the token in the request input matches the token stored in the session.

- This field will evaluate to a hidden field like this



```
<form method="POST" action="/posts">
    <input type="hidden" name="_token" value="BW7ALa9QJz3e50nb73XdD9EorKUBgM6xSp7g6wrx">
```

- And if we not included {{ csrf_field() inside our <form > we get following error.



1/1 TokenMismatchException in VerifyCsrfToken.php line 68:

- We may also tell laravel to exclude some urls from token checking , you can do this. see docs.

Handling Request Data

- There are two ways of access request data
  - Using 'Request' class' object
  - Using **request()** helper method.

Using 'Request' class' object

Accessing The Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the `Illuminate\Http\Request` class on your controller method. The incoming request instance will automatically be injected by the service container:

- If we have **Request** as argument in our controller method then request object of the form will be automatically be injected to it.

```
class UserController extends Controller
{
    /**
     * Store a new user.
     *
     * @param  Request  $request
     * @return Response
     */
    public function store(Request $request)
    {
        $name = $request->input('name');

        //
    }
}
```

- We can also have route parameters, which also be passed to function.

IN ROUTE

```
Route::put('user/{id}', 'UserController@update');
```

IN CONTROLLER

```
public function update(Request $request, $id)
{
    //
}
```

- We need not to have controller, we can also access Request in route closure.

```
use Illuminate\Http\Request;

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

Retrieving The Request Path

```
$uri = $request->path();
```

Retrieving The Request URL

```
// Without Query String...
$url = $request->url();

// With Query String...
$url = $request->fullUrl();
```

Retrieving Input

Retrieving All Input Data

You may also retrieve all of the input data as an `array` using the `all` method:

```
$input = $request->all();
```

Retrieving An Input Value

```
$name = $request->input('name');
```

You may pass a default value as the second argument to the `input` method.

```
$name = $request->input('name', 'Sally');
```

Retrieving Input Via Dynamic Properties

- Instead of using input() we can also use object like notation

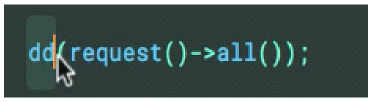
```
$name = $request->name;
```

Determining If An Input Value Is Present

```
if ($request->has('name')) {
    //
}
```

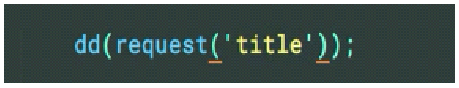
Using request() helper method

- We can access all inputs using all() function, as we do with $request object.



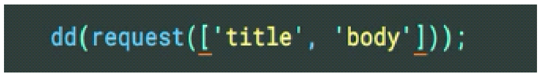
```
dd(request()->all());
```

- To get only one element we can use like this



```
dd(request('title'));
```

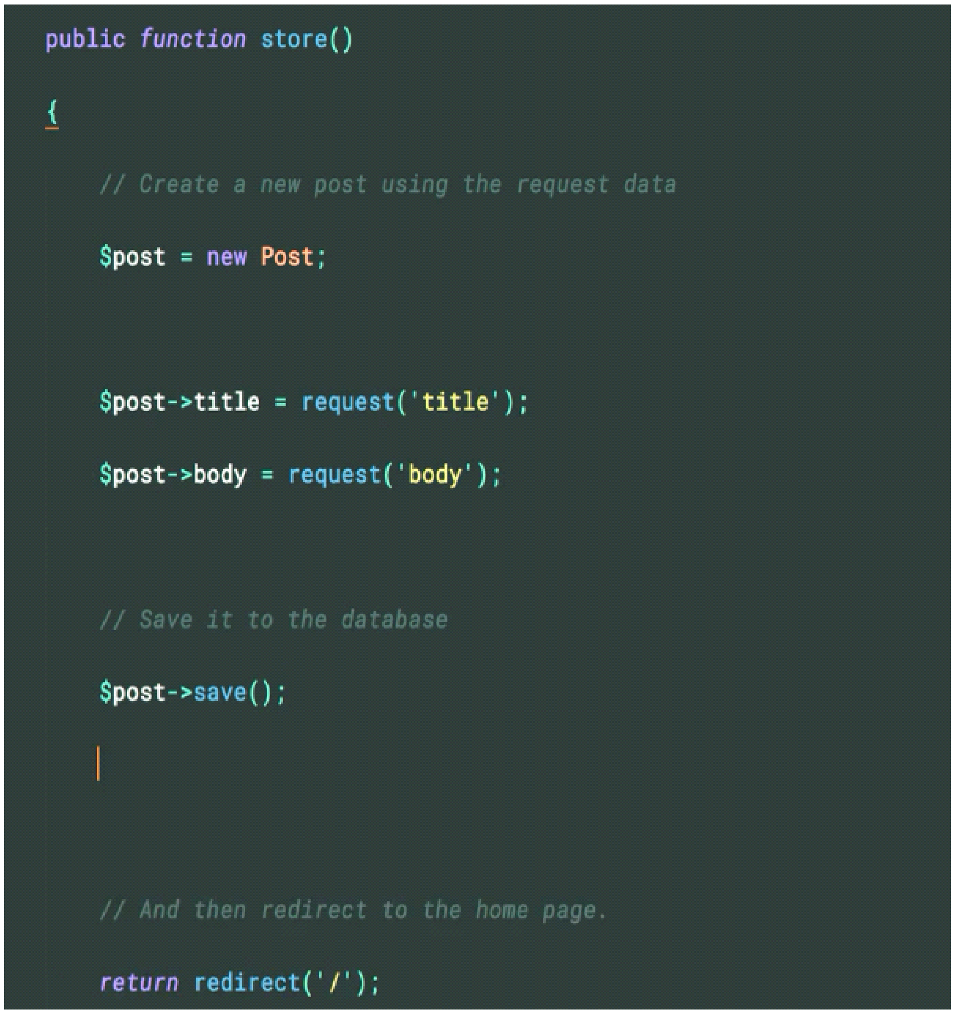
- If we want only selective of all.



```
dd(request(['title', 'body']));
```

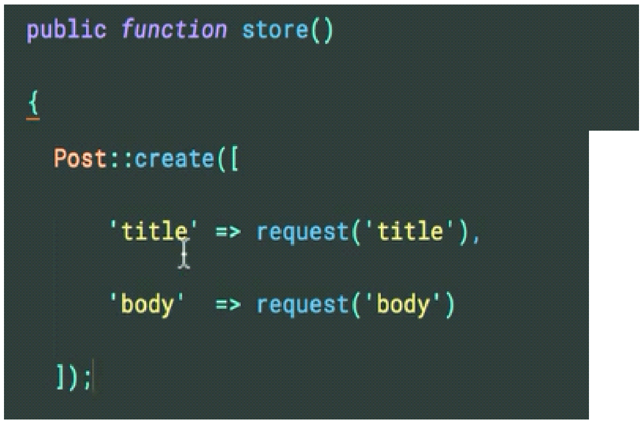
Example - storing new Post to database.

Method 1- we create a new object and store using Eloquent Model save() method)



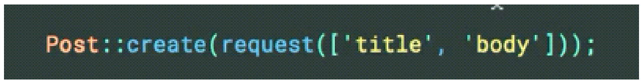
```
public function store()

{

    // Create a new post using the request data

    $post = new Post;


    $post->title = request('title');

    $post->body = request('body');


    // Save it to the database

    $post->save();




    // And then redirect to the home page.

    return redirect('/');
```

Method 2- we create a new object and Eloquent Model create())



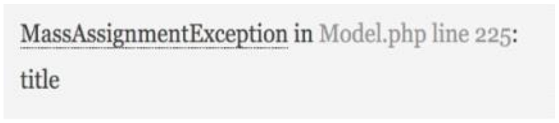
```
public function store()

{

  Post::create([

      'title' => request('title'),

      'body'  => request('body')

  ]);
```

OR



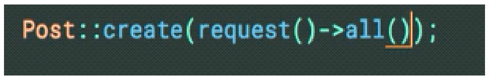
```
Post::create(request(['title', 'body']));
```

- However, if we do it we get following



```
MassAssignmentException in Model.php line 225:
title
```

because by default laravel prohibit 'mass assignment of all variable', because if we can do this someone can do something like this.



```
Post::create(request()->all());
```

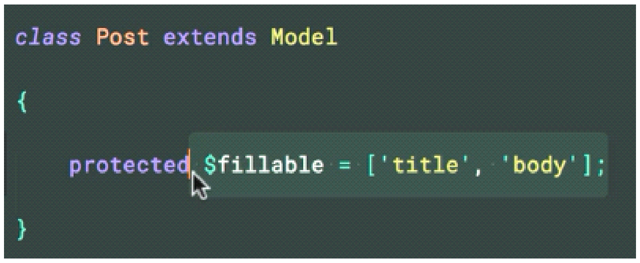
and now if someone goes to html code in browser and add new element field like,
<input type ="text" name="hello" value="some">
and post it , then it will also go to database for saving, which cause database error.

HANDLING MassAssignmentException problem

- There are two of handling this problem,
  1- Add protected $fillable in model
  2- Add protected $guarded in model

Adding $fillable in Model

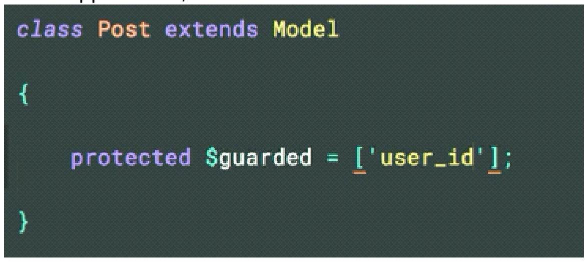
- Only those field which are there in $fillable will be allowed to pass through **create()** function.



```
class Post extends Model

{

    protected $fillable = ['title', 'body'];

}
```

Adding $guarded in Model

- Only those field which are there in $guarded will not be allowed to pass through **create()** function.
- This is opposite to $fillable.



```
class Post extends Model

{

    protected $guarded = ['user_id'];

}
```

- If $guarded set to empty, then everything will be allowed.

RULE OF THUMP - never use request()->all() in create and set $guarded to empty.

Form Validation

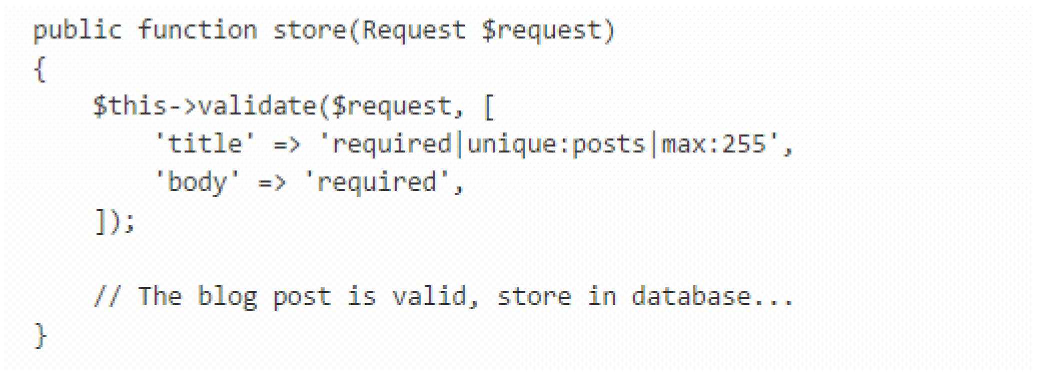
Wednesday, March 29, 2017 1:10 AM

- We should always use two layers of validation
  1- HTML 5 validaton
  2- Laravel server side validation

- As we know in HTML5 validation we use 'required' attribute in <input> field
  and when we give type='email' it make sure that it is email type.

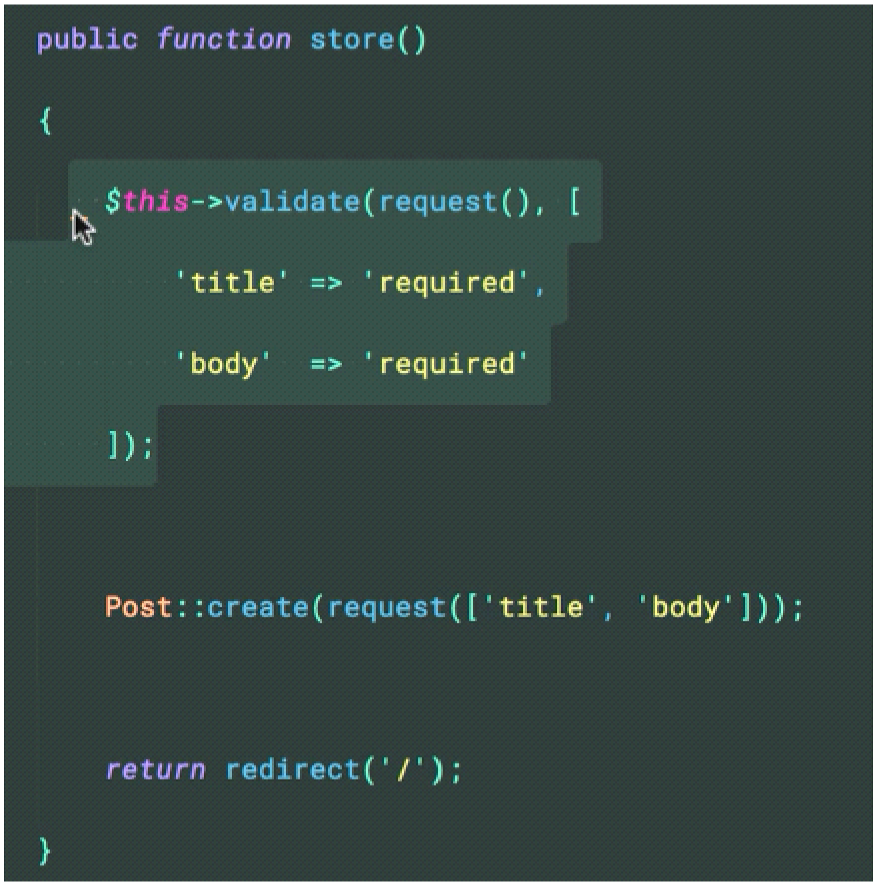
Laravel Validation

- Laravel provides validate() function to each Controller object(by inheriting Controller Class), which lets us do valdition using 'Request' object or request() magic method.



```
public function store(Request $request)
{
    $this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // The blog post is valid, store in database...
}
```

OR



```
public function store()

{

    $this->validate(request(), [

        'title' => 'required',

        'body'  => 'required'

    ]);


    Post::create(request(['title', 'body']));


    return redirect('/');

}
```

- Validate method , take inputs from request and then validate with given rules,
  if any rule fails, then it redirect back to same page with **$error** variable populated,
  If succedded it go forward and see where we want to redirect (as above).

NOTE - **$error** variable will be available to every view always.

Redirects

Wednesday, March 29, 2017 6:19 PM

redirect() method

- The simplest method is to use the global `redirect` helper:

```
Route::get('dashboard', function () {
    return redirect('home/dashboard');
});
```

back() method

Sometimes you may wish to redirect the user to their previous location, such as when a submitted form is invalid. You may do so by using the global `back` helper function. Since this feature utilizes the session, make sure the route calling the `back` function is using the `web` middleware group or has all of the session middleware applied:

- You can return back without anything, like this

```
return back();
```

- Or you can return all request input back, like this.

```
Route::post('user/profile', function () {
    // Validate the request...

    return back()->withInput();
});
```

- We can also redirects back with Errors using withErrors()
  - If there are validation errors we can do this in this page,
    Now the errors in **$error** will go to that page.

```
return back()->withErrors();
```

  - Or we can also be specific about the errors and send ours.

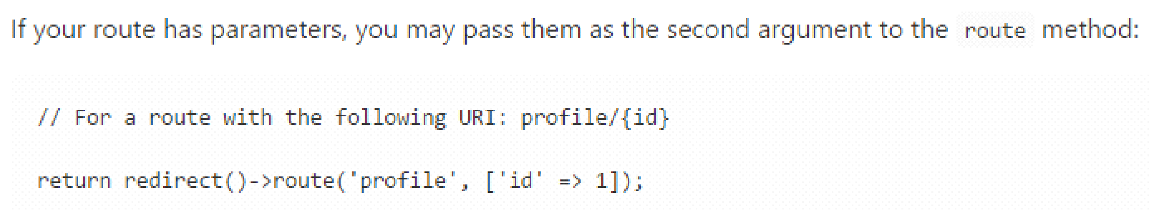
```
return back()->withErrors([
    'message' => 'Please check your credentials and try again.'
]);
```

NOTE - errors will always go to **$error** array.

Redirecting To Named Routes

When you call the `redirect` helper with no parameters, an instance of `Illuminate\Routing\Redirector` is returned, allowing you to call any method on the `Redirector` instance. For example, to generate a `RedirectResponse` to a named route, you may use the `route` method:

```
return redirect()->route('login');
```



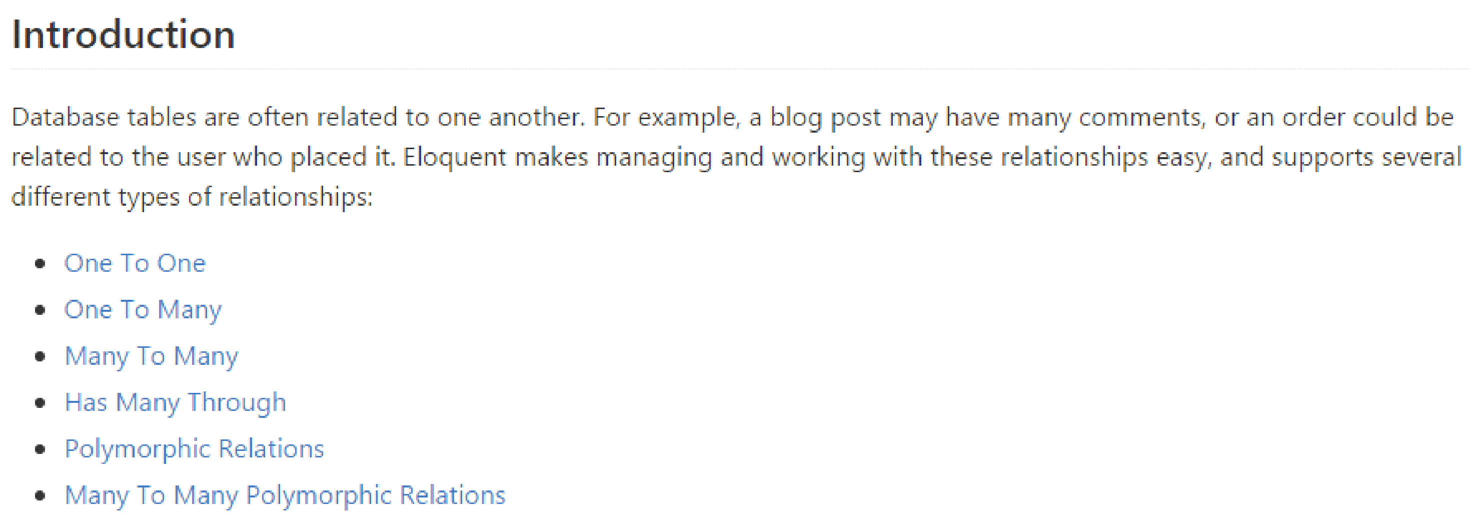
If your route has parameters, you may pass them as the second argument to the `route` method:

```
// For a route with the following URI: profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

For more see doc

Eloquent Relationships

Wednesday, March 29, 2017 5:39 PM



Introduction

Database tables are often related to one another. For example, a blog post may have many comments, or an order could be related to the user who placed it. Eloquent makes managing and working with these relationships easy, and supports several different types of relationships:

- One To One
- One To Many
- Many To Many
- Has Many Through
- Polymorphic Relations
- Many To Many Polymorphic Relations

NOTE -

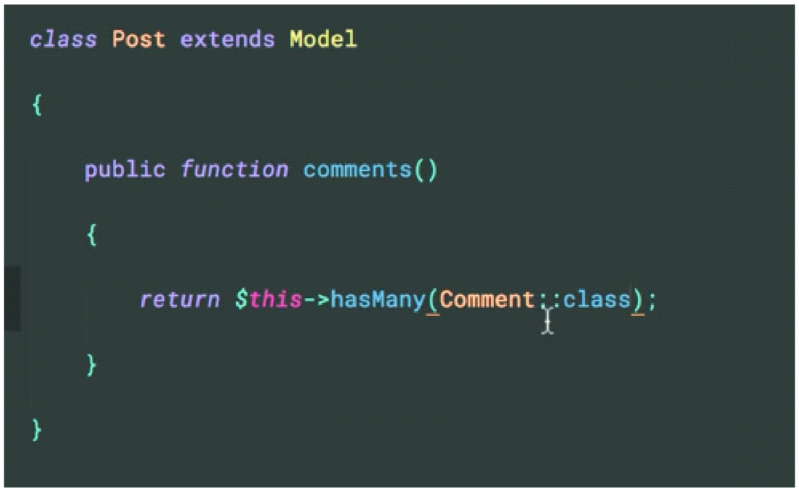
- oneToOne relationship can be written in two ways
  1- HasOne
  2- belongsTo(inverse of above)
- oneToOne relationship can be written in two ways
  1- HasMany
  2- belongsTo(inverse of above)

CASE STUDY

- Lets we are creating are creating comments for a post,
  So we create comment model and table(having post_id).
- Now we establish relationship between Post model and Comment Model.
  Clearly ,
  "a post has many comments"
  So relationship is 'has many'

Fetching comments of a post

- Now , we want to create a method inside Post model 'comments()'
  which will return all comment related to this post, we can use relationship like,

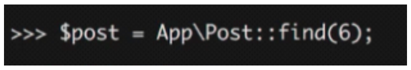


```
class Post extends Model

{

    public function comments()

    {

        return $this->hasMany(Comment::class);

    }

}
```

- Here **Comment::class** will return string representation of full class path.
- Once the relationship is defined, we may retrieve the related record using Eloquent's dynamic properties. Dynamic properties allow you to access relationship functions as if they were

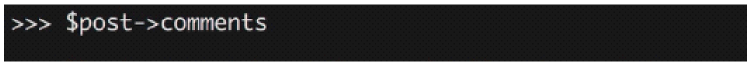
properties defined on the model.(so we can access as properties)

- So accessing comment of a post object
    - First get post object



```
>>> $post = App\Post::find(6);
```

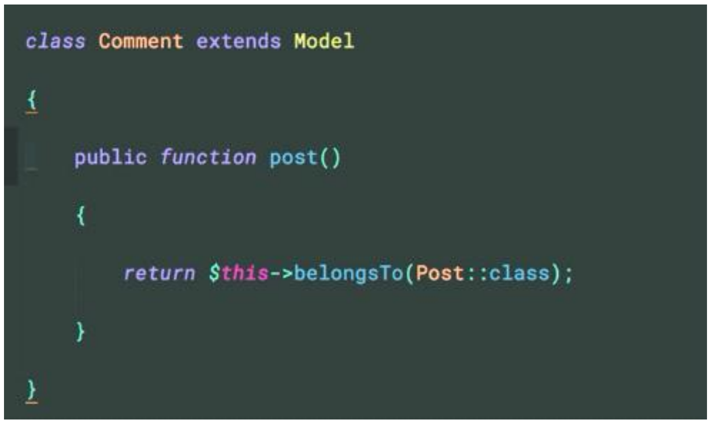
- Now, get comments as property.



```
>>> $post->comments
```

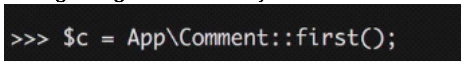
Fetching Post related to a comment.

- Relationship - **belongTo (inverse)**



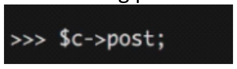
```
class Comment extends Model

{

    public function post()

    {

        return $this->belongsTo(Post::class);

    }

}
```

- Now getting Comment object



```
>>> $c = App\Comment::first();
```

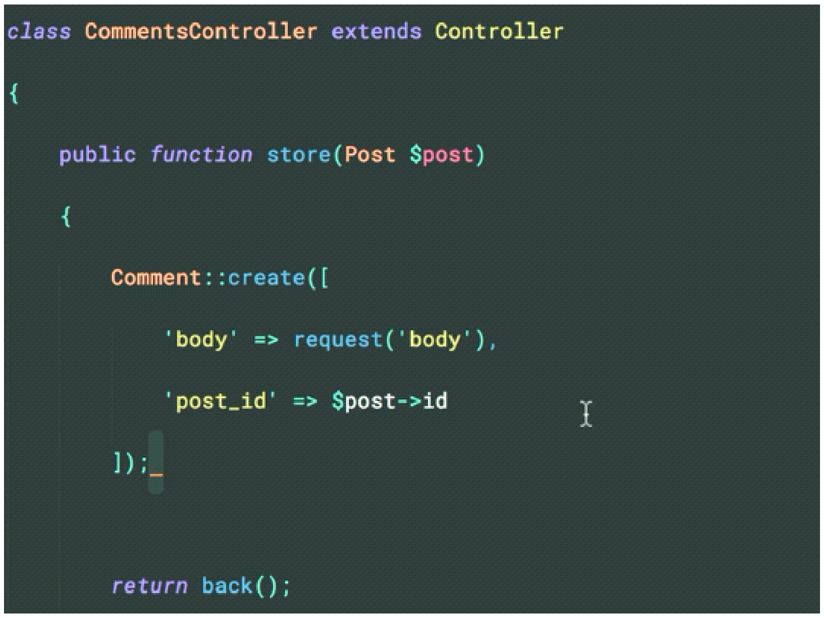
- And fetching post of that comment.



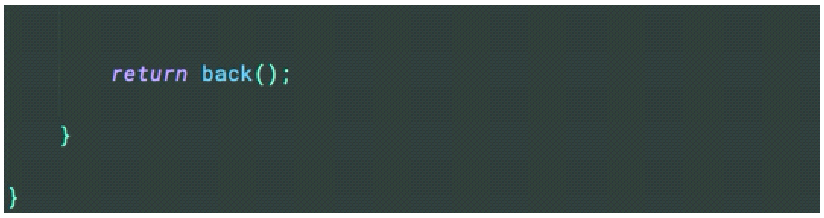
```
>>> $c->post;
```

Adding a comment to a post

- METHOD 1 - We can use long form way like this, using CommentController@store (or make a function inside Comment Model)

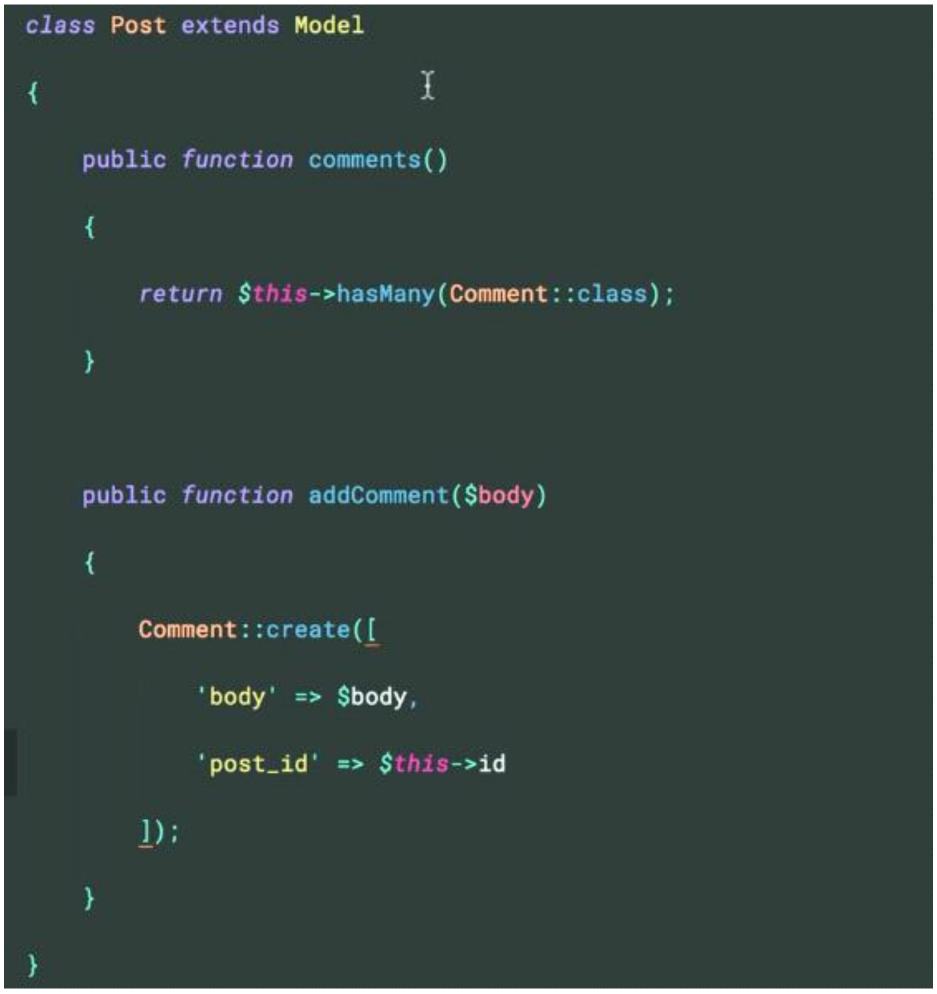


```
class CommentsController extends Controller

{

    public function store(Post $post)

    {

        Comment::create([

            'body' => request('body'),

            'post_id' => $post->id

        ]);


        return back();
```



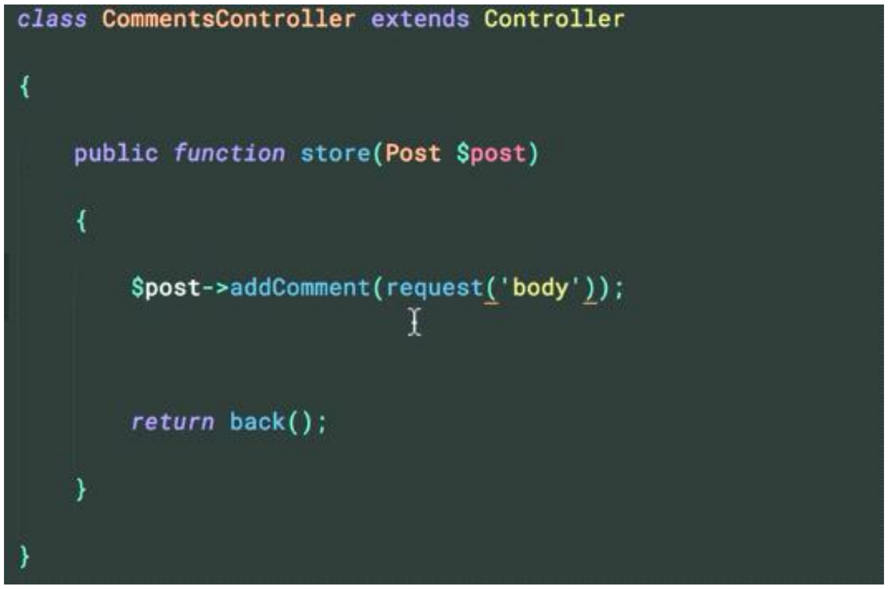
```
        return back();

    }

}
```

- METHOD 2 - Or We can also create a method like, Post@addComment



```
class Post extends Model

{

    public function comments()

    {

        return $this->hasMany(Comment::class);

    }


    public function addComment($body)

    {

        Comment::create([

            'body' => $body,

            'post_id' => $this->id

        ]);

    }

}
```

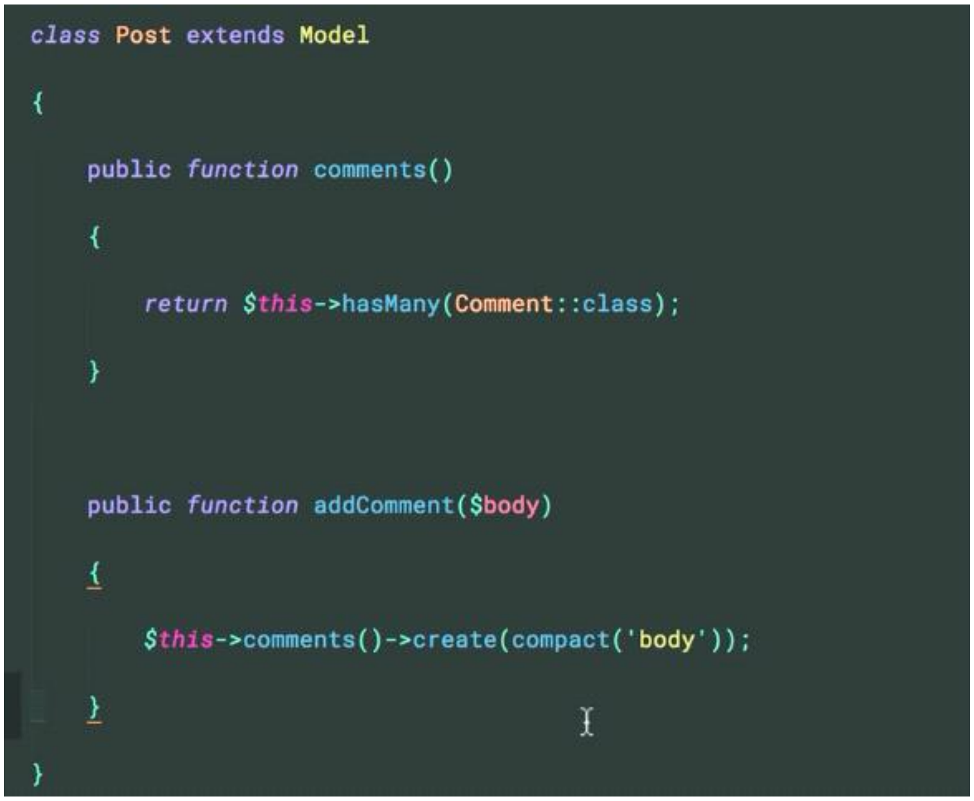
- And we are coming from PostController here,



```php
class CommentsController extends Controller

{

    public function store(Post $post)

    {

        $post->addComment(request('body'));


        return back();

    }

}
```

METHOD 3 - (best) using eloquent Relationships

- As we already have method comments() inside Post we can use it to create a post, amazing thing is that it will automatically insert id of Post, because of relationship.



```php
class Post extends Model

{

    public function comments()

    {

        return $this->hasMany(Comment::class);

    }


    public function addComment($body)

    {

        $this->comments()->create(compact('body'));

    }

}
```

- Other things remain same as method - 2

  NOTE - as we can see, when we use **$this->comments** it returns all comments associated with it, But when we use $this->comments(), it allows us to add comment to it, and also set post_id in commnet table.

User Authentication

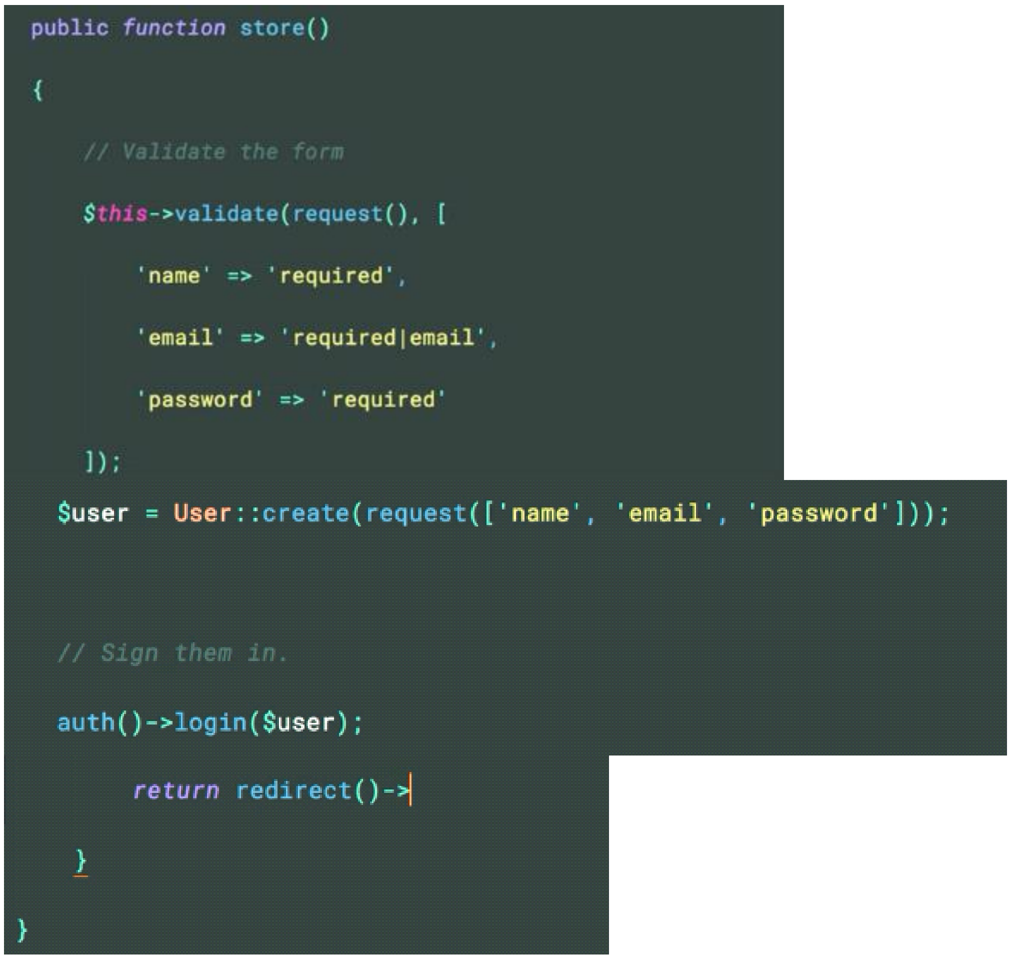
Wednesday, March 29, 2017 6:44 PM

- Laravel provides rapid authentican using ,

```
$php artisan make:auth
```

- Use bcrypt() function to hash the password

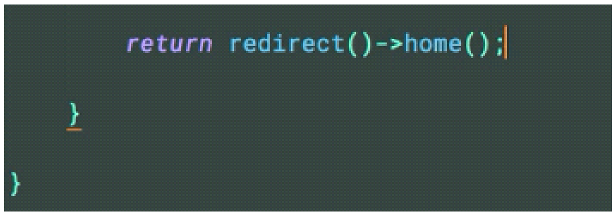
Creating Function for registering a user, after logged it in…



```
public function store()
{
    // Validate the form
    $this->validate(request(), [
        'name' => 'required',
        'email' => 'required|email',
        'password' => 'required'
    ]);
  $user = User::create(request(['name', 'email', 'password']));


  // Sign them in.
  auth()->login($user);
        return redirect()->
   }
}
```

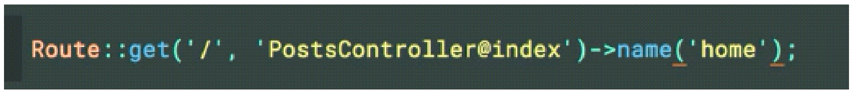
NOTE -
Here auth() is a helper function like request(), it is for Auth:: Façade.

- There is another way of redirecting to home.



```
        return redirect()->home();
    }
}
```

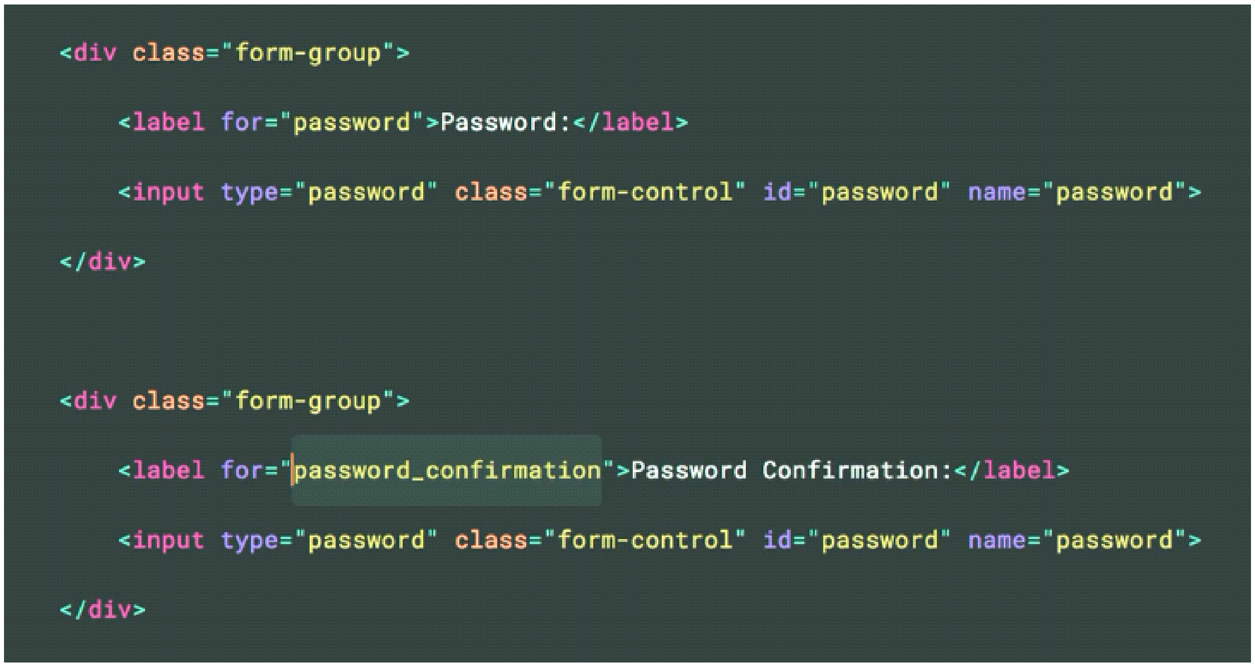
But for using this we have to name a route as home in route/web.php, like this..



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

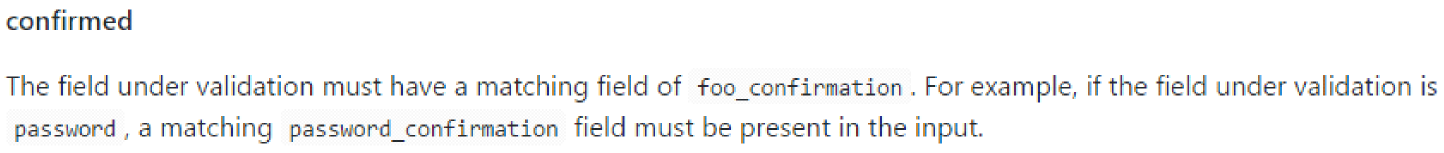
Password confirmation

- It is quite common to use password confirmation when a user is registered.



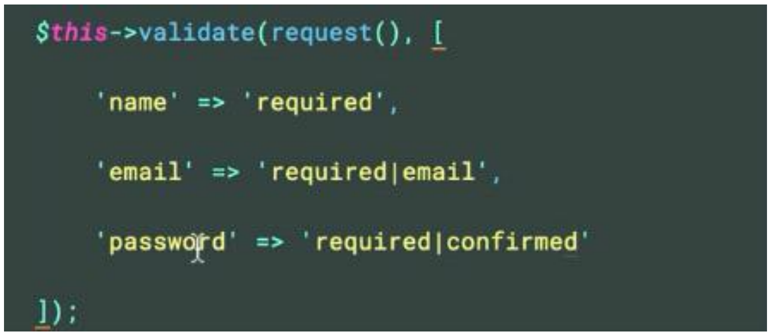
```
<div class="form-group">

    <label for="password">Password:</label>

    <input type="password" class="form-control" id="password" name="password">

</div>



<div class="form-group">

    <label for="password_confirmation">Password Confirmation:</label>

    <input type="password" class="form-control" id="password" name="password">

</div>
```

- Laravel automatically provide password confirmation, we only have to give name to password confirmation field in a specific way, it always need to be names as "password_confirmation".
- Now, we can add validation **'confirmed'** to input as 'password', then it will match with 'password_confirmation", similary, if the field name 'foo' it will match with 'foo_confirmation'.



confirmed

The field under validation must have a matching field of `foo_confirmation`. For example, if the field under validation is `password`, a matching `password_confirmation` field must be present in the input.

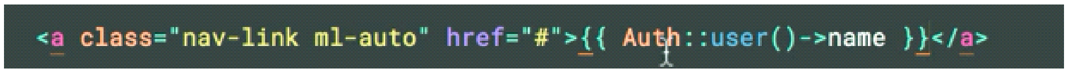


```
$this->validate(request(), [

    'name' => 'required',

    'email' => 'required|email',

    'password' => 'required|confirmed'

]);
```

- Now "password" field should be matched with "password_confirmation".

Getting information about Authenticated User

- We can use auth() helper function or Auth:: façade, to access information about , authenticated user.



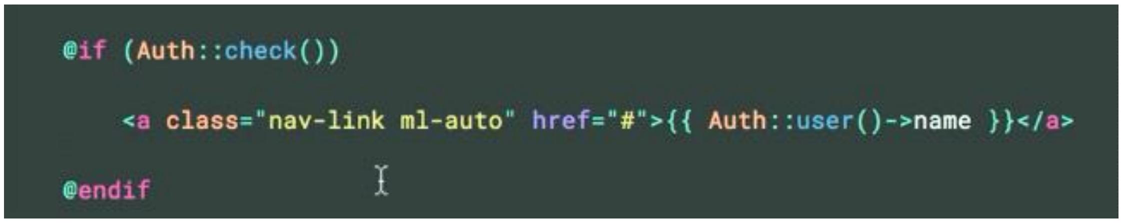
```
<a class="nav-link ml-auto" href="#">{{ Auth::user()->name }}</a>
```

Or we can also , use like **auth()->user()->name;** (we can always replace auth() with Auth::),but for Auth:: proper namespace should be included using **use.**

- Similary we can get all properties (see php tinker to know about all properties)

Checking if author is Signed in or not.

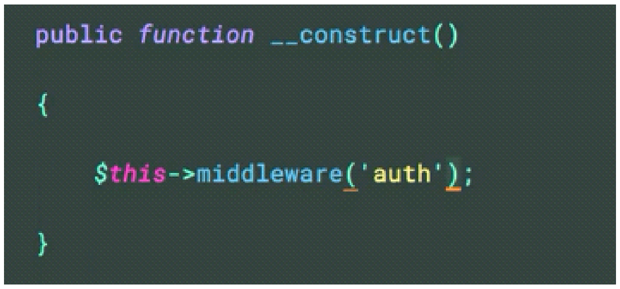
- We can use check() function.



```
@if (Auth::check())

    <a class="nav-link ml-auto" href="#">{{ Auth::user()->name }}</a>

@endif
```

Restricting user

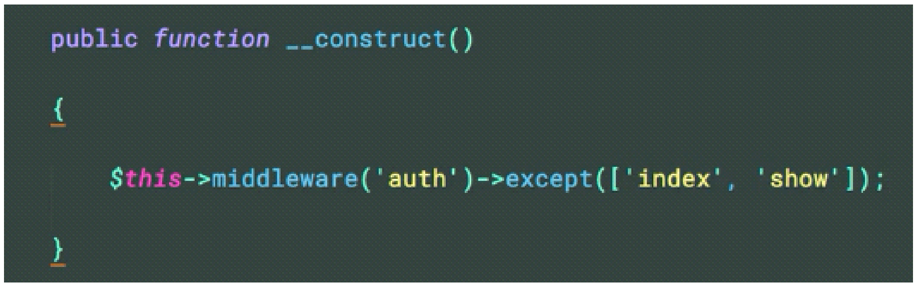
- We can restrict user using predefined middleware , or we can create our own.
- **Auth** middle ware can be used to check if the user is logged in or not.
- For this we have to register the middle ware for that controller in __constructor().



```
public function __construct()

{

    $this->middleware('auth');

}
```

- If we do this, then a guest user will not be able to excess any function of this controller.

- We can tell it to except() some functions from validation using except() function like this..



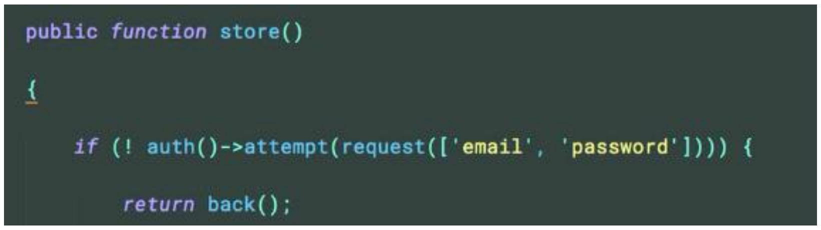
```
public function __construct()

{

    $this->middleware('auth')->except(['index', 'show']);

}
```

  - this is telling to lock down all functions except() these given,
    it will automatically redirect to **/login** if the user is not logged in.

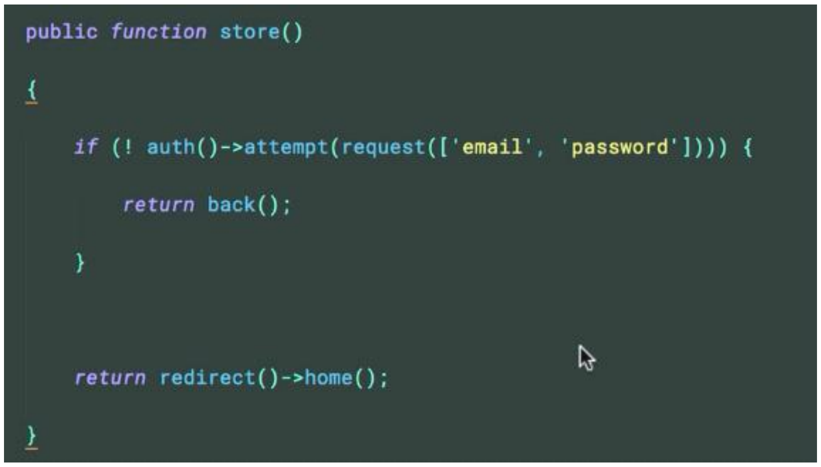
Signing in A user

- We can use auth()/Auth:: 's **attempt()** method to try signing in the user from some credentials.
- If credentials are right, it will automatically signin and return **true**,
  otherwise it returns **false.**

- Inside SessionController@store, in which we are signing in,



```
public function store()

{

    if (! auth()->attempt(request(['email', 'password']))) {

        return back();
```

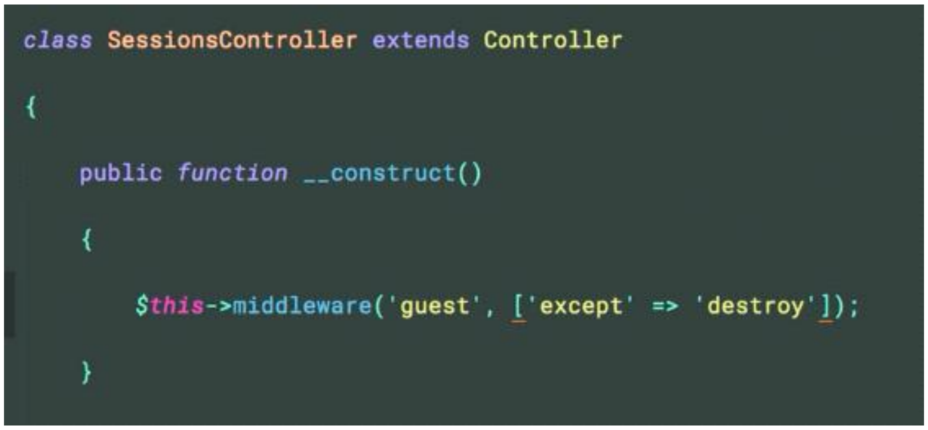


```
public function store()
{
    if (! auth()->attempt(request(['email', 'password']))) {
        return back();
    }

    return redirect()->home();
}
```

- Offcourse it will pass the test if able to sign in , using credentials.

Using Guest Middleware

- We can use guest middleware to check If the user is guest or not,
  because it may not be right to show login page to user if the user is already logged in.
- So we will add a guest filter in SessionController@__construct



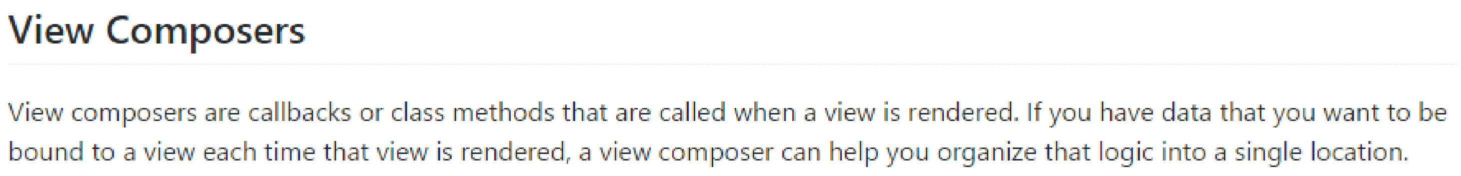
```
class SessionsController extends Controller
{
    public function __construct()
    {
        $this->middleware('guest', ['except' => 'destroy']);
    }
```

- Now it will not let any user to go to any function except 'logout' function 'destroy' and redirect to **/home**, if the user is not right.
  we can create route **/home** or edit 'guest' middle ware to redirect to anyother route.

View Composers

Thursday, March 30, 2017 12:29 AM

- Suppose we have some variable which is same for all views, what we can do ,
  - Method 1 - copy and paste the code for getting that variable for all controllers, and pass variable to every view.
  - Method 2 - use view composers.



View Composers

View composers are callbacks or class methods that are called when a view is rendered. If you have data that you want to be bound to a view each time that view is rendered, a view composer can help you organize that logic into a single location.

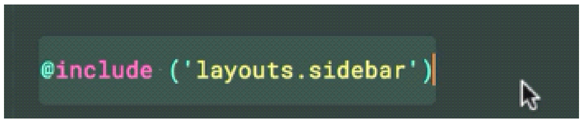
- you'll encounter the situation when multiple views require the same piece of data. For example, if a sidebar widget requires a particular variable for every single page, well we can't be expected to pass that data from every single controller. No, instead, we can leverage view composers.

- View Composers are function which will be called when a view is loaded.

- View composers will be stored in service providers, which are stored in **app/Providers ,**
  Inside this directory there is a service provider **AppServiceProvider.**php which is like a dedicated service provider for us to get started.
- There are two function in that service providers
  - register() - used to register things (we will study later)
  - boot() - whatever this method be called on each registered providers, after calling register(),,
    So we can register our view composers here.

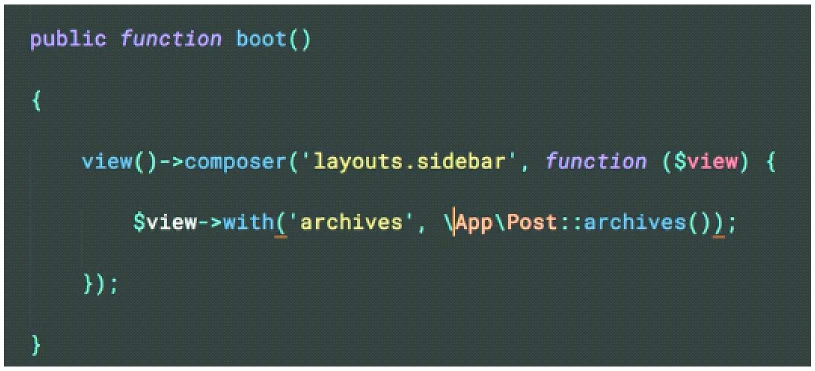
Registering View composers

- Here we have loaded our archieves using view layouts/sidebar.
  so we wanted **$archeive** variable which have all our information about archieves to present every where this view is loaded.



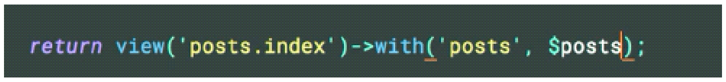
```
@include ('layouts.sidebar')
```

- We can register composer using **View::composer()** or **view()->composer()**



```
public function boot()

{

    view()->composer('layouts.sidebar', function ($view) {

        $view->with('archives', \App\Post::archives());

    });

}
```

  - Here 'layouts.sidebar' is view for which we are making 'view composer'
  - view object will be passed in $view argument of callback(we can also use dedicated like Controllers), we use this $view object to bind something with the view.
  - Now we want to pass, variable so
    - With(<variable_name>, <value_passed>)
  - We can notice this syntax is same as we use to pass variable in views through routes/controllers.



```php
return view('posts.index')->with('posts', $posts);
```

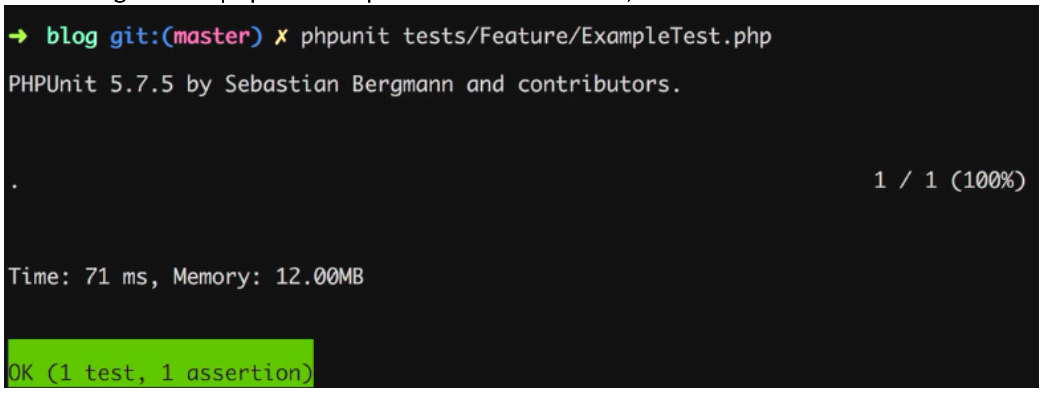
Testing

Thursday, March 30, 2017 1:46 AM

- There is a tests directory inside our laravel project , which contains two folder
    - Unit - support unit testing(testing simple parts)
    - Feature - support Integrated testing
- It is good to create testing for each unit, so when in future we refactor anything we can test it right away.

- **Phpunit** comes as a dependency(as associate) with laravel project, so will have it with every laravel project.
  we can invoke using **phpunit command,** or **vendor/bin/phpunit** (add vendor/bin in path)

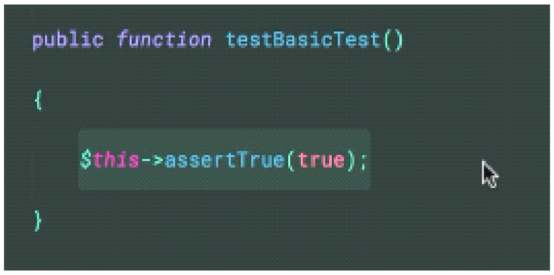
- For testing we use phpunit and pass the testfile with it, like



```
➜ blog git:(master) ✗ phpunit tests/Feature/ExampleTest.php
PHPUnit 5.7.5 by Sebastian Bergmann and contributors.

.                                                                  1 / 1 (100%)

Time: 71 ms, Memory: 12.00MB

OK (1 test, 1 assertion)
```

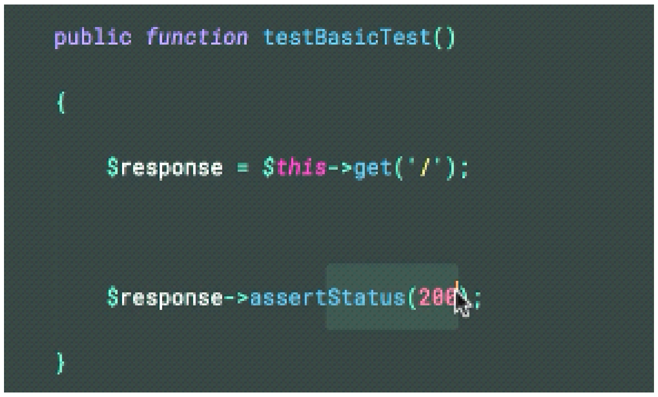
Asserting

- We can assert if something comes out to be true, using **assertTrue()** method



```
public function testBasicTest()
{
    $this->assertTrue(true);
}
```

- We can check if we get status code to something on going to some page.



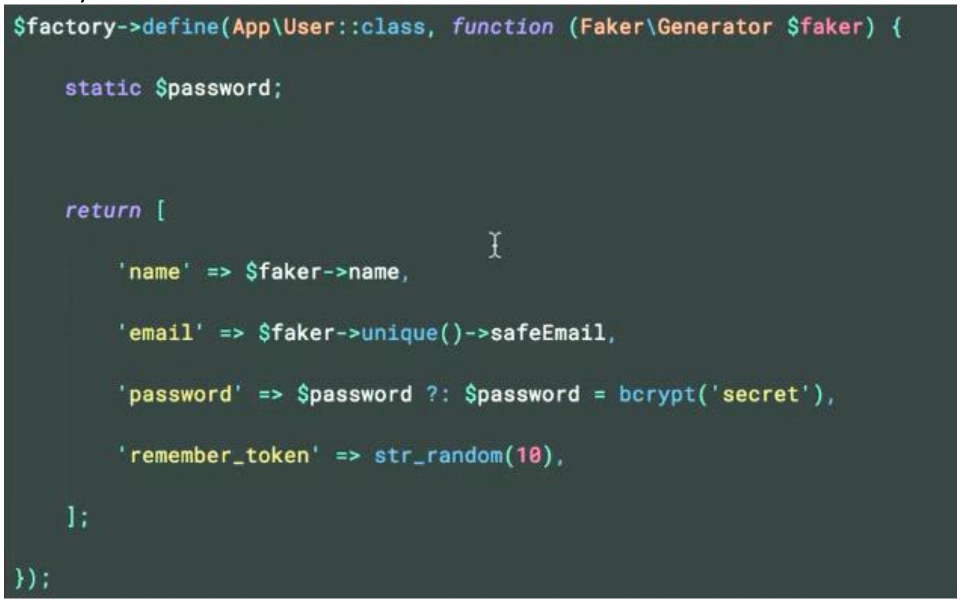
```
public function testBasicTest()
{
    $response = $this->get('/');

    $response->assertStatus(200);
}
```

Here we are making **get()** request to " / " and asserting if status code is 200.

- There are many assertions available at http://laravel.com/docs/5.4/http-tests.html#available-assertions

Model Factories

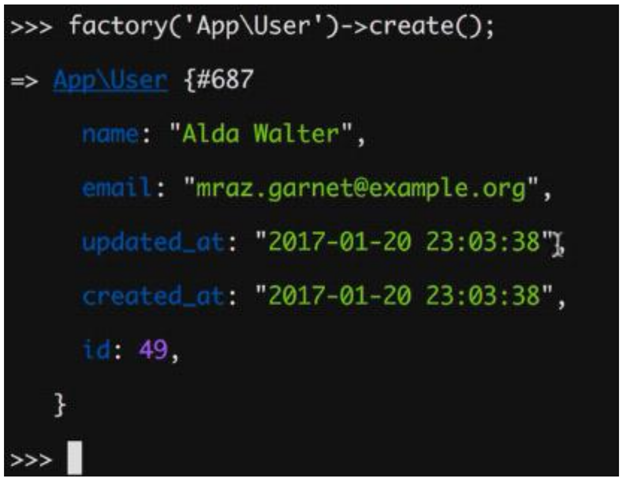
- Model factories can be used to generate fake objects for testing.
- by Default model factory for "user" elequont model is provided inside **/database/factories/ModelFactory.php**
- Factory is blue print for a elequont model.
- Factory for use looks like this



```
$factory->define(App\User::class, function (Faker\Generator $faker) {

    static $password;


    return [

        'name' => $faker->name,

        'email' => $faker->unique()->safeEmail,

        'password' => $password ?: $password = bcrypt('secret'),

        'remember_token' => str_random(10),

    ];

});
```

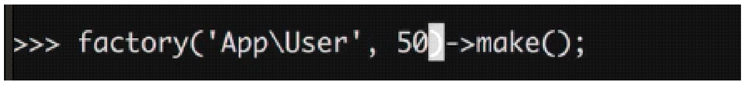
as we can see, it has a call back which returns the fake object using a $faker object of class "Faker\Generator".

- We can use factory() helper method to create dummy object of a model.



```
>>> factory('App\User')->create();

=> App\User {#687

     name: "Alda Walter",

     email: "mraz.garnet@example.org",

     updated_at: "2017-01-20 23:03:38",

     created_at: "2017-01-20 23:03:38",

     id: 49,

   }

>>> 
```

- We can create many fake object by passing another argument as number of element to the factory() method

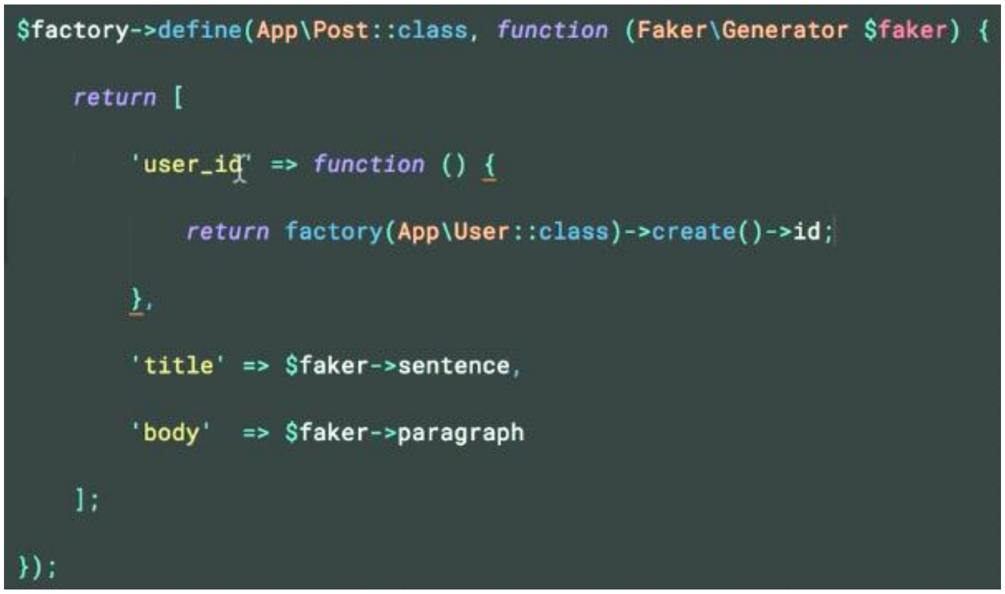


```
>>> factory('App\User', 50)->make();
```

- We can use this technique not only for testing but also for database seeding(filling fake database)

Example - creating Factory for posts?

- Of course the post consist of user_id,title and body.
- But for getting a user we have to create a new user also(fake user) and then get the id.



```
$factory->define(App\Post::class, function (Faker\Generator $faker) {

    return [

        'user_id' => function () {

            return factory(App\User::class)->create()->id;

        },

        'title' => $faker->sentence,

        'body'  => $faker->paragraph

    ];

});
```

- The function will be called when we look for 'user_id' which will return the value of id from newly created user.
- Here when we use create() method, the data is made persistant in database, we can use make() function which do not persist data , so do not generate fields like **id,created_at,updated_at(which are generated at db insertion).**

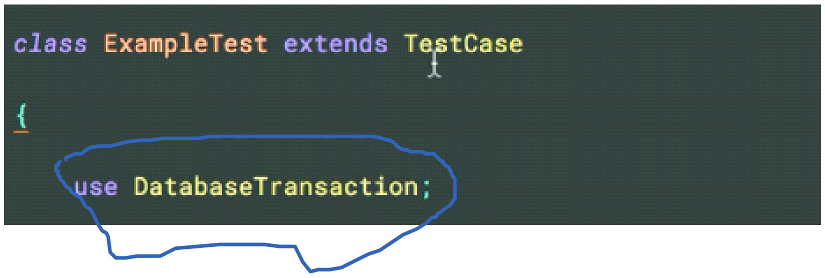
Not Persisting Testing data

- When we test something we create some data and store in the database, but we do not want to keep that data for ever.
- There are two traits which comes to the Testing files by default and we can use them by using **use** keyword.

```
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;
```

  - DatabaseMigrations work by migrations and when test finished it roll back migrations
  - DatabaseTransactions work by transactions and when test finished it roll back transactions.

- So we just need to add the follosing line in our class in test file to enable DatabaseTransactions Trait, which will persist data using transactions and roll back all transactions after test is done.



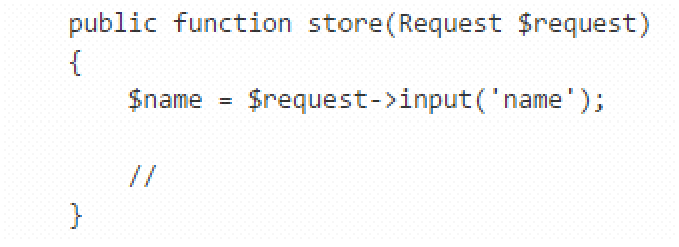
```
class ExampleTest extends TestCase

{

    use DatabaseTransaction;
```

Watch video for more information.

Dependency Injection

Tuesday, May 2, 2017 11:58 PM

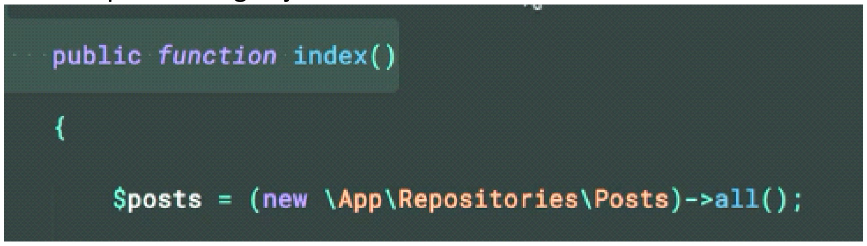
- Dependency for a method are the arguments which are passed to it.
- Laravel support automatica DI to every constructor and its method.
  Laravel automatically pass the object argument which it could be able to pass by using PHP's reflection API.
  Laravel solved the dependency recursively , so if the object required want another object in its constructor then its object is passed by laravel and so on.
- We have already seen example of DI when the object of request is automatically passed by laravel inside controller class.



```
public function store(Request $request)
{
    $name = $request->input('name');

    //
}
```

- We can apply this kind of injection to every where, and laravel tries its best to pass the object to us.

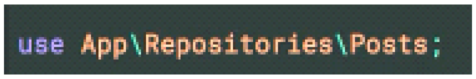
Example - let we want to get a object of **Posts** class to call function all()
Method 1- simple creating object



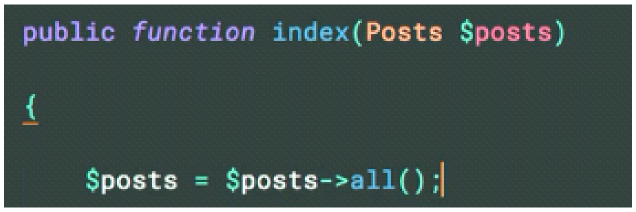
```
public function index()
{
    $posts = (new \App\Repositories\Posts)->all();
```

Method 2- dependency injection.

- Add following line first(can add in method1 also)



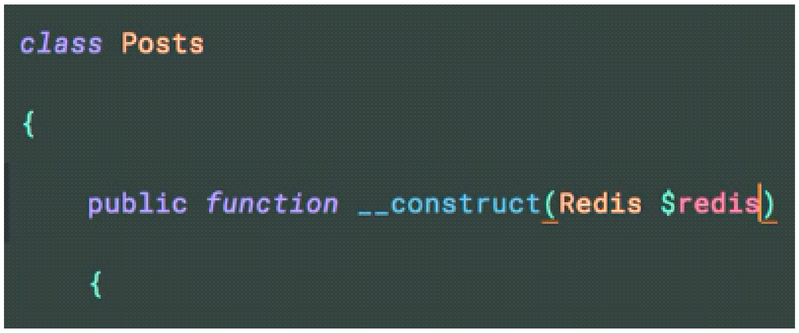
```
use App\Repositories\Posts;
```



```
public function index(Posts $posts)
{
    $posts = $posts->all();
```

- This is called automatic resolution or automatic DI.

- Now, suppose **Posts** constructor depend up another constructor **Redis** class

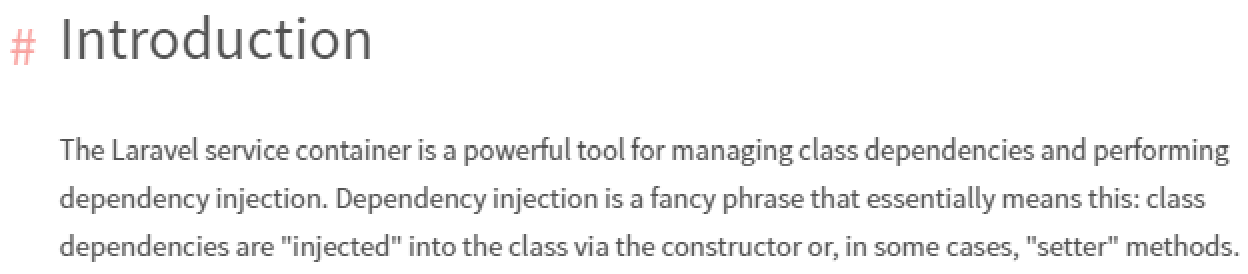


```
class Posts

{

    public function __construct(Redis $redis)

    {
```

- Now laravel will make object or Radis class, then pass the object reference to constructor of Posts and then give back object.
  laravel can resolve any number of nested dependencies.

Service Container

Wednesday, May 3, 2017 12:26 AM



Introduction

The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. Dependency injection is a fancy phrase that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

Routing Basics

Tuesday, March 28, 2017 3:33 PM

Basic Routing

- The most basic Laravel routes simply accept a URI and a `Closure`, providing a very simple and expressive method of defining routes:

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

Available Router Methods

- The router allows you to register routes that respond to any HTTP verb:

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

- Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the `match` method. Or, you may even register a route that responds to all HTTP verbs using the `any` method:

```
Route::match(['get', 'post'], '/', function () {
    //
});
Route::any('foo', function () {
    //
});
```

CSRF Protection

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

```
<form method="POST" action="/profile">
    {{ csrf_field() }}
    ...
</form>
```

Migrations basics

Tuesday, March 28, 2017 3:45 PM

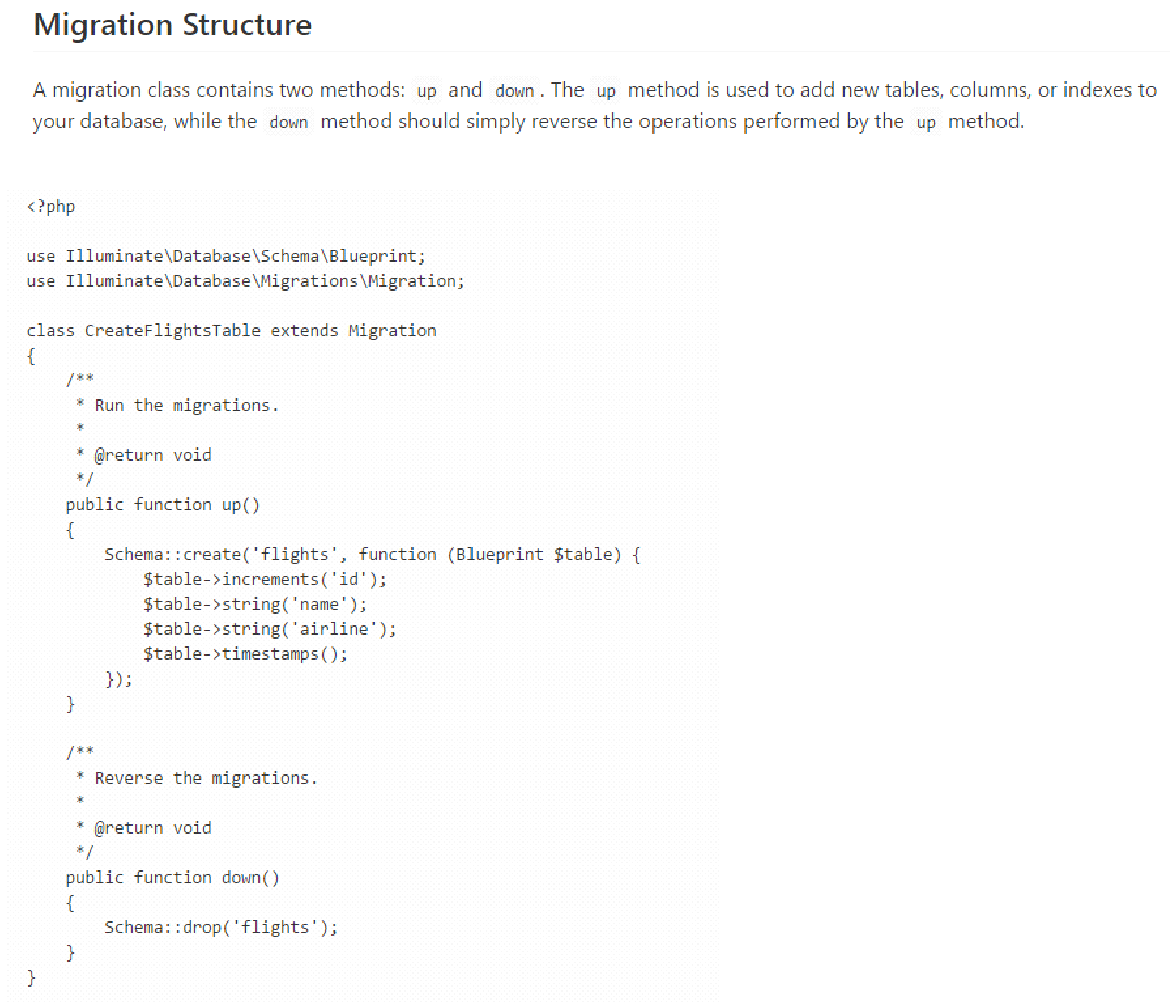
Introduction

Migrations are like version control for your database, allowing your team to easily modify and share the application's database schema. Migrations are typically paired with Laravel's schema builder to easily build your application's database schema.

Generating Migrations

To create a migration, use the `make:migration` Artisan command:

```
php artisan make:migration create_users_table
```



Migration Structure

A migration class contains two methods: `up` and `down`. The `up` method is used to add new tables, columns, or indexes to your database, while the `down` method should simply reverse the operations performed by the `up` method.

```
<?php

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

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

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

Running Migrations

To run all of your outstanding migrations, execute the `migrate` Artisan command:

```
php artisan migrate
```

Rolling Back Migrations

To rollback the latest migration operation, you may use the `rollback` command. This command rolls back the last "batch" of migrations, which may include multiple migration files:

```
php artisan migrate:rollback
```

You may rollback a limited number of migrations by providing the `step` option to the `rollback` command. For example, the following command will rollback the last five migrations:

```
php artisan migrate:rollback --step=5
```

The `migrate:reset` command will roll back all of your application's migrations:

```
php artisan migrate:reset
```

Rollback & Migrate In Single Command

The `migrate:refresh` command will roll back all of your migrations and then execute the `migrate` command. This command effectively re-creates your entire database:

```
php artisan migrate:refresh

// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

You may rollback & re-migrate a limited number of migrations by providing the `step` option to the `refresh` command. For example, the following command will rollback & re-migrate the last five migrations:

```
php artisan migrate:refresh --step=5
```

Tables

Creating Tables

To create a new database table, use the `create` method on the `Schema` facade. The `create` method accepts two arguments. The first is the name of the table, while the second is a `Closure` which receives a `Blueprint` object that may be used to define the new table:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
});
```

Renaming / Dropping Tables

To rename an existing database table, use the `rename` method:

```
Schema::rename($from, $to);
```

To drop an existing table, you may use the `drop` or `dropIfExists` methods:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Columns

Creating Columns

The `table` method on the `Schema` facade may be used to update existing tables. Like the `create` method, the `table` method accepts two arguments: the name of the table and a `Closure` that receives a `Blueprint` instance you may use to add columns to the table:

```
Schema::table('users', function ($table) {
    $table->string('email');
});
```

- For type of variables available see documentation.

What are Middlewares?

Friday, September 15, 2017 1:53 PM

Introduction

Middleware provide a convenient mechanism for filtering HTTP requests entering your application. For example, Laravel includes a middleware that verifies the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application.

Of course, additional middleware can be written to perform a variety of tasks besides authentication. A CORS middleware might be responsible for adding the proper headers to all responses leaving your application. A logging middleware might log all incoming requests to your application.

There are several middleware included in the Laravel framework, including middleware for authentication and CSRF protection. All of these middleware are located in the `app/Http/Middleware` directory.

Defining Middleware

To create a new middleware, use the `make:middleware` Artisan command:

```
php artisan make:middleware CheckAge
```

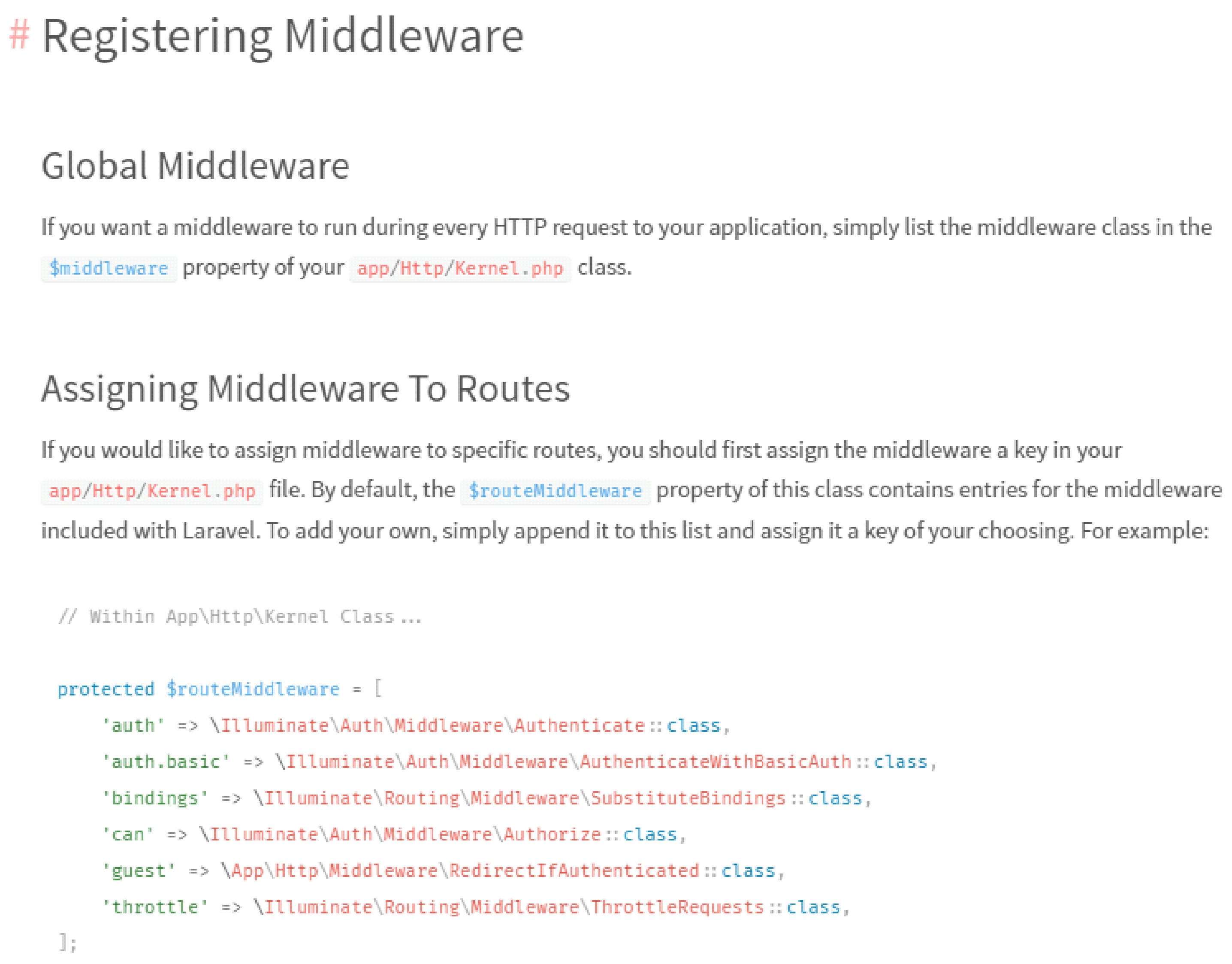
This command will place a new `CheckAge` class within your `app/Http/Middleware` directory. In this middleware, we will only allow access to the route if the supplied `age` is greater than 200. Otherwise, we will redirect the users back to the `home` URI.

```php
<?php

namespace App\Http\Middleware;

use Closure;

class CheckAge
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->age <= 200) {
            return redirect('home');
        }

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

As you can see, if the given `age` is less than or equal to `200`, the middleware will return an HTTP redirect to the client; otherwise, the request will be passed further into the application. To pass the request deeper into the application (allowing the middleware to "pass"), simply call the `$next` callback with the `$request`.

Registering Middleware

Friday, September 15, 2017 7:46 PM



Registering Middleware

Global Middleware

If you want a middleware to run during every HTTP request to your application, simply list the middleware class in the `$middleware` property of your `app/Http/Kernel.php` class.

Assigning Middleware To Routes

If you would like to assign middleware to specific routes, you should first assign the middleware a key in your `app/Http/Kernel.php` file. By default, the `$routeMiddleware` property of this class contains entries for the middleware included with Laravel. To add your own, simply append it to this list and assign it a key of your choosing. For example:

```
// Within App\Http\Kernel Class...

protected $routeMiddleware = [
    'auth' => \Illuminate\Auth\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
];
```

Once the middleware has been defined in the HTTP kernel, you may use the `middleware` method to assign middleware to a route:

```
Route::get('admin/profile', function () {
    //
})->middleware('auth');
```

You may also assign multiple middleware to the route:

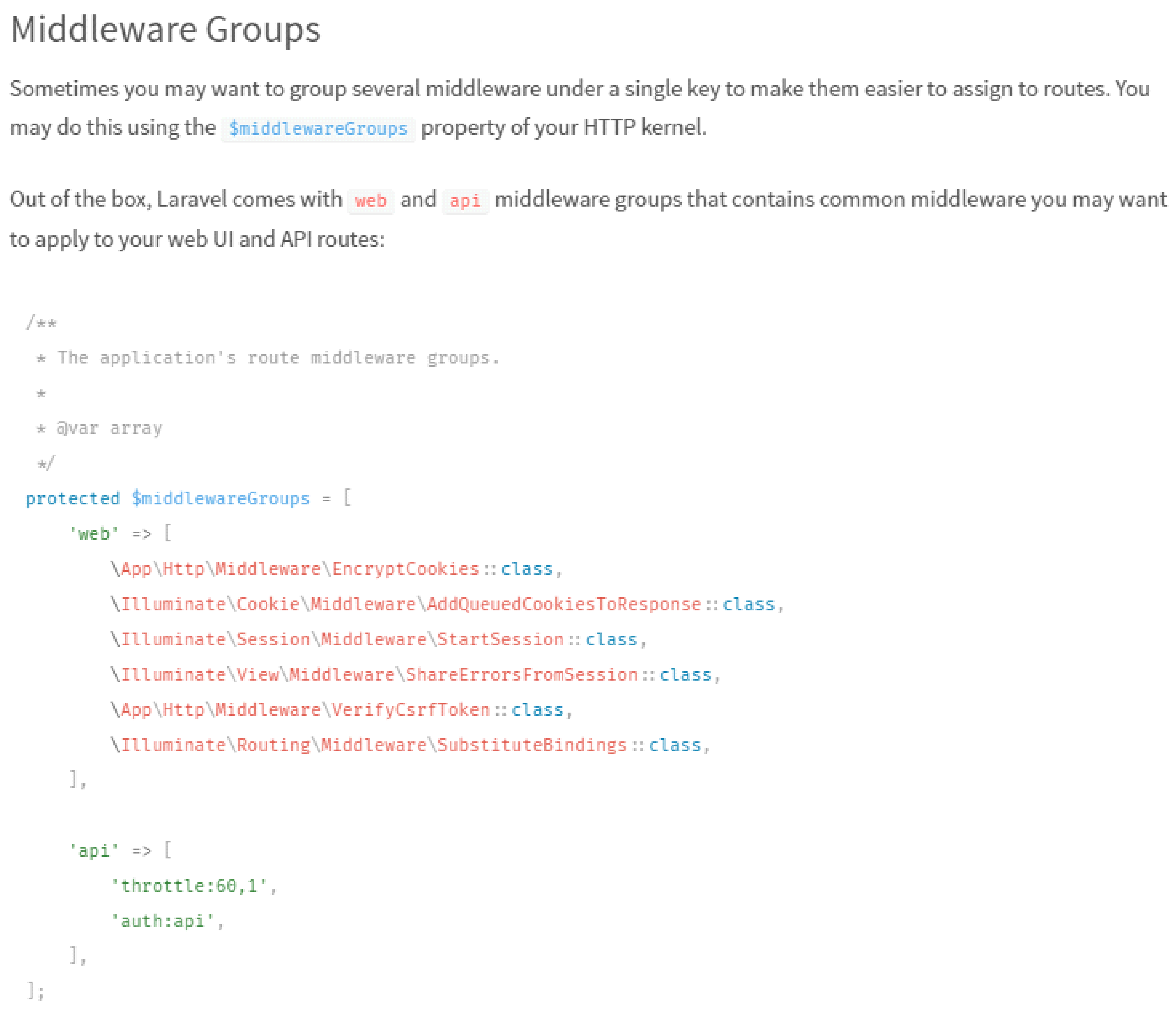
```
Route::get('/', function () {
    //
})->middleware('first', 'second');
```

When assigning middleware, you may also pass the fully qualified class name:

```
use App\Http\Middleware\CheckAge;

Route::get('admin/profile', function () {
    //
})->middleware(CheckAge::class);
```

Middleware Groups

Friday, September 15, 2017 7:47 PM



Middleware Groups

Sometimes you may want to group several middleware under a single key to make them easier to assign to routes. You may do this using the `$middlewareGroups` property of your HTTP kernel.

Out of the box, Laravel comes with `web` and `api` middleware groups that contains common middleware you may want to apply to your web UI and API routes:

```
/**
 * The application's route middleware groups.
 *
 * @var array
 */
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        'throttle:60,1',
        'auth:api',
    ],
];
```

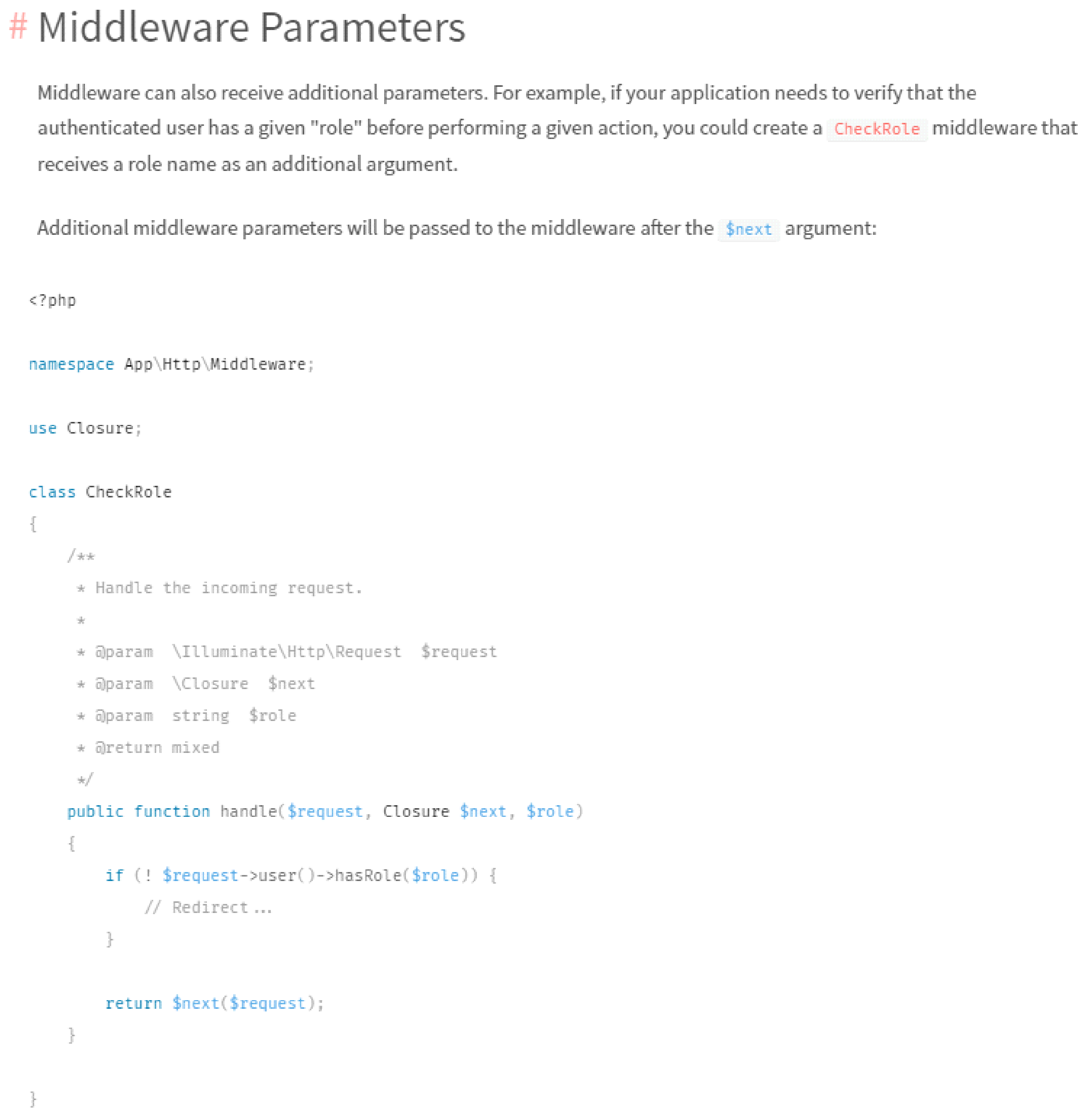
Middleware groups may be assigned to routes and controller actions using the same syntax as individual middleware. Again, middleware groups simply make it more convenient to assign many middleware to a route at once:

```
Route::get('/', function () {
    //
})->middleware('web');

Route::group(['middleware' => ['web']], function () {
    //
});
```

Out of the box, the `web` middleware group is automatically applied to your `routes/web.php` file by the `RouteServiceProvider`.

Middleware Parameters

Friday, September 15, 2017 7:51 PM



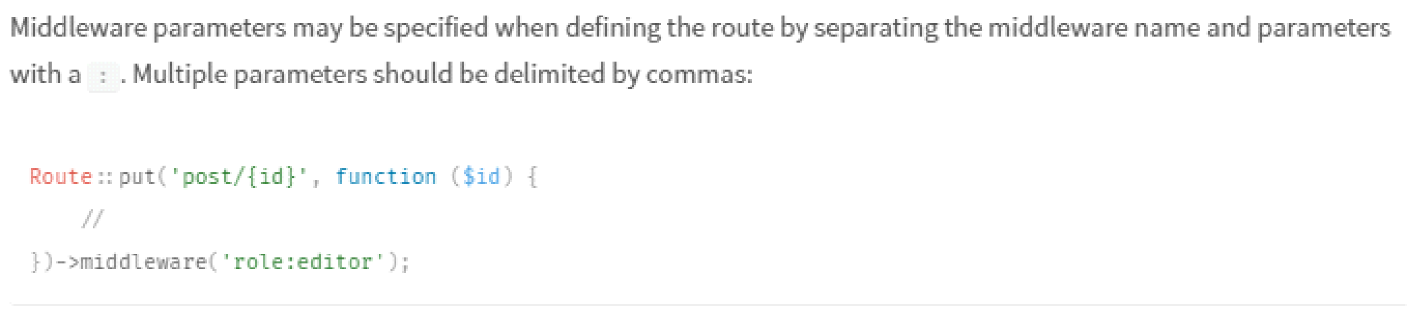
Middleware Parameters

Middleware can also receive additional parameters. For example, if your application needs to verify that the authenticated user has a given "role" before performing a given action, you could create a `CheckRole` middleware that receives a role name as an additional argument.

Additional middleware parameters will be passed to the middleware after the `$next` argument:

```php
<?php

namespace App\Http\Middleware;

use Closure;

class CheckRole
{
    /**
     * Handle the incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string  $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }

}
```

- Here hasRole() may be a function in 'User' Model class.



Middleware parameters may be specified when defining the route by separating the middleware name and parameters with a `:`. Multiple parameters should be delimited by commas:

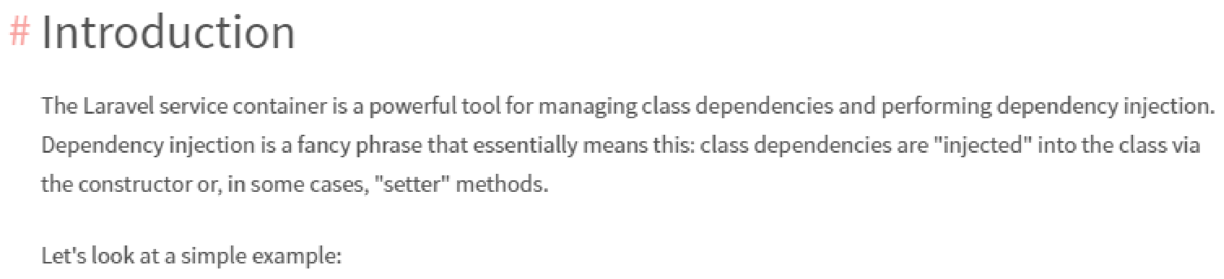
```php
Route::put('post/{id}', function ($id) {
    //
})->middleware('role:editor');
```

- Synatax - var:value

What are Service Containers?

Friday, September 15, 2017 8:25 PM

see about service containers -https://m.dotdev.co/understanding-laravel-service-container-bd488ca052800



Introduction

The Laravel service container is a powerful tool for managing class dependencies and performing dependency injection. Dependency injection is a fancy phrase that essentially means this: class dependencies are "injected" into the class via the constructor or, in some cases, "setter" methods.

Let's look at a simple example:

- Service Container is just an ordinary PHP class, but We can think of it as my “Bag of tricks”. This “Bag” is where we will place or “Bind” everything we need to run a Laravel application smoothly, from **interfaces** implementations to directories paths and so on. Yes you can place pretty much anything you want in it !… Hence, the term “Bag of tricks”.

  Now, since we have a single **Object** that contains all of our various bindings, it is very easy to retrieve them back or “resolve” them at any point in our code.

- Within a service provider, you always have access to the container via the `$this->app` property,
  We can use this container in every provider.

- First laravel look in Service container for binding when dependency injection is required, if it doesn't find, then it tries to use Reflection otherwise it throws Exception.

```
<?php

namespace App\Http\Controllers;

use App\User;
use App\Repositories\UserRepository;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * The user repository implementation.
     *
     * @var UserRepository
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }

    /**
     * Show the profile for the given user.
     *
     * @param  int  $id
     * @return Response
     */
    public function show($id)
    {
        $user = $this->users->find($id);

        return view('user.profile', ['user' => $user]);
    }
}
```

In this example, the `UserController` needs to retrieve users from a data source. So, we will **inject** a service that is able to retrieve users. In this context, our `UserRepository` most likely uses Eloquent to retrieve user information from the database. However, since the repository is injected, we are able to easily swap it out with another implementation. We are also able to easily "mock", or create a dummy implementation of the `UserRepository` when testing our application.

A deep understanding of the Laravel service container is essential to building a powerful, large application, as well as for contributing to the Laravel core itself.

Binding

Binding Basics

Almost all of your service container bindings will be registered within service providers, so most of these examples will demonstrate using the container in that context.

There is no need to bind classes into the container if they do not depend on any interfaces. The container does not need to be instructed on how to build these objects, since it can automatically resolve these objects using reflection.

- If we have to bind only when we want to get object of child when creating of parent using binding, so when we want to change something we just need to change name,

  Example - intial implementation with **FooClass**

  ```
  $this->app->bind(FooInterface::class, FooClass::class);
  ```

  Changing something and creating new file **BarClass**

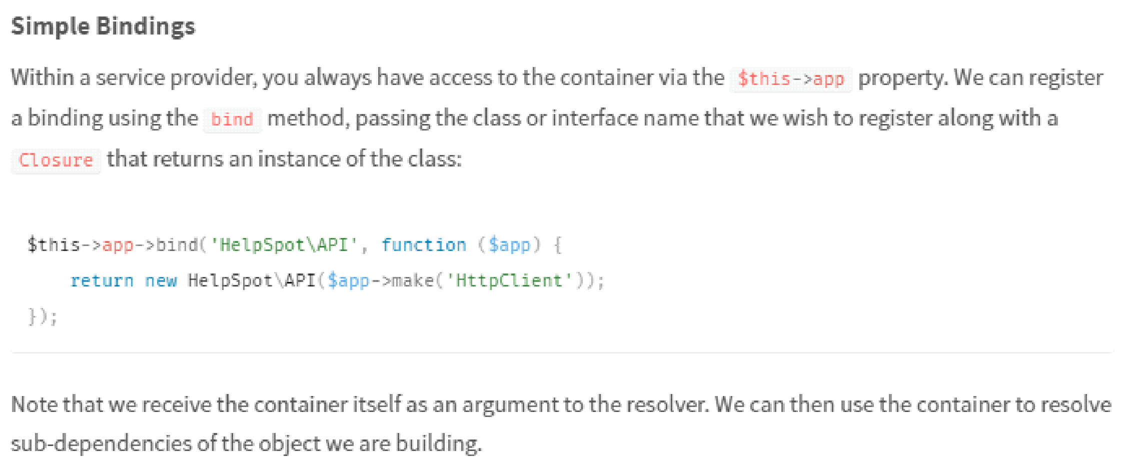
  ```
  $this->app->bind(FooInterface::class, BarClass::class);
  ```

  - Here only name one time class name need to be changed from **FooClass** to **BarClass** and other things will be same, Because we are making object of **FooInterface** as before, but getting new Object of **BarClass** now instead of **FooClass.**

NOTE- both class **FooClass** and **BarClass** must implements FooInterface.

see more interface binding -https://m.dotdev.co/understanding-laravel-service-container-bd488ca052800

Types of Bindings



Simple Bindings

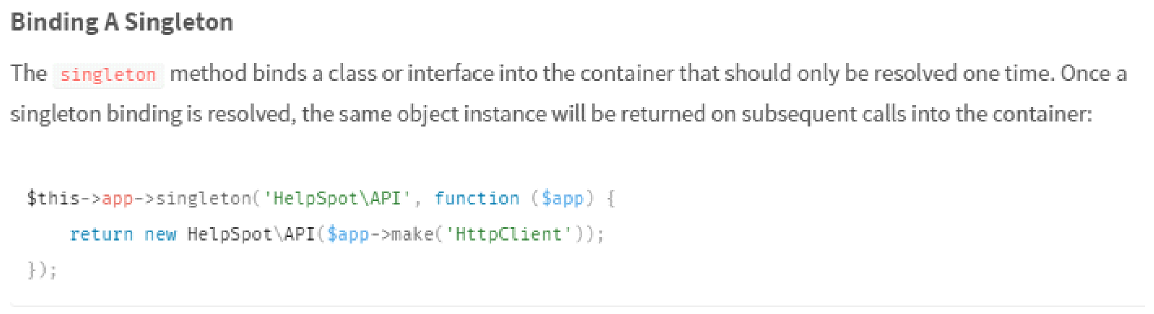
Within a service provider, you always have access to the container via the `$this->app` property. We can register a binding using the `bind` method, passing the class or interface name that we wish to register along with a `Closure` that returns an instance of the class:

```
$this->app->bind('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});
```

Note that we receive the container itself as an argument to the resolver. We can then use the container to resolve sub-dependencies of the object we are building.

NOTE- here we can easily feel this can't be done using **reflection,** as an extra parameter is needed for making **object** Of **HelpSpot\API** , so we do binding. (see laracast tutorial of service containers)

- We can resolve binding using **resolve() , make() method** //see laracasts video



Binding A Singleton

The `singleton` method binds a class or interface into the container that should only be resolved one time. Once a singleton binding is resolved, the same object instance will be returned on subsequent calls into the container:

```
$this->app->singleton('HelpSpot\API', function ($app) {
    return new HelpSpot\API($app->make('HttpClient'));
});
```

- Singleton class is useful when we want to get same object always, so any number of times we resolve this service container we get the same object.

Binding Instances

You may also bind an existing object instance into the container using the `instance` method. The given instance will always be returned on subsequent calls into the container:

```
$api = new HelpSpot\API(new HttpClient);

$this->app->instance('HelpSpot\API', $api);
```

Binding Primitives

Sometimes you may have a class that receives some injected classes, but also needs an injected primitive value such as an integer. You may easily use contextual binding to inject any value your class may need:

```
$this->app->when('App\Http\Controllers\UserController')
          ->needs('$variableName')
          ->give($value);
```

- We can bind for variables also, so when this variable is needed in **injection**(if not passed) then, then it checks for binding in dependency injection and provide with value.

Binding Interfaces To Implementations

A very powerful feature of the service container is its ability to bind an interface to a given implementation. For example, let's assume we have an `EventPusher` interface and a `RedisEventPusher` implementation. Once we have coded our `RedisEventPusher` implementation of this interface, we can register it with the service container like so:

```
$this->app->bind(
    'App\Contracts\EventPusher',
    'App\Services\RedisEventPusher'
);
```

This statement tells the container that it should inject the `RedisEventPusher` when a class needs an implementation of `EventPusher`. Now we can type-hint the `EventPusher` interface in a constructor, or any other location where dependencies are injected by the service container:

```
use App\Contracts\EventPusher;

/**
 * Create a new class instance.
 *
 * @param  EventPusher  $pusher
 * @return void
 */
public function __construct(EventPusher $pusher)
{
    $this->pusher = $pusher;
}
```

Contextual Binding

Sometimes you may have two classes that utilize the same interface, but you wish to inject different implementations into each class. For example, two controllers may depend on different implementations of the `Illuminate\Contracts\Filesystem\Filesystem` contract. Laravel provides a simple, fluent interface for defining this behavior:

```
use Illuminate\Support\Facades\Storage;
use App\Http\Controllers\PhotoController;
use App\Http\Controllers\VideoController;
use Illuminate\Contracts\Filesystem\Filesystem;

$this->app->when(PhotoController::class)
          ->needs(Filesystem::class)
          ->give(function () {
              return Storage::disk('local');
          });

$this->app->when(VideoController::class)
          ->needs(Filesystem::class)
          ->give(function () {
              return Storage::disk('s3');
          });
```

Resolving

The `make` Method

You may use the `make` method to resolve a class instance out of the container. The `make` method accepts the name of the class or interface you wish to resolve:

```
$api = $this->app->make('HelpSpot\API');
```

If you are in a location of your code that does not have access to the `$app` variable, you may use the global `resolve` helper:

```
$api = resolve('HelpSpot\API');
```

If some of your class' dependencies are not resolvable via the container, you may inject them by passing them as an associative array into the `makeWith` method:

```
$api = $this->app->makeWith('HelpSpot\API', ['id' => 1]);
```

- As **$this->app** is available only inside **ServiceProviders** , we have to use resolve() magic method if we are not inside it.
- If we also need to send some variables to container of given dependency class, we have to pass them to, for example if the class bind with **helpSpot/API** require **$id** in its construction and we want to pass it them using **makeWith()** method.

- However the most used way is **Automatic Injection** when laravel want to do dependency inject, it first see, if binding is available in **ServiceContainer** , if yes then it will be automatically create object and pass to us.

Automatic Injection

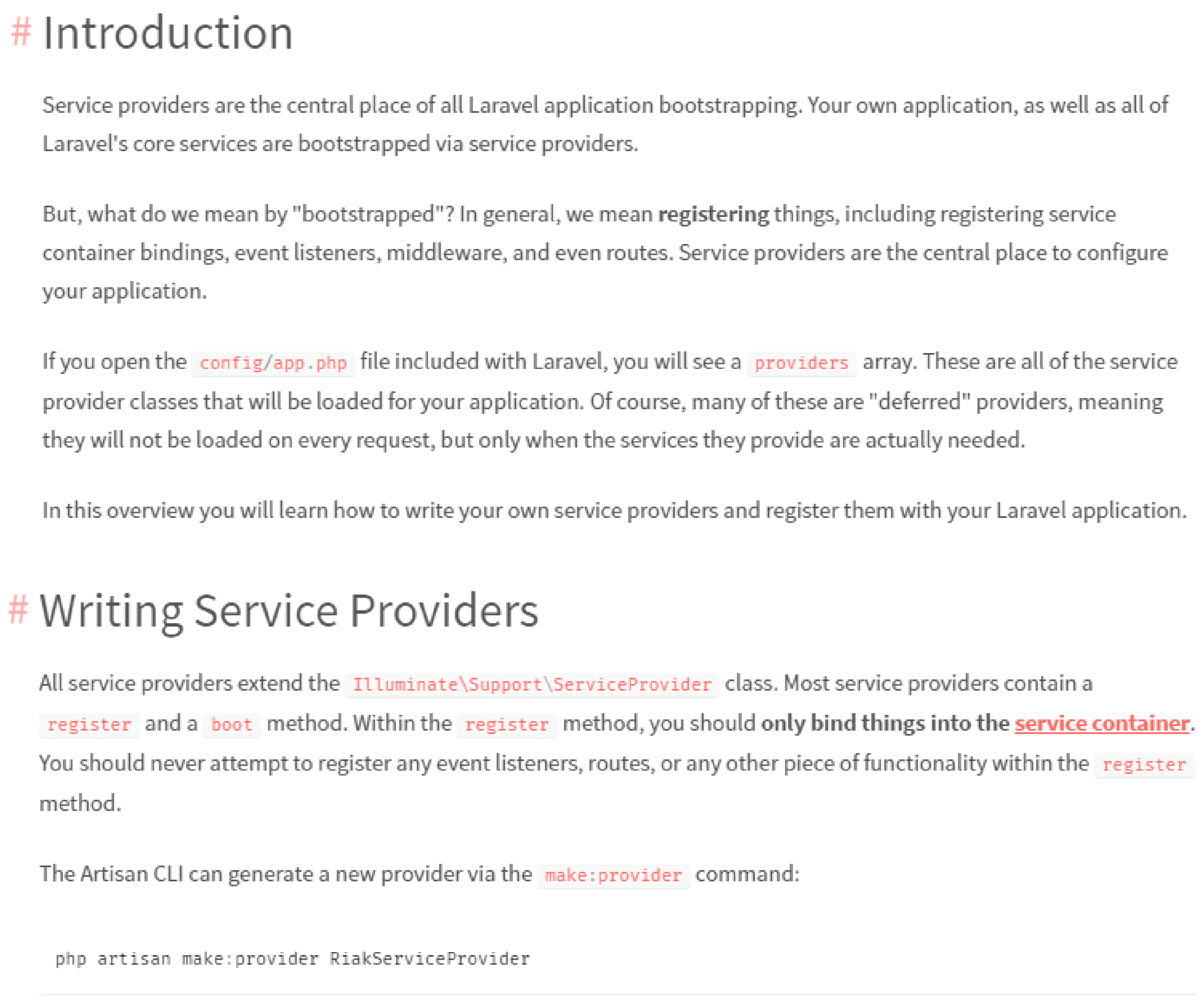
Alternatively, and importantly, you may simply "type-hint" the dependency in the constructor of a class that is resolved by the container, including controllers, event listeners, queue jobs, middleware, and more. In practice, this is how most of your objects should be resolved by the container.

For example, you may type-hint a repository defined by your application in a controller's constructor. The repository will automatically be resolved and injected into the class:

```php
<?php

namespace App\Http\Controllers;

use App\Users\Repository as UserRepository;

class UserController extends Controller
{
    /**
     * The user repository instance.
     */
    protected $users;

    /**
     * Create a new controller instance.
     *
     * @param  UserRepository  $users
     * @return void
     */
    public function __construct(UserRepository $users)
    {
        $this->users = $users;
    }


   /**
    * Show the user with the given ID.
    *
    * @param  int  $id
    * @return Response
    */
   public function show($id)
   {
       //
   }
}
```

What are Service Providers?

Friday, September 15, 2017 8:25 PM



Introduction

Service providers are the central place of all Laravel application bootstrapping. Your own application, as well as all of Laravel's core services are bootstrapped via service providers.

But, what do we mean by "bootstrapped"? In general, we mean **registering** things, including registering service container bindings, event listeners, middleware, and even routes. Service providers are the central place to configure your application.

If you open the `config/app.php` file included with Laravel, you will see a `providers` array. These are all of the service provider classes that will be loaded for your application. Of course, many of these are "deferred" providers, meaning they will not be loaded on every request, but only when the services they provide are actually needed.

In this overview you will learn how to write your own service providers and register them with your Laravel application.

Writing Service Providers

All service providers extend the `Illuminate\Support\ServiceProvider` class. Most service providers contain a `register` and a `boot` method. Within the `register` method, you should **only bind things into the service container**. You should never attempt to register any event listeners, routes, or any other piece of functionality within the `register` method.

The Artisan CLI can generate a new provider via the `make:provider` command:

```
php artisan make:provider RiakServiceProvider
```

NOTE-

First of all , all service provides get registered by calling their register() method one by one, after this boot() methods of all starts calling, so we should do things when all providers has been registered, so only binding should be done in register() method.

The Register Method

As mentioned previously, within the `register` method, you should only bind things into the service container. You should never attempt to register any event listeners, routes, or any other piece of functionality within the `register` method. Otherwise, you may accidentally use a service that is provided by a service provider which has not loaded yet.

Let's take a look at a basic service provider. Within any of your service provider methods, you always have access to the `$app` property which provides access to the service container:

```php
<?php

namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection(config('riak'));
        });
    }
}
```

This service provider only defines a `register` method, and uses that method to define an implementation of `Riak\Connection` in the service container. If you don't understand how the service container works, check out its documentation.

The Boot Method

So, what if we need to register a view composer within our service provider? This should be done within the `boot` method. **This method is called after all other service providers have been registered**, meaning you have access to all other services that have been registered by the framework:

```php
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;

class ComposerServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        view()->composer('view', function () {
            //
        });
    }
}
```

Boot Method Dependency Injection

You may type-hint dependencies for your service provider's `boot` method. The service container will automatically inject any dependencies you need:

```php
use Illuminate\Contracts\Routing\ResponseFactory;

public function boot(ResponseFactory $response)
{
    $response->macro('caps', function ($value) {
        //
    });
}
```

Registering Providers

All service providers are registered in the `config/app.php` configuration file. This file contains a `providers` array where you can list the class names of your service providers. By default, a set of Laravel core service providers are listed in this array. These providers bootstrap the core Laravel components, such as the mailer, queue, cache, and others.

To register your provider, simply add it to the array:

```
'providers' => [
    // Other Service Providers

    App\Providers\ComposerServiceProvider::class,
],
```

- If we make a new **Provider,** then we have to tell laravel to use it in **config/app.php** file which is in **providers** array.

Deferred Providers

If your provider is **only** registering bindings in the service container, you may choose to defer its registration until one of the registered bindings is actually needed. Deferring the loading of such a provider will improve the performance of your application, since it is not loaded from the filesystem on every request.

Laravel compiles and stores a list of all of the services supplied by deferred service providers, along with the name of its service provider class. Then, only when you attempt to resolve one of these services does Laravel load the service provider.

To defer the loading of a provider, set the `defer` property to `true` and define a `provides` method. The `provides` method should return the service container bindings registered by the provider:

```php
<?php

namespace App\Providers;

use Riak\Connection;
use Illuminate\Support\ServiceProvider;

class RiakServiceProvider extends ServiceProvider
{
    /**
     * Indicates if loading of the provider is deferred.
     *
     * @var bool
     */
    protected $defer = true;

    /**
     * Register the service provider.
     *
     * @return void
     */
    public function register()
    {
        $this->app->singleton(Connection::class, function ($app) {
            return new Connection($app['config']['riak']);
        });
    }

     /**
      * Get the services provided by the provider.
      *
      * @return array
      */
     public function provides()
     {
         return [Connection::class];
     }

 }
```