Laravel 4

Hand-Crafted Artisinal PHP, for web Artisans.

What is Laravel?!

“Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable, creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in the majority of web projects, such as authentication, routing, sessions, and caching.”

Awesome marketing lingo.

What is Laravel Really?!

- Laravel is an MVC framework with an emphasis on elegant syntax, modular construction, and easy routing.
- A good portion of its components are composer packages.
- It uses Symfony 2 components as its kernel.
  - HttpKernel
  - Console
  - EventDispatcher
  - Routing
  - And More - http://symfony.com/projects/laravel
- Has a strong and enthusiastic community.
- Baked in IoC container for unit testing.

Some Server Requirements

- PHP >= 5.4.0
  - Thank the gods, we need 5.3 to die.
  - I’d advise going straight to 5.5
- The PHP mcrypt library.
- You *might* need to install the php5-json package manually on ubuntu if you’re using 5.5 or higher.

Great, how do I get started?

Quickstart guide here: http://laravel.com/docs/quick

Get the installer composer package, and ensure that composer's vendor directory is in your path

```
composer global require "laravel/installer=~1.1"
```

After that, you can just do `laravel new $projectname` to create a new project directory wherever you specify.

You can also do this without the installer via composer:

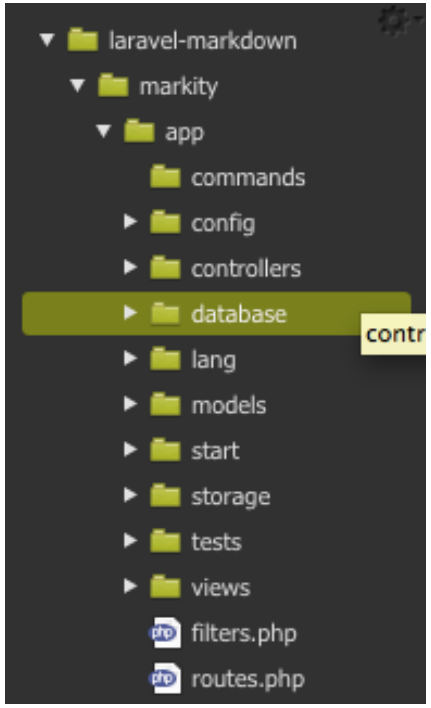
```
composer create-project laravel/laravel your-project-name --prefer-dist
```

Building a Markdown Parser

- Let's do some Laravel-by-example. We're going to build out a simple pastebin-like solution with a Markdown parser.
- We'll cover Routing, Controllers, Views, Database Migrations, and Using External composer packages.
- Slight Plug: I did all of this originally in cloud 9 IDE : https://c9.io/

Create The Project

- composer create-project laravel/laravel markity --prefer-dist
  - This generates all of the laravel core framework code in our markity directory



```
▼ laravel-markdown
  ▼ markity
    ▼ app
        commands
      ▶ config
      ▶ controllers
      ▶ database
      ▶ lang
      ▶ models
      ▶ start
      ▶ storage
      ▶ tests
      ▶ views
        filters.php
        routes.php
```

This shows the app directory. We'll want to pay special attention to the controllers, database, config, models, and views directories to get started.

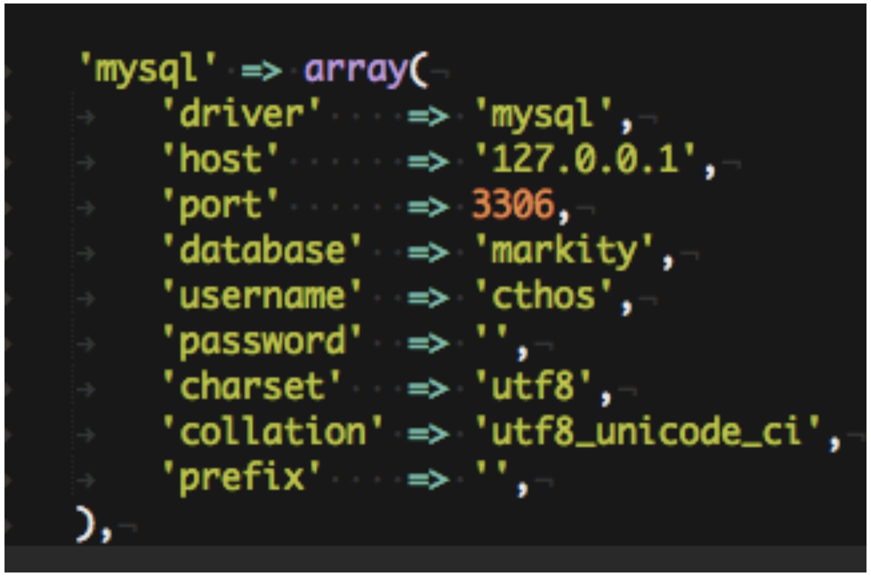
Test out your dev server

- PHP 5.4 has a nifty built in development server (php –S) which laravel can run with a simple `php artisan serve`.
- You should see a static homepage from the default laravel install. That’s how you know the sauce is working.

You have arrived.

Configure the local DB

- Application configuration is based in app/config (for generics and defaults), and app/config/{$env} (for environment specific information).
- We'll want to config the local env's database.php file to include the configuration for a locally running database, like so:



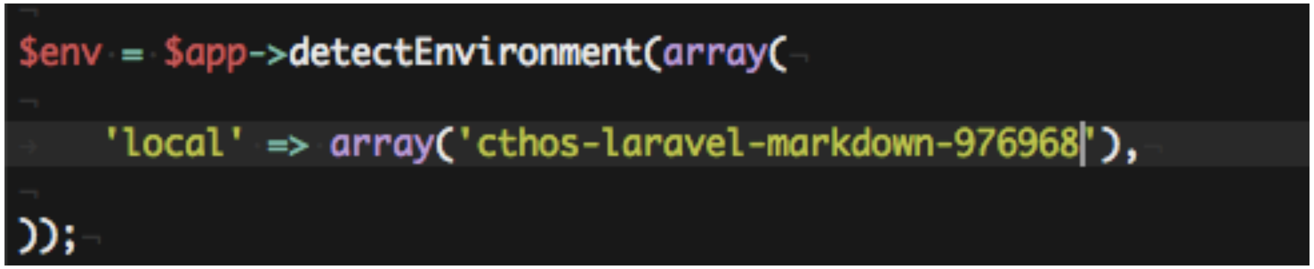
```
'mysql' => array(
        'driver'    => 'mysql',
        'host'      => '127.0.0.1',
        'port'      => 3306,
        'database'  => 'markity',
        'username'  => 'cthos',
        'password'  => '',
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
),
```

Configure the local DB (cont)

- The previous stanza is in the ‘connections’ section of the configuration array.
- This connection will only apply to the local environment. You can define global database connections in app/config/ databases.php, but I recommend keeping them in their environment directories.
- A note about config files:
    - They are all just php arrays which are being returned from the scope file and as such are assigned with “require” or “include” statements.

Configure The Local Env

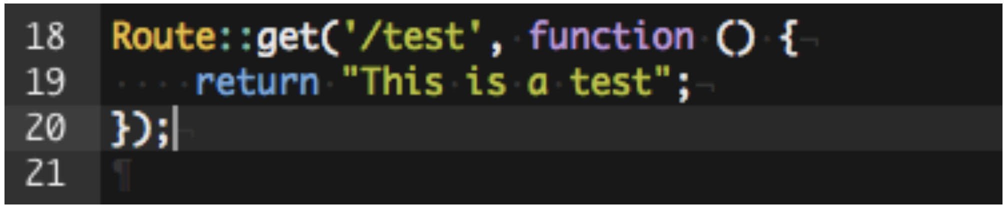
- Laravel has a magical method for determining what environment you’re on based on the server’s host name. As such you might need to configure it in app/bootstrap/start.php
  - The method is $app->detectEnvironment, and there should already be a stub for the local environment.



```
$env = $app->detectEnvironment(array(

    'local' => array('cthos-laravel-markdown-976968'),

));
```

Wire up a route!

- Routes are defined in the application's app/routes.php file.
- There are several static methods you can use to define routes.
  - ::get and ::resource being common.
  - http://laravel.com/docs/routing
- Routes can be directed to controller methods, or they can be closures. Here's a simple closure example:

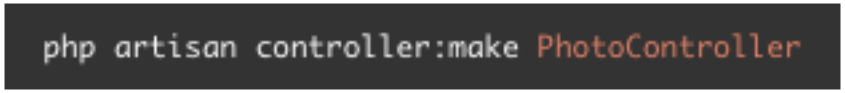


```php
Route::get('/test', function () {
    return "This is a test";
});

```

- The output from this is simply the string printed to the screen, as if you had echoed it.

Controllers: Betterness for all

- Controllers hold the logic that will be sent to the view in order to render content to a page (or provide service output).
- Laravel has the concept of RESTful controllers, as well as being able to tie those to a model for a “Resource” controller.
- There’s a handy generator in the artisan script to handle this:

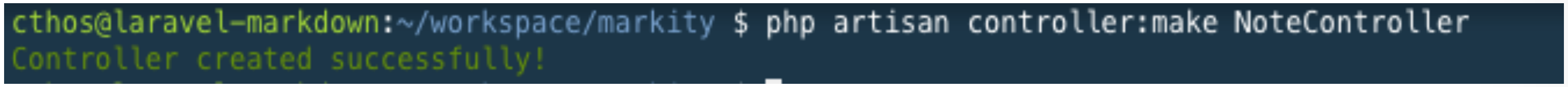


```
php artisan controller:make PhotoController
```

- It also has --only and --except flags which can exclude various methods from the controller
  - Or you can delete them yourself, doesn’t really matter.

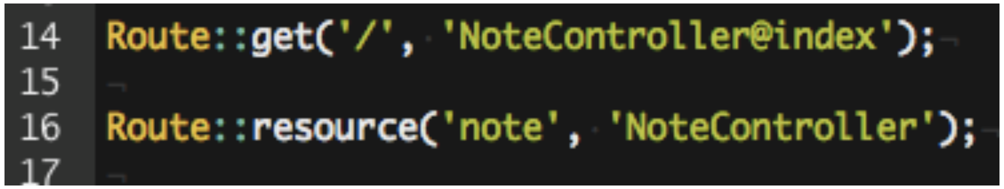
Creating the Note controller

- Create the controller:



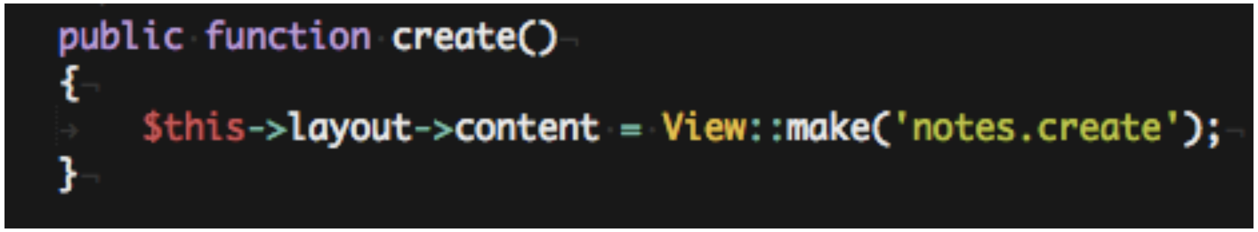
```
cthos@laravel-markdown:~/workspace/markity $ php artisan controller:make NoteController
Controller created successfully!
```

- Wire it up to a route:



```
Route::get('/', 'NoteController@index');

Route::resource('note', 'NoteController');

```

- Stub out the create method:

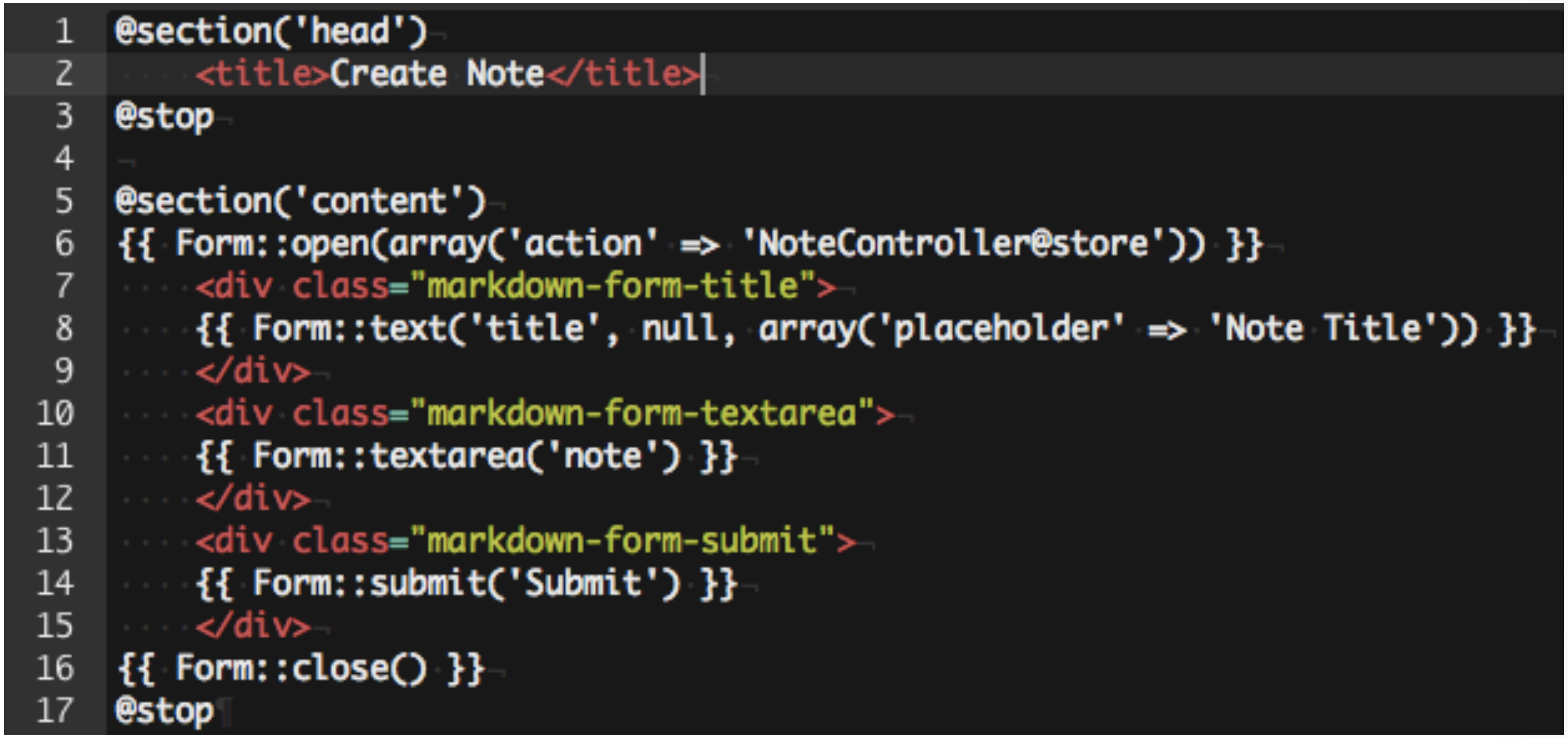


```
public function create()
{
    $this->layout->content = View::make('notes.create');
}
```

Creating a view for “create”

- Views are generally stored in the form app/views/{resourcename}/{action}.blade.php (or just .php)
  - Laravel knows whether or not to expect blade syntax based on this file extension.
- So, we’ll start off by creating the folder for notes: app/views/notes, and then inside of that make create.blade.php

Contents of the “create” view



```
@section('head')
    <title>Create Note</title>
@stop

@section('content')
{{ Form::open(array('action' => 'NoteController@store')) }}
    <div class="markdown-form-title">
    {{ Form::text('title', null, array('placeholder' => 'Note Title')) }}
    </div>
    <div class="markdown-form-textarea">
    {{ Form::textarea('note') }}
    </div>
    <div class="markdown-form-submit">
    {{ Form::submit('Submit') }}
    </div>
{{ Form::close() }}
@stop
```

Blade template engine

- http://laravel.com/docs/4.2/templates#blade-templating
- Blade is an extensible templating language for Laravel, full of magic and glory.
  - Actually it's pretty similar to most other templating languages.
- Basics
  - {{{ $variable }}} – Echo, but escape first
  - {{ $variable }} – Echo without escaping
  - @if, @elseif, @else, @endif – If control blocks
  - @foreach, @endforeach; @for, @endfor;
    - @forelse and @empty – foreach else empty == awesomeness
  - @yield – display sub-template contents.

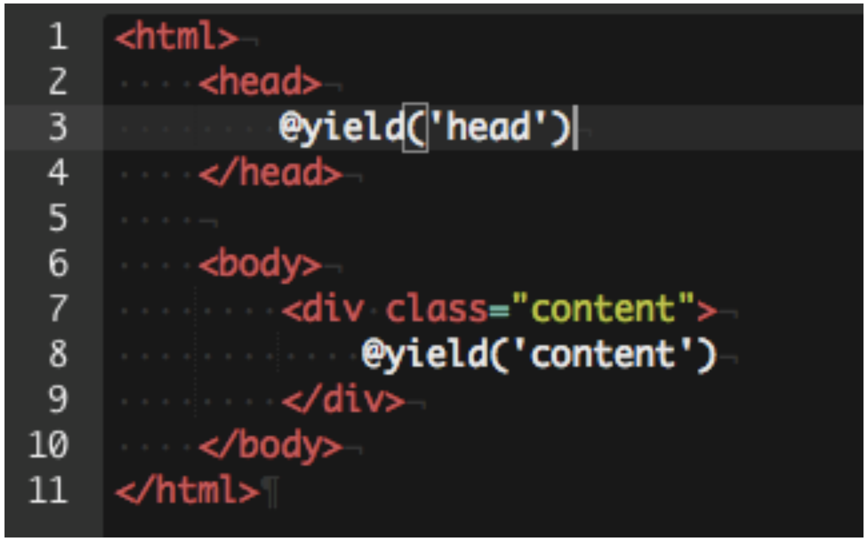
Form Helper

- Laravel has a nifty Form helper which can help generate form markup.
- Form::open opens the form and generates a CSRF token and automatically injects it into the form.
  - You can also bind its action to a controller route, which is what's going on in that array('action' => 'NoteController@store') section
- Form::text – Standard <input> element
- Form::textarea - <textarea>
- Form::submit – Submit button

Layouts

- Another nifty feature of Laravel's view system is the ability to have an over-arching layout (or sub-layout) for a given controller.
  - These can either be handled in php or via blade.
  - For this application, let's take advantage of blade's layout capabilities
- Layouts are stored in app/views/layouts.
  - I've created a base.blade.php layout for us to work off of (containing basic wrapper html).

Base.blade.php



```
<html>
    <head>
        @yield('head')
    </head>
    
    <body>
        <div class="content">
            @yield('content')
        </div>
    </body>
</html>
```

Yeah, not so much going on…

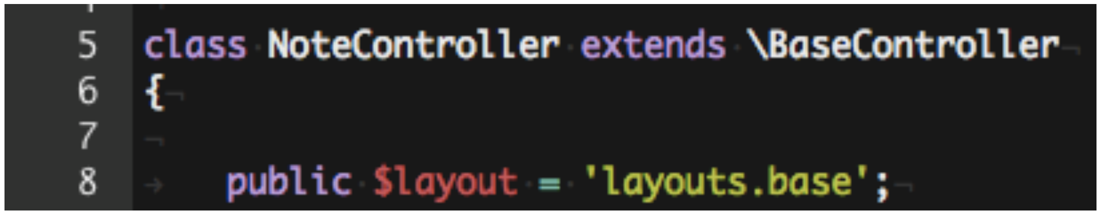
- It’s a pretty basic layout. It only contains the basic tags we need to have a valid HTML base, and then provides two @yield blocks for which our sub templates can take over for.
- The @yield(‘head’) section is designed for sub templates to be able to pass along extra header tags as necessary
  - My example is a bit of bad form, since the title should probably be handled by a controller variable, but eh, tomato tomato.
- The @yield(‘content’) section is for the actual guts of the view.

Getting content into @yield

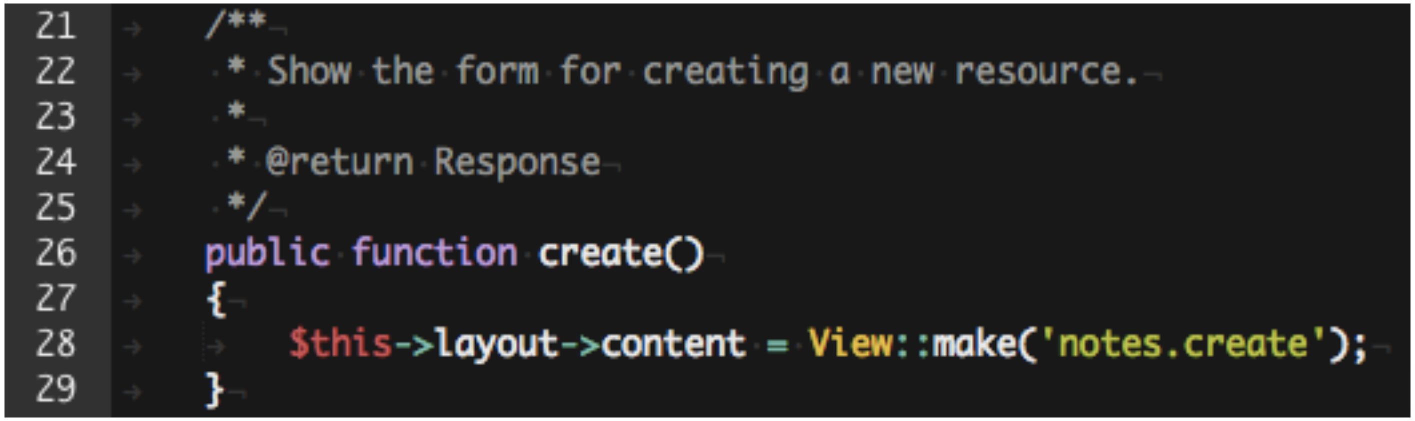
- So! How the heck do we get the content to show up in the @yield sections?
- This is the job for @section and @stop.
- The @section('name') syntax defines the given section you're wanting to operate on.
- The @stop directive closes out that section block.
- Defining sections that aren't in the layout will cause those sections to not be rendered at all.
  - AKA if there's no @yield block for a corresponding @section block, it's not going to show up.

Great, how do I render that?

- So, there are a couple of things we need to deal with in the controller.



```php
class NoteController extends \BaseController
{

    public $layout = 'layouts.base';
```



```php
    /**
     * Show the form for creating a new resource.
     *
     * @return Response
     */
    public function create()
    {
        $this->layout->content = View::make('notes.create');
    }
```

Whoh, $this->layout?

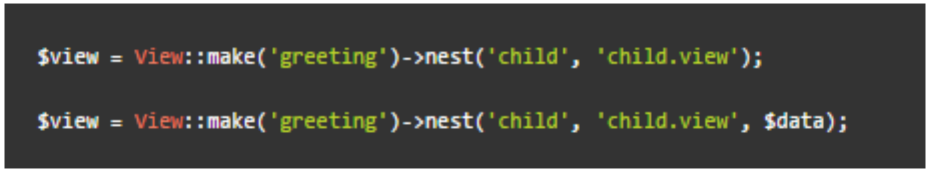
- Okay, that's part of the view modeling fanciness.
- It allows you to define what the layout is going to be.
- You can then nest content using $this->layout->content
  - This tells the renderer to load the layout first, and then you can nest a sub template within it.
- If there's no $layout variable, you can just return View::make

View::make

- This is a helper method that allows you to bind parameters to the view, so that you can render them to the page.
- To do this, you can pass an array of variables as the second parameter.
  - It returns an object with various methods for doing this as well.
    - ->with() takes 2 parameters, the variable name and its contents.
    - ->with{$param} uses magic methods to unpack the variable name from the method, and only takes the data as a param.
  - Best practice is to not send a model directly, but instead to send a static representation of it to the view, that way it can't be inadvertently saved or whatnot.

Nesting Views

- In addition to the layout code demonstrated earlier, views can also be nested within each other.
  - This is done with the aptly-named "nest" method

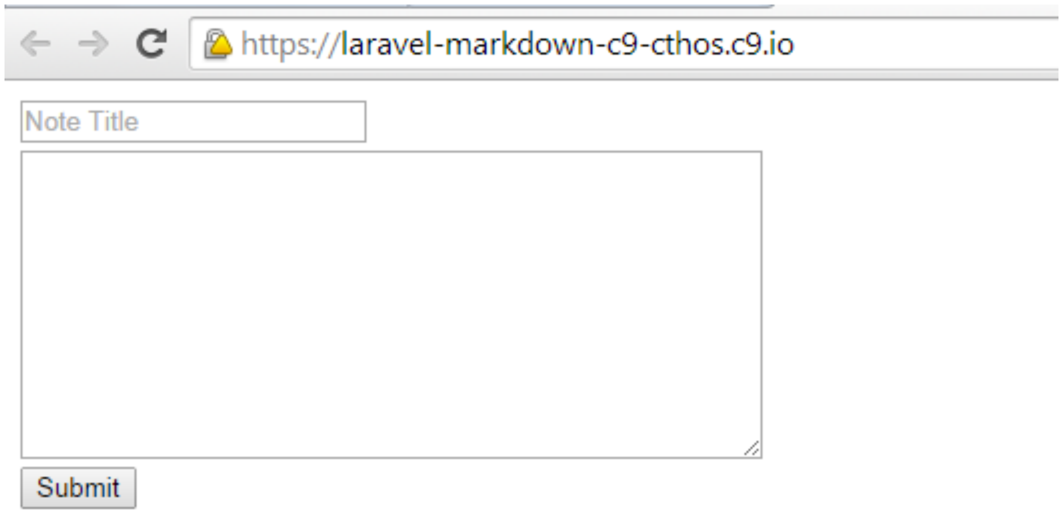


```
$view = View::make('greeting')->nest('child', 'child.view');

$view = View::make('greeting')->nest('child', 'child.view', $data);
```

View Composers

- View composers are a nifty way to bind data to a view every time it is rendered.
  - You might do this on say a template that renders a user bar to render the user's username and details when the view is rendered.
  - Additionally, you can bind the same composer to multiple views at the same time.
- Composers have no convention to where they can be stored. So long as your composer.json's autoloader rules can find them, they'll be loaded.
- View *creators* are essentially the same thing, but are loaded immediately when the view is instantiated.

Alright, let's check out the page

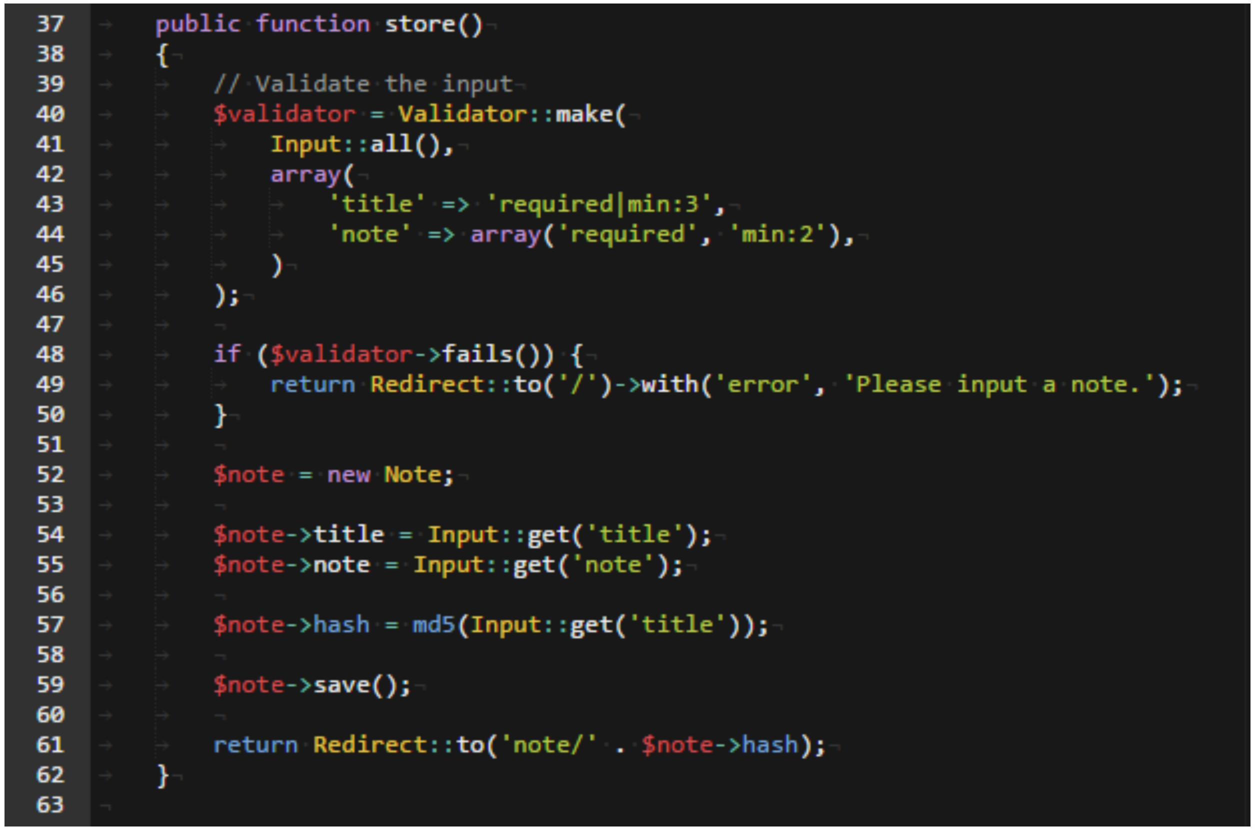

https://laravel-markdown-c9-cthos.c9.io

Note Title

Submit

Dealing with note submission

- We need to create the ->store method on the NoteController



```php
	public function store()
	{
		// Validate the input
		$validator = Validator::make(
			Input::all(),
			array(
				'title' => 'required|min:3',
				'note' => array('required', 'min:2'),
			)
		);

		if ($validator->fails()) {
			return Redirect::to('/')->with('error', 'Please input a note.');
		}

		$note = new Note;

		$note->title = Input::get('title');
		$note->note = Input::get('note');

		$note->hash = md5(Input::get('title'));

		$note->save();

		return Redirect::to('note/' . $note->hash);
	}

```

Input validation

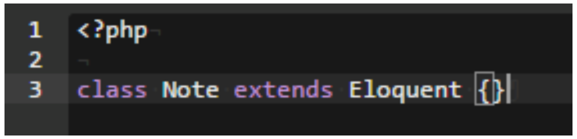
- Validation is typically handled by the Validator class.
  - Validator::make takes 2 parameters, the first being the data to validate, and the second being the rule sets
    - Input::all() will generally get you what you want out of the post/get data.
    - The rules structure and list can be found here: http://laravel.com/docs/4.2/validation
  - Rules can be either pipe-delimited or an array of different rules.
    - This is demonstrated in the note's create controls.

Redirects and Flash messages

- The Redirect class handles returning various types of redirects.
- It also has the ability to pass along flash messages.
  - These are stored in the session, and thus need to have a valid session store (defaults to phpsession).
- You can pass some other parameters to ::to as well, to make the redirect status be a 302 instead of the standard 301
  - This is only documented in the code.
- http://laravel.com/docs/4.2/responses#redirects

Note model

- Now it's time to create something that can store the note data in the database.
- We need to make a file in app/models named Note.php
- It contains:



```php
<?php

class Note extends Eloquent {}
```

- Yep. The Eloquent controller uses the ActiveRecord pattern to figure out what parameters it has.

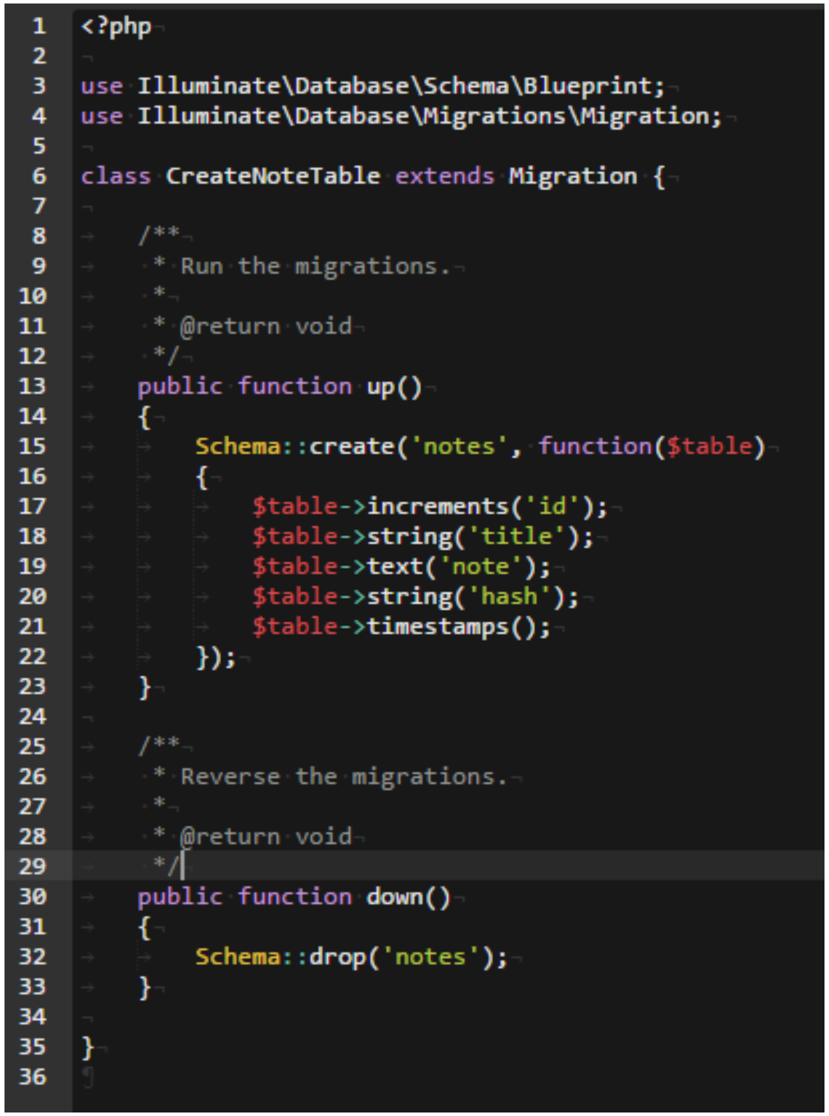
Database Migrations

- Database migrations cover the situation where you need to alter the database at some point in time.
- They live in app/database/migrations, and can be created with an artisan command
  - php artisan migrate:make {name of migration}
  - You may need to php artisan migrate:install to create the database tables where the current position of the database is stored
    - This is how you can rollback database changes

Note Database Migration

- So we'll create our stub with the following:
  - php artisan migration:make create_note_table
  - That will create a file prefixed with the date and time in the migrations folder.
- A class is created automatically, with stubs for the up and down methods
  - These determine what happens during migrate and migrate:rollback

Note migration



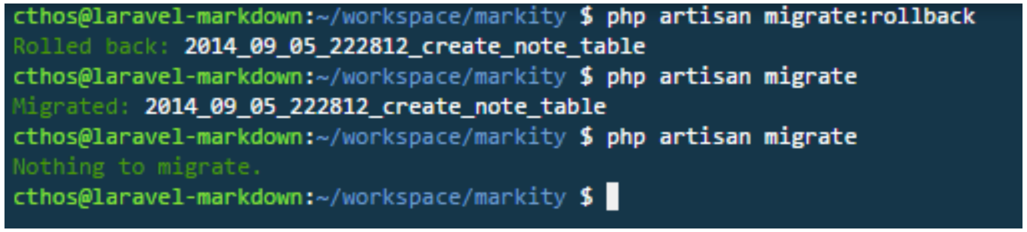
```php
<?php

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

class CreateNoteTable extends Migration {

	/**
	 * Run the migrations.
	 *
	 * @return void
	 */
	public function up()
	{
		Schema::create('notes', function($table)
		{
			$table->increments('id');
			$table->string('title');
			$table->text('note');
			$table->string('hash');
			$table->timestamps();
		});
	}

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

}
```

Schema Class

- This is a helper class which creates database-agnostic code to generate the appropriate tables.
- Schema::create takes an anonymous function (or any other callable, really) and passes the $table class to it. On that, you may determine the appropriate columns.
- Schema::drop drops the named table. Typically that is all your down method will contain, unless you're doing alters.
- Table alters are handled by Schema::table
  - You can drop columns, rename them, change the definitions, etc.
- http://laravel.com/docs/4.2/schema

Note migration finished

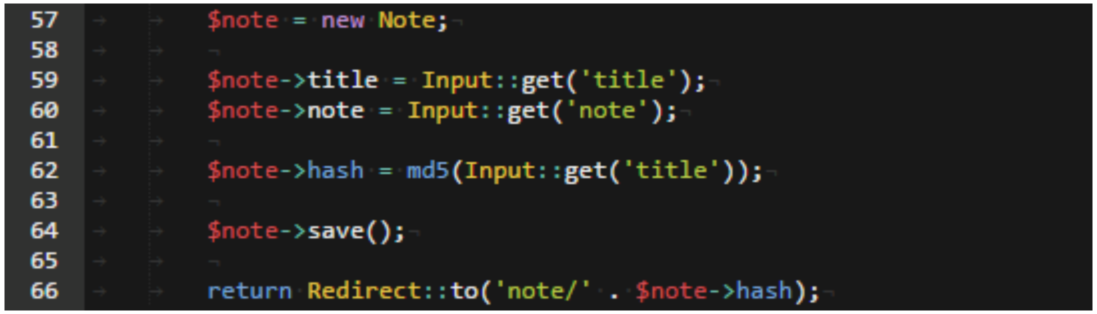
- Note that I do a rollback first, because I'd already run a migration for this previously.



```
cthos@laravel-markdown:~/workspace/markity $ php artisan migrate:rollback
Rolled back: 2014_09_05_222812_create_note_table
cthos@laravel-markdown:~/workspace/markity $ php artisan migrate
Migrated: 2014_09_05_222812_create_note_table
cthos@laravel-markdown:~/workspace/markity $ php artisan migrate
Nothing to migrate.
cthos@laravel-markdown:~/workspace/markity $ 
```

Create action reexamined

- Let’s look at the note model creation bit again:

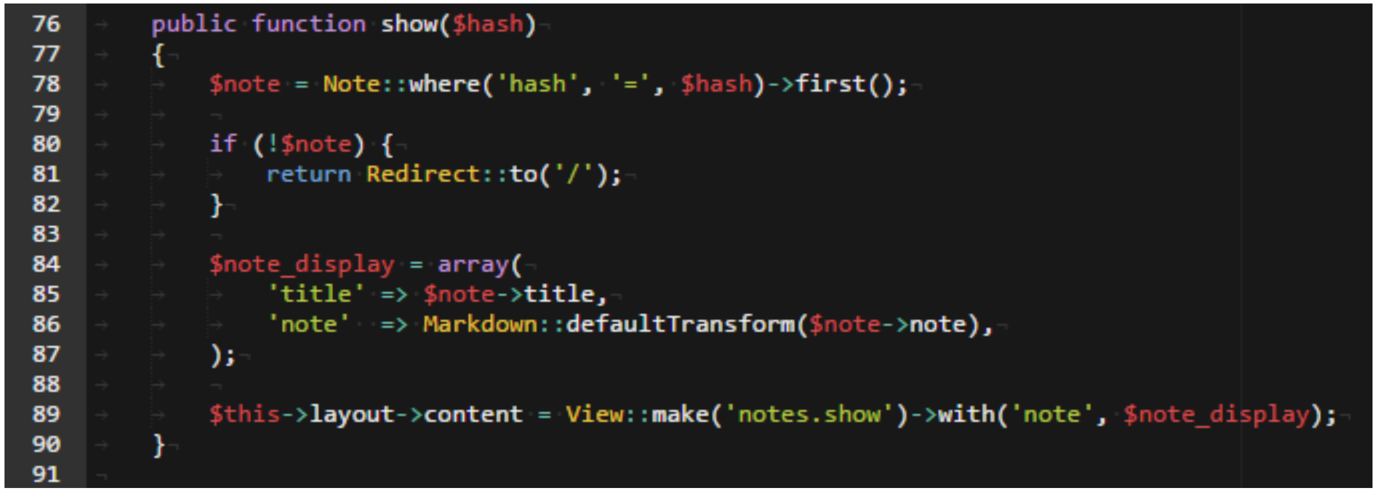


```
		$note = new Note;

		$note->title = Input::get('title');
		$note->note = Input::get('note');

		$note->hash = md5(Input::get('title'));

		$note->save();

		return Redirect::to('note/' . $note->hash);
```

- We create a new Note model by instantiating its class.
- After that we can attach parameters to it directly.
  - It also supports mass assignment, but you have to do some extra hoops to make it secure: http://laravel.com/docs/4.2/eloquent#mass-assignment
- Finally calling the ->save() method persists the model to the database. (Update if the primary key is assigned, otherwise Insert)

Creating the View Method

- The ->show method takes a single parameter (usually the primary key) and can display data based on that.

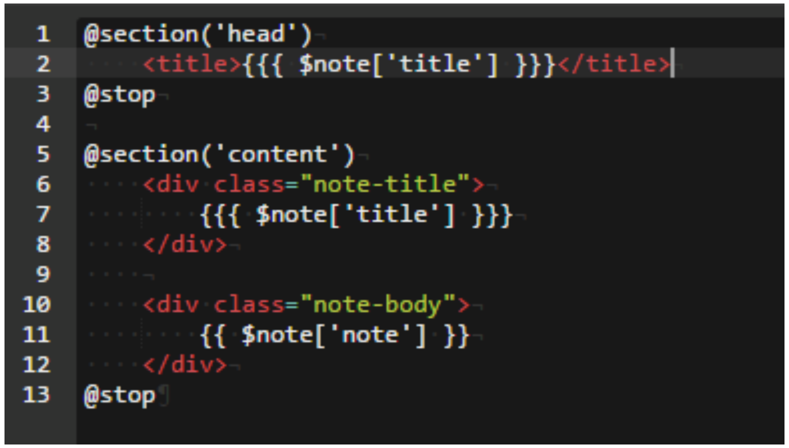


```
	public function show($hash)
	{
		$note = Note::where('hash', '=', $hash)->first();

		if (!$note) {
			return Redirect::to('/');
		}

		$note_display = array(
			'title'  => $note->title,
			'note'   => Markdown::defaultTransform($note->note),
		);

		$this->layout->content = View::make('notes.show')->with('note', $note_display);
	}

```

Examining the Show Method

- The first important bit is using the Note model itself to make a query against the database.
  - Most of the methods return a Fluent Query Builder object and can be used normally.
    - http://laravel.com/docs/4.2/queries
- Second bit is binding the note parameters to an array which is then passed to the view.
- Finally, I’m using a composer package to parse the Markdown into HTML. This is particularly important because it requires something special out of the view.

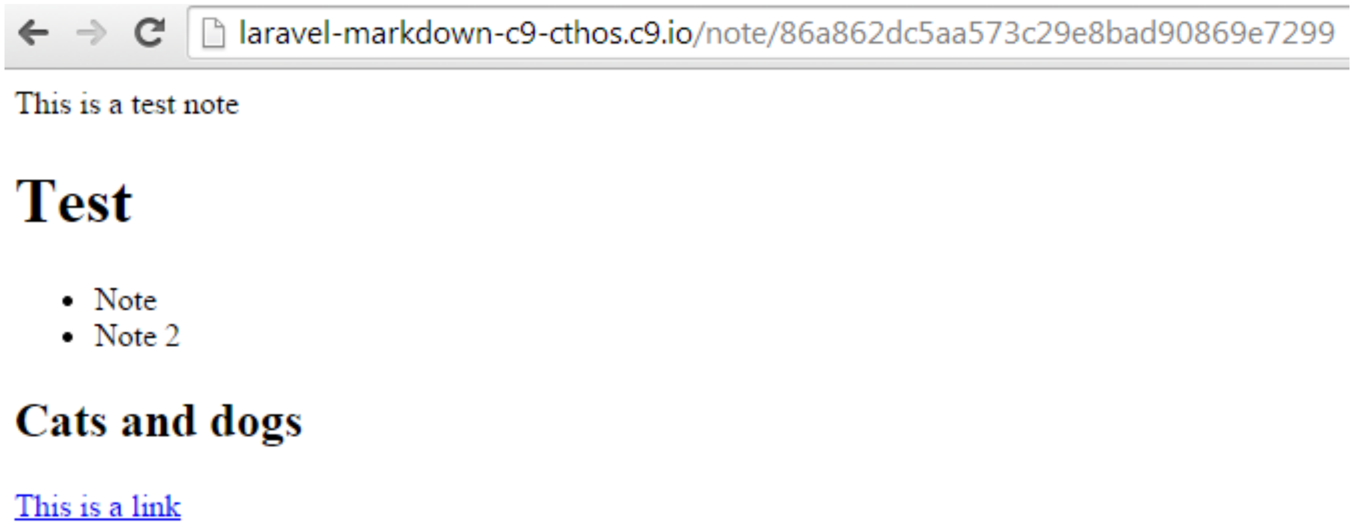
The Show view



```
@section('head')
    <title>{{{ $note['title'] }}}</title>
@stop

@section('content')
    <div class="note-title">
        {{{ $note['title'] }}}
    </div>

    <div class="note-body">
        {{ $note['note'] }}
    </div>
@stop
```

The important bit to note here is that the $note['note'] renderer has to be surrounded in {{ }} so that it won't be escaped before it is rendered. This is because the markdown processor is handling the conversion to HTML on its own

A view of a rendered note


laravel-markdown-c9-cthos.c9.io/note/86a862dc5aa573c29e8bad90869e7299

This is a test note

Test

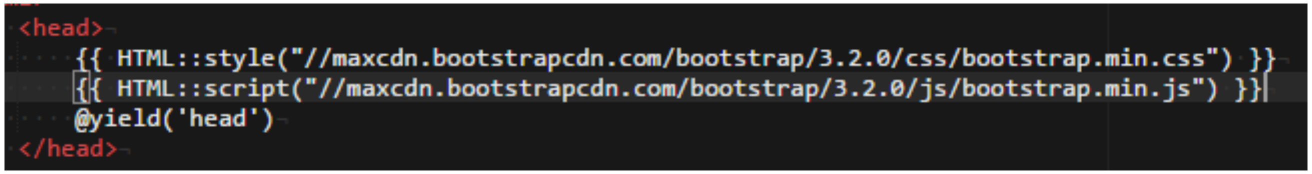
- Note
- Note 2

Cats and dogs

This is a link

Basics are done!

- But this looks a little plain. Let’s handle adding some assets and styling.



```
<head>
    {{ HTML::style("//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css") }}
    {{ HTML::script("//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js") }}
    @yield('head')
</head>
```

- Adding bootstrap to the header, loading it from a CDN is a good start.
  - Using the HTML class to render the proper html is handy, and a good idea most of the time.

Fin!

- Questions?

- Contacting me:
  - @cthos
  - http://alextheward.com
  - daginus@gmail.com