Introduction to the Laravel Framework

Fall 2018 – Introduction to Full-Stack Development

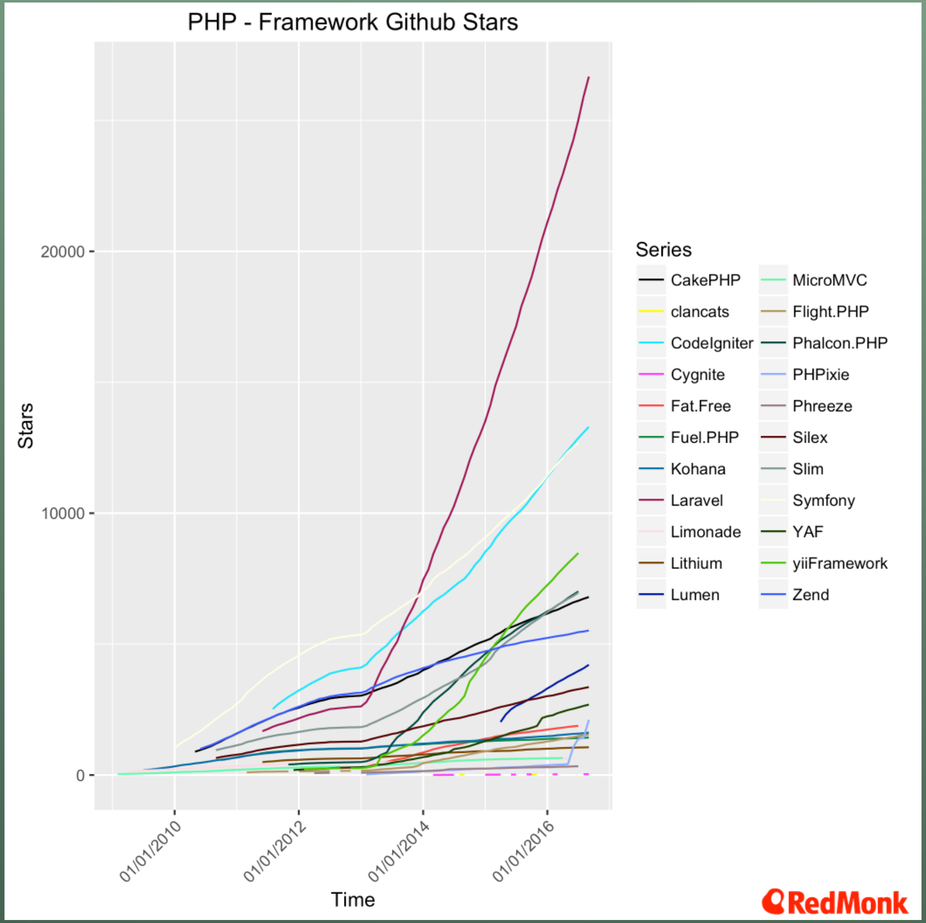
Instructor: Matthew Fritter

matt@frit.me

Overview: The Laravel Framework

- Laravel is an open-source PHP framework released under MIT license
- Developed by Taylor Otwell as a more powerful alternative to existing PHP frameworks like CodeIgniter, initially released in 2011
- Heavily based on the Symfony PHP framework – you may occasionally see Symfony error pages when exceptions occur related to the Symfony base code
- Initially supported PHP 5, but switched to PHP 7 in August of 2017
- We are using Laravel 5.7, which was actually released this month

Why Laravel? Why PHP?

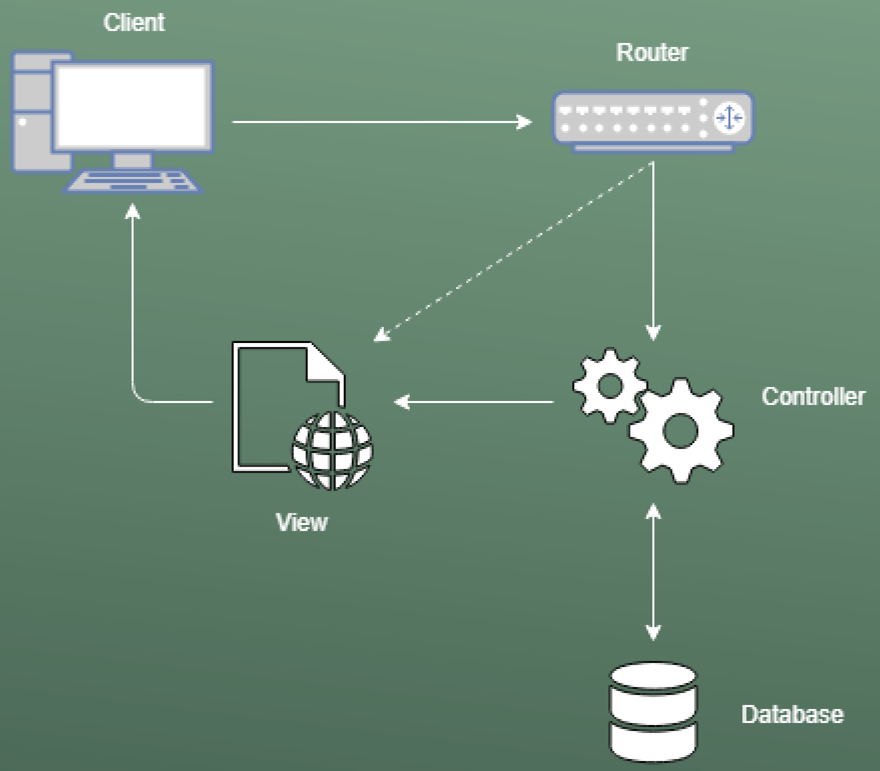


- PHP remains one of the most popular languages for web development: W3Techs suggest 79% of websites currently use PHP (Sept 2018), down from 80% (Jan 2018), but up from 72% (Jan 2010)
- This situates PHP as the most widely used web programming language today
- As far as PHP frameworks go, GitHub statistics suggest the Laravel is far and away the most popular PHP framework, with Symfony and CodeIgniter coming distant second and third
- In summary: Laravel is a modern, updated, and extremely popular framework for the most widely used language in web development – a natural choice for a framework to learn

Is Laravel an MVC Framework?

- The Model-View-Controller pattern is commonly used in web development – but is Laravel an example of this pattern?
- Earlier versions of Laravel had greater MVC influence – they had specific folders for models, views, and controllers
- More recent versions of Laravel have arguably expanded past the MVC model – Laravel can support a far more complicated system than MVC, although many users will never use these features
- Opinions are mixed and inconclusive. Whether Laravel is an MVC or not is largely based on how you choose to use it

★ Basic Laravel Architecture



- The core structure of the Laravel architecture:
  - The Router (Route files)
  - The Controller(s)
  - Database connection (Query Builder or Eloquent ORM)
  - Views (Blade Templates)

Routes in Laravel

★ Routing in Laravel

- The Router is responsible for handling incoming requests (GET, POST, PUT, etc) and determining the appropriate course of action based on the route files
- The route files allow us to define a URI, and define one or more request types, and then specify what we want to happen when that specific URI receives those request types
- For example, we might define a GET route for "about" that simply returns a view. When the client visits http://<website>.com/about, this route is triggered, and the view is sent back to the client
- We might also define a POST route for a form submission URI on our site. This POST route might pass the posted form data along to a controller, which in turn could manipulate submitted data

★ Simple Get Routes in Laravel

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

- This is the simplest form of routes – it simply takes a URI and a Closure (an anonymous function)
- When the user visits the prescribed URI (“hello”, in this case), the Closure is executed
- The returned value of the Closure is passed to the client – in this case, we’d simply see a page that said “Hello World”. Note: This will be wrapped in a basic HTML document
- While it might be tempting to write simple Closures like this and push most of our logic into the routes file itself, this is messy and definitely ***<u>not</u>*** what the router was designed for
- That being said, simple closure routes like this can be used to quickly build APIs, particularly where the only actions are single DB queries followed by a conversion from associative arrays to JSON format

★ Routing to Controllers

```
Route::get(‘/products’, ‘ProductController@index’);
```

- Rather than using a Closure, we’ll usually want to define a function from a controller – the Router will call that function, and the return of that function will be presented to the client
- In this case, ‘ProductController’ is the name of the controller file itself, as well as the class name declaration within the file (we’ll get to this later)
- The ‘index’ is a defined function within the ProductController class – in this case, it might query a database of products and return all the products currently stocked
- POST routes are handled identically:

```
Route::post(‘/submit’, ‘UserController@newUser’);
```

Route Parameters

- Parts of the URI for the route can be passed along as parameters – this can often be used as an alternative to a GET request query string. Consider the two following URIs:
  - `www.<website>.com/articles?id=16839&topic=news`
  - `www.<website>.com/articles/news/16839`
- You can define required URI parameters by surrounding the route URI segment with { }, or { ?} for optional parameters
- Parameters can be passed to a Closure, or to a Controller function.
- Below is an example of a route using a parameter and a Closure:

```
Route::get(‘articles/{id}', function ($id) {
    if($id == 123){
        return $id;
    }else{
        return “Article doesn’t exist”;
    }
});
```

View and Redirect Routes

- Sometimes, we might not need to use any back-end functionality – we might just be returning a static web page
- In this case, we can use a View route, which will simply return a View:

```
Route::view('/about', 'about');
```

- In this case, 'about' is the name of the view
- Another simple route structure is the Redirect structure, which allows us to easily redirect on URI to another:

```
Route::redirect('/here', '/there', 301);
```

- The first parameter is the URI that is being redirected, and the second URI is the destination of the redirect. The third parameter is the HTTP code to be used for the redirection, i.e. 301 for a permanent redirect, or 307 for a temporary redirect

★ The Route Files

- When you look at your /routes directory, you will notice there are several different route files – we are primarily concerned with the web and API route files
- The files apply different middleware to the routes the contain
- The web route file is designed for the main web interface, and includes the middleware to support session states and CSRF (Cross-Site Request Forgery) protection
  - An important note about CSRF protection – forms and inputs submitted via POST ***must*** include a generated CSRF token, or the server will reject them – we'll discuss this later
- The API route file is stateless and don't feature CSRF protection. In addition, all route URIs in the API route file are automatically prefixed with /api/

Other Router Functionality

- For the most part, we're going to be dealing with GET and POST routes with the default web and API middleware groups
- It is also possible to apply middleware on a route-by-route basis within the Route file – Laravel includes a variety of potentially useful middleware, such as rate limiting and authentication
- `Route::fallback` may also be useful, as it catches any requests that don't match any defined routes – good for custom or non-conventional 404 behaviour
- Named routes – for when you're lazy (or just efficient) and want to be able to quickly reference an existing route elsewhere in Laravel
- Full documentation is available at https://laravel.com/docs/5.7/routing
  - You are encouraged to read it and experiment

Controllers in Laravel

★ Controllers

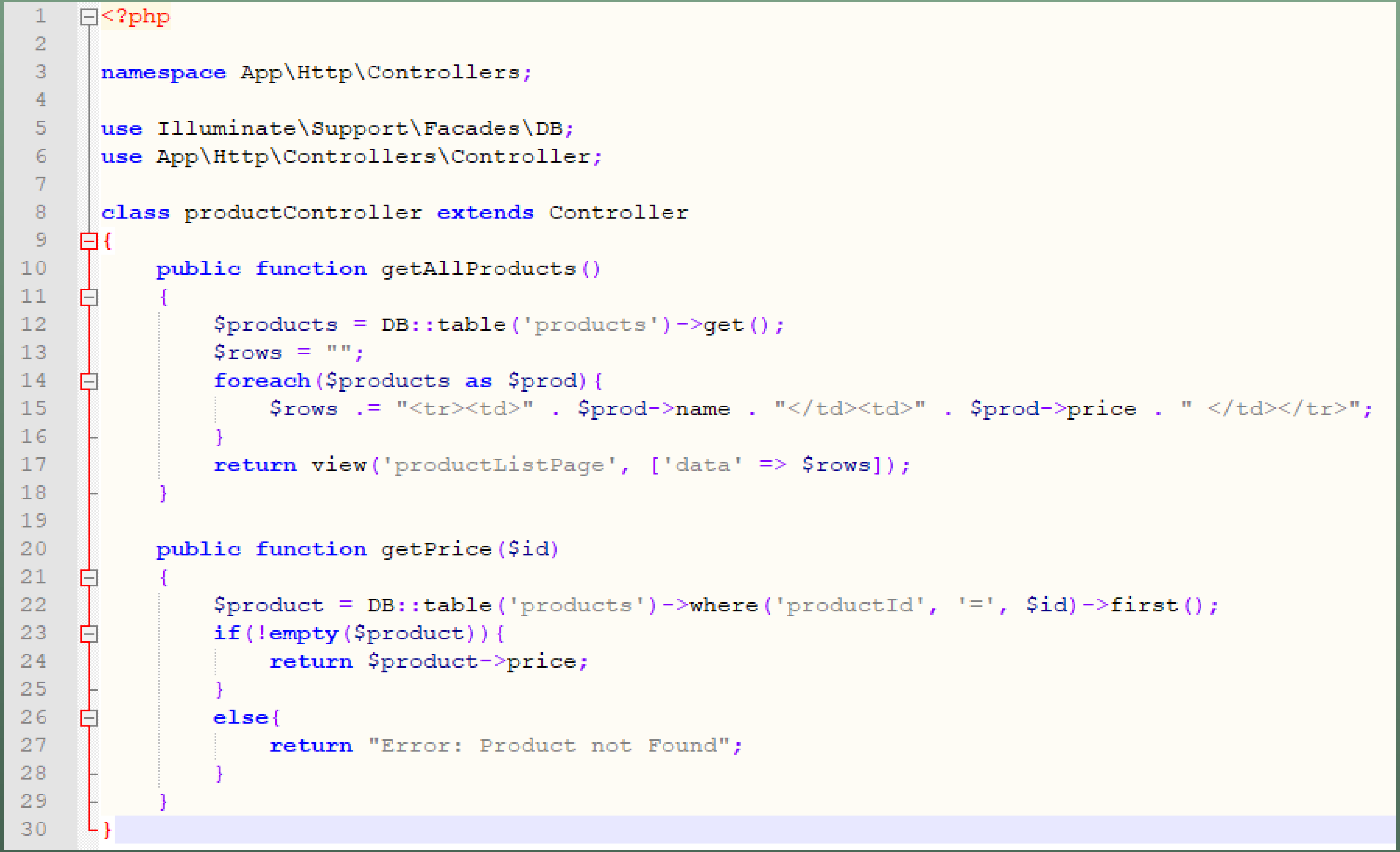
- Controllers are where the majority of your back-end logic is going to go for your website
- Each controller is a named class that extends Controller, and contains some set of functions that perform logic
- These functions may accept input parameters (such as those produced by parameterized GET routes) or a request parameter (such as those produced by a POST request)
- Controller functions may return a view with additional data, plain text output, or no return at all, similar to routes with closures
- In a typical Laravel architecture, the controller serves to handle request data, query/update the database, and return the results of the request via a view

★ Setting up a Controller

- Your controllers are available in the folder `/app/Http/Controllers`, relative to your Laravel installation folder
- The controller will have a defined `namespace`, usually `App\Http\Controllers` by default. This allows the router to correctly find and identify controllers specified in a route
- It may also `use` certain other classes and functions provided by Laravel, for example the DB façade, which allows you to use Laravel's Query Builder
- You can either manually create a new controller, or you can use the Artisan console utility which can automatically generate new controllers for you:

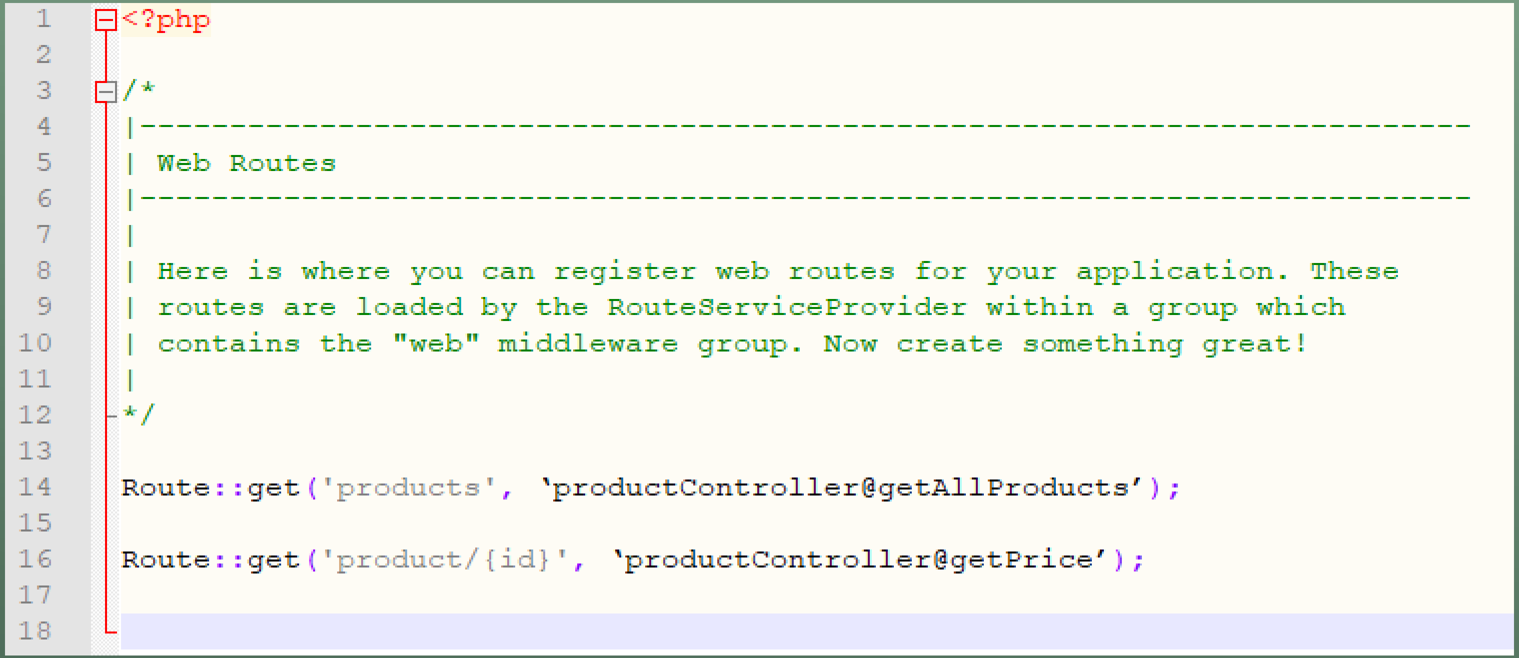
```
php artisan make:controller <controllerName>
```

★ A Basic Controller – Controller File



```php
<?php

namespace App\Http\Controllers;

use Illuminate\Support\Facades\DB;
use App\Http\Controllers\Controller;

class productController extends Controller
{
    public function getAllProducts()
    {
        $products = DB::table('products')->get();
        $rows = "";
        foreach($products as $prod){
            $rows .= "<tr><td>" . $prod->name . "</td><td>" . $prod->price . " </td></tr>";
        }
        return view('productListPage', ['data' => $rows]);
    }

    public function getPrice($id)
    {
        $product = DB::table('products')->where('productId', '=', $id)->first();
        if(!empty($product)){
            return $product->price;
        }
        else{
            return "Error: Product not Found";
        }
    }
}
```

★ A Basic Controller – Routes File



```php
<?php

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

Route::get('products', ‘productController@getAllProducts’);

Route::get('product/{id}', ‘productController@getPrice’);

```

Invokable Controllers

- If you have a controller that contains only one function, you can make it invokeable – this means it can be used without having to specify a function name
- To do this, simply name the single function __invoke(), with whatever parameters you consider necessary
- When referencing the controller in a route, you need only provide the controller name:

```
Route::get(‘/about’, ‘AboutController’)
```

- Considering a major impetus for using controllers is to group together and organize similar code, the usefulness of single-function invokeable controllers is questionable (Increased readability vs. organization)

★ Requests and Controllers

- Often, we'll be handling data submitted via a POST request – this data may need to be inserted into a database or manipulated in some way, ideally without having to use the PHP global _POST and _GET variables
- Fetching this data in the controller is relatively simple: we use the `Request` object:

```
use Illuminate\Http\Request;
```

- For our function, we have it take a Request as an input parameter:

```
Public function getPrice(Request $request){ …
```

- No special handling is required in the routes file, other than defining the route as a Route::post, rather than a Route::get
- Requests can also be handled as standard input parameters in route closure functions, if you want to process the POST data in your route file (again, not highly recommended)

★ The Request Object

- The Request object contains all of the data submitted with the request
- We can pull out all of the data at once in an associative array format using `$request->all();`
- We can also pull out individual values using `$request->input('varname');`
- To check if a given variable exists in the Request object, we can use `$request->has('varname')`
  - Note: `has('varname')` will return true if the input exists, but is empty. Use `filled('varname')` to test if an input exists ***and*** has a value
- Request objects can also be used to fetch GET parameters using `query('varname')`
- Many of these functions also allow for a second parameter as a default value should the variable not be found

★ Returning a View

- Returning a view from a controller is relatively simple:

```
return view('welcome');
```

- When we're serving a dynamic page, we usually want the page structure to remain the same, while changing the actual data that is being served
- To do this, we can return a view, as well as additional data:

```
return view('welcome', ['user' => $username]);
```

- There are many ways of returning data with the view. The associative array above works well, but can be difficult to read and bulky if we have a lot of variables
- An alternative is using the PHP function compact():

  Return view('welcome', compact('username', 'date', 'role', 'location');

  - Compact() can take strings, and will attempt to match them to existing variables in the symbol table. It then returns an array of the variables
  - A downside of this is that your variable names in your controller must match the variable names in your view if you use compact

Views in Laravel

★ Views & Blade Syntax

- The views for your application are available in the `/resources/views` directory in your Laravel install
- Views in Laravel use the `.blade.php` file extension
- Blade is the templating engine used by Laravel. It provides special functionality for echoing data into HTML, as well as supporting an inheritance system
- The Blade system is designed to provide a better interface for dynamically changing the markup (HTML) of a page based on PHP variables, without the headache of having to generate HTML in PHP through concatenation and similar means
- While it might be tempting to put additional PHP code into your views for small fixes and quick changes, try to keep your code (controllers) and your markup (views) separated. The only logic that should be in your views is logic based solely on data passed from the controller that directly impacts how markup is displayed

★ Blade Inheritance

- Blade files can extend existing blade files, allowing you to combine different layouts and reuse front-end HTML, rather than having to duplicate identical elements across multiple pages
- Typically, you'll have a 'master' template that contains structures constant across all pages
  - For example, you might have a <head> element, body container, navbar, and footer that remain constant across all pages
- This master template will have `@yield` directives that tell the Blade engine to insert markup at specific points
- Child templates in turn `@extend` the master template, and specify may specify one or more `@section` directives, which will be inserted into the master template at the appropriate `@yield`

An Example of Blade Inheritance

Master.blade.php

```
<!DOCTYPE html>
<html>
        <head>
                <title>@yield('page')</title>
                <link rel="stylesheet" href="css/style.css">
                @yield('pageCSS')
        </head>
        <body>
                <div class="container">
                @yield('content')
                </div>
                <div class="footer">
                        <p>Web Design © Matt Fritter</p>
                </div>
        </body>
</html>
```

Page.blade.php

```
@extends('Master')

@section('page', 'My Webpage')

@section('pageCSS')
        <link rel="stylesheet" href="css/pageStyle.css">
@endsection

@section('content')
        <p>Lorem ipsum dolor sit amet...</p>
        <img src="article1.png" width="100%">
        <p>... Further content</p>
@endsection
```

An Example of Blade Inheritance

Resultant View Returned

```
<!DOCTYPE html>
<html>
	<head>
		<title>My Webpage</title>
		<link rel="stylesheet" href="css/style.css">
		<link rel="stylesheet" href="css/pageStyle.css">
	</head>
	<body>
		<div class="container">
			<p>Lorem ipsum dolor sit amet...</p>
			<img src="article1.png" width="100%">
			<p>... Further content</p>
		</div>
		<div class="footer">
			<p>Web Design © Matt Fritter</p>
		</div>
	</body>
</html>
```

- In order to return the view, we call the child view, which will also include the content of the parent template that it extends:

  ```
  return view('Page');
  ```
- You can have multiple levels of extension, i.e. a blade that extends a blade that extends a blade, etc
- Note that `@yield` directives don't necessarily have to return anything. For example, you could extend Master, and not provide a 'pageCSS' section. As a result, that `@yield` will simply be ignored (blank)

★ Other Means of Combining Templates

- The Blade system provides other means for combining templates beyond inheritance
- The `@include` directive allows you to inject the contents of an entire Blade template into another template, without having to use sections or yields
- An example use case: You have a widget such as an image slider that you may want to use in a variety of contexts on some pages
  - Create a Blade template for the widget that contains all the markup required for the slider – `slider.blade.php`
  - Use `@include(‘slider’)` to inject the slider code somewhere in another view
- You can also use the render() function in a controller to generate a raw HTML string, allowing you to pass the HTML as a variable to another view:

```
$html = view(‘slider’)->render();
return view(‘page’, [‘html’ => $html]);
```

★ Handling Variables in Blades

- Recall that we can pass data from controllers and routes to views:

```
return view('welcome', ['user' => $username]);
```

- Displaying this data in the blade is very simple. Simply wrap the variable in double braces to echo the variable into the HTML:

```
<p>Welcome, User {{ $username }}</p>
```

  - The double braces automatically escape markup and script content to prevent Cross-Site-Scripting (XSS) attacks. Attempting to echo HTML will result in the HTML being shown as plain text
- There is also an unescaped version, which allows you to properly echo HTML and script content:

```
<p>Welcome, User {!! $username !!}</p>
```

  - This can potentially be very dangerous. As a general rule, you should always escape user-generated content (usernames, comments, etc)
  - If you must echo user-generated content in an unescaped form due to it being combined with HTML (say, when using a render() function as previously used), it is recommended you first manually escape the user-generated content using the `htmlspecialchars(<string>)` function, which escapes &,",',<, and >

Handling JSON Arrays in Blades

- When passing data to a view for use in JavaScript, it is often useful to convert it to JSON notation, particularly when handling arrays. PHP arrays aren't especially compatible with JS, and it's usually easier to convert them to JSON and parse them through JS than it is to convert the PHP array into a JS array declaration
- The blade engine contains a special command for converting PHP array variables into JSON outside of the controller, `@json($vars)`
- This is identical to using the json_encode($var) PHP function
  - Whether you want to handle JSON encoding in your controller or your view is your choice, although keep in mind that UTF-8 characters may pose problems

★ Logic in Blades – Isset and Empty

- What? I thought you said logic was supposed to stay in the controller?
- For the most part it should, but sometimes there is a use case for logic in your view, particularly when it involves transforming markup based on variables
- Probably the most useful logic included in the Blade engine: the `@isset` and `@empty` directives
  - Trying to echo a non-existent variable {{ $non-existent }} will cause Laravel to throw an error
  - Solution: Wrap it in an isset:

```
@isset($var)
    {{ $var }}
@endisset
```

  - The `@empty` directive functions identically, but only runs it's code if the passed variable is empty. Note: empty is ***not*** the same as non-existent. `@empty` is good for printing out a message if a DB query returns nothing (i.e. "No search results found")
  - Any markup, including echoing variables, contained between the beginning and end of the directive will only be included if the directive returns true (i.e. the variable is set, or the variable is empty)
- Using these directives, you can repurpose a template and handle unexpected outputs without having to create multiple Blade files
  - Want a simple user notification system for errors or notifications? Use an isset that appends a content box to the top of the page containing the variable contents. If something goes wrong, pass the error message as a variable from the controller to the view

★ Logic in Blades – Conditionals

- The Blade engine also supports `@if`, `@elseif`, `@else`, and `@endif` directives, allowing you to change markup based on passed variables:

```
@if ($userRole === “Full-Stack Developer”)
    <p>Welcome, benevolent web God!</p>
@elseif ($userRole === “BOFH”)
    <p>Please don’t rm –rf me!</p>
@else ($userRole === “User”)
    <p>Try not to break anything</p>
@endif
```

- Note: You can nest directives: an `@isset` around an `@if` will ensure the conditional variable exists before trying to evaluate the condition (which would lead to an error, if it didn’t)

★ Logic in Blades - Loops

- The Blade engine can also be used to loop over arrays using the `@foreach` directive
- This is very handy when you have a result set from a DB query that you want to echo out in some kind of table, without having to create it as a string in your controller, pass it to the view, and echo it
  - Also allows you to use Laravel's built-in string escaping for user-generated content
- An example:

```
@foreach ($products as $product)
    <tr><td>{{$product->name}}</td><td>{{$product->price}}</td><tr>
@endforeach
```

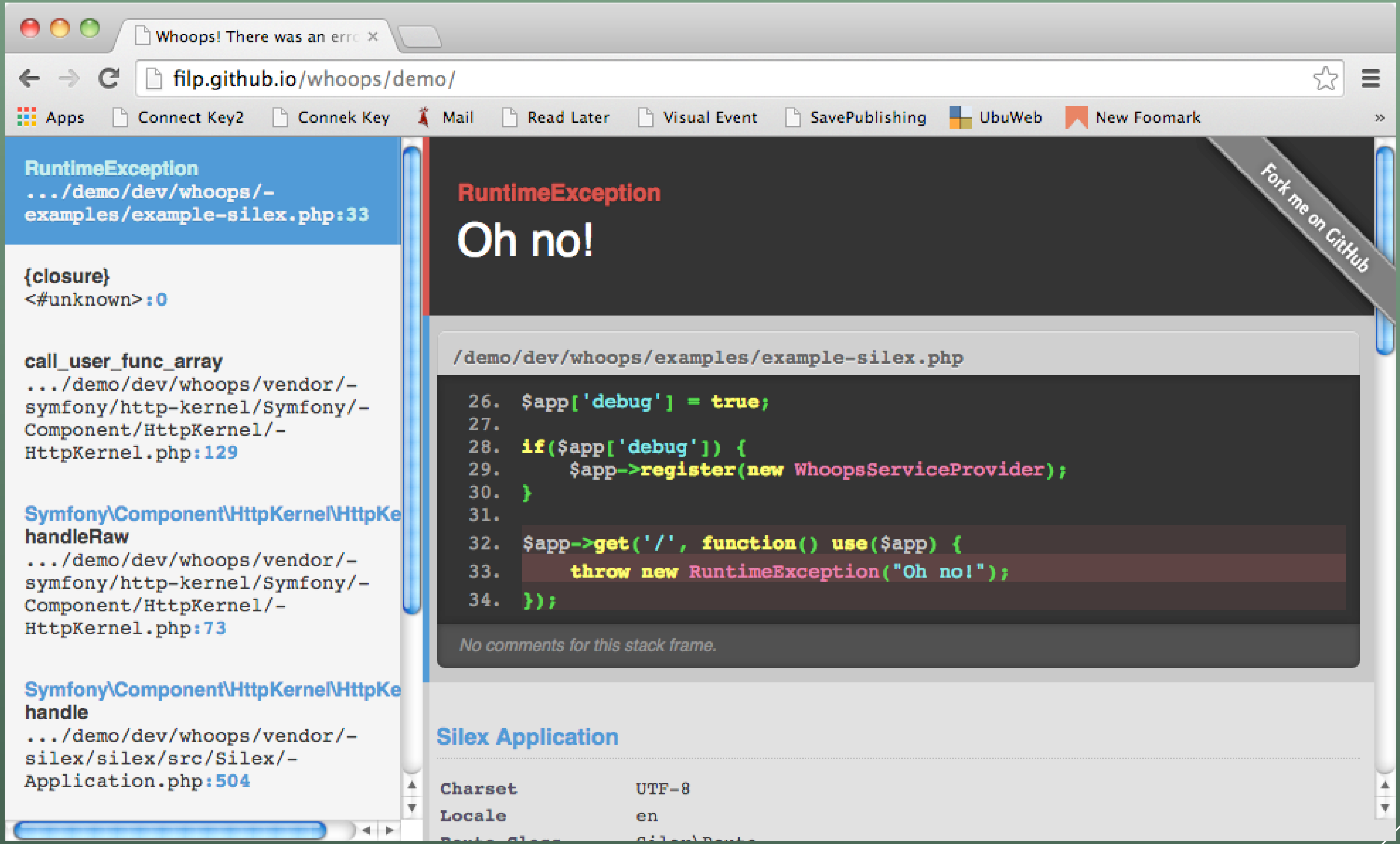
- Again, like the conditionals, these directives can be nested within each other. Nest the foreach and table markup in an if directive to check if the result is empty or not – then use the empty directive to return a message instead notifying the user

Blade Documentation

- The Blade engine is an extremely powerful and easy to use tool, and there are additional logic statements that aren't covered here
- We'll talk about Laravel's authentication system next week, including a couple of conditional Blade directives designed to check user authentication status
- Full documentation (or almost full) can be found at: https://laravel.com/docs/5.7/blade
- Blades also support custom directives, if there's a particular function you think you'll be re-using frequently in your views

Errors in Laravel

Eventually You'll See this



RuntimeException
.../demo/dev/whoops/-
examples/example-silex.php:33

{closure}
<#unknown>:0

call_user_func_array
.../demo/dev/whoops/vendor/-
symfony/http-kernel/Symfony/-
Component/HttpKernel/-
HttpKernel.php:129

Symfony\Component\HttpKernel\HttpKe
handleRaw
.../demo/dev/whoops/vendor/-
symfony/http-kernel/Symfony/-
Component/HttpKernel/-
HttpKernel.php:73

Symfony\Component\HttpKernel\HttpKe
handle
.../demo/dev/whoops/vendor/-
silex/silex/src/Silex/-
Application.php:504

RuntimeException

Oh no!

/demo/dev/whoops/examples/example-silex.php

```
$app['debug'] = true;

if($app['debug']) {
    $app->register(new WhoopsServiceProvider);
}

$app->get('/', function() use($app) {
    throw new RuntimeException("Oh no!");
});
```

No comments for this stack frame.

Silex Application

Charset	UTF-8
Locale	en

The Serial Offenders

- PHP as a language is very insistent about syntax
- The worst offenders, and the ones you should check for first and foremost are:
  - Missing semi-colons on line endings;
  - (([Mismatched] brackets)
  - “Mismatched quotation marks around strings’
  - Missing {{ dollar }} {{ $signs }}
- Sometimes, it’s useful to see an entire stack trace when you’re trying to track down why a certain input is causing an error. The two log files you’ll probably want to see are:
  - Laravel log file: storage/logs/laravel.log, from the Laravel base directory
  - Apache log file: /var/log/httpd/error_log, from the root
  - The Laravel log file will catch PHP errors and internal Laravel/Symfony/Package errors – these are most likely what are causing the problem
  - The Apache log file will catch primarily errors to do with your server configuration and/or permissions issues

A Note on the Debug Page

- Security-wise, it's usually not a good idea make debug stack traces publicly available – they can contain sensitive data and expose the inner workings of the server
- You can disable the Laravel debug web page, and replace it with a generic "Oops, something went wrong" error page by editing your .env environment file:

```
APP_DEBUG=false
```

That’s All Folks

Next Week – Query Builder, Advanced Laravel Functionality, API Development