28

Building a Web-Based Email Client with Laravel Part I

MORE OFTEN THAN NOT YOU'LL USE A FRAMEWORK TO BUILD YOUR APPLICATIONS, and they are as varied as the colors in a rainbow. In this chapter we introduce one of the more popular modern PHP frameworks, Laravel 5. And then in the next chapter we walk you through the creation of a basic web-based IMAP client built using it.

Introducing Laravel 5

Laravel 5 as a framework could most easily be described as a highly opinionated framework. That is to say, Laravel applications impose a significant amount of pre-defined structure as to how your applications should be organized. The plus side to this approach is it typically is very easy to get started with Laravel building most applications. The down side, however, is as your application's needs become more specific it can require more effort than a less opinionated framework.

Creating a New Laravel Project

The best way to get started with Laravel is simply to create a new Laravel project using one of many different tools available for doing so. Laravel requires an environment running PHP version 5.5.9 or above with the OpenSSL, PDO, Mbstring, and Tokenizer extensions available. If you don't have a suitable environment, the Laravel project also provides a virtual machine called Laravel Homestead designed specifically for developing Laravel applications.

To create a Laravel project, you have a few different options. If you are looking to create a one-off Laravel project the easiest approach is simply to use `composer create-project` and give your project directory a name as follows:

```
$ composer create-project –prefer-dist laravel/laravel pmwd-chap29
```

In this example, this would create a basic Laravel project in the `pmwd-chap29` directory.

If you are going to be using Laravel frequently for multiple web applications, it might be easier to install the Laravel Installer command-line tool using composer and then use the tool to create projects as needed. In this route, you'd start again by using composer but this time to install the Laravel Installer:

```
$ composer global require "laravel/installer"
```

This will install the laravel command in the composer vendor bin directory (generally `~/.composer/vendor/bin`) for use in creating projects. To use it, make sure this directory is available in your user's PATH and then create a project simply as follows:

```
$ laravel new pmwd-chap29
```

Once the project has been created, there are many things that could be configured or otherwise optionally done to further customize the base generic Laravel project. Of these things, the only one that is really critical will be ensuring certain directories of the project or writable by the web server. Starting from your project directory (in our case, `pmwd-chap29`), you'll need to make sure the following directories are writable by your web server in order for Laravel to properly function:

```
./storage/
./bootstrap/cache
```

Note

While Laravel does not require much configuration out of the box to function, there are plenty of configuration settings to tweak as necessary for various Laravel components. The `config/` directory is loaded with various configuration files to set up caching, debugging, database access, mail servers, and more. It is recommended that at the very least you open the `config/app.php` configuration to adjust some application configuration settings appropriately (such as the application timezone or locale).

The Structure of a Laravel Application

The overall structure of a new Laravel application contains a number of folders described below:

- The bootstrap directory containing logic necessary for the framework to start, configuring autoloading, etc.
- The config directory containing all of the application's configuration values
- The database directory containing database schema migrations and logic to seed those schemas with data as necessary
- The public directory which is the document-root of the application and thus houses all served assets such as stylesheets, images, etc. as well as the `index.php` entry point into the application

- The resources directory which contains views, internationalization files, and other application resources
- The storage directory which contains temporary storage items for the framework and application such as compiled views, session data, cache data, and logs
- The tests directory which contains any automated tests you create
- The vendor directory which contains all application dependencies loaded via Composer
- The app directory containing the code application logic

The primary focus of your development work in Laravel will be dealing with files in the app directory which is designed to contain the vast majority of your application logic. By default this directory is namespaced under the App namespace and thus you can add to the provided structure of classes easily. For example, if you wanted to create a class `Foo` for your app it could be placed in the `app\Library\Foo.php` file and will automatically assume the fully qualified namespace of `\App\Library\Foo`.

Note

If you would like to change the name of the base Namespace from the default App to something else, please refer to the Laravel Artisan command `app:name` in the Laravel documentation

In a new installation of Laravel 5, the app directory is populated with the basic classes and structure needed to create a simple one-page application. Thus, the app directory's contents can be further broken down as follows:

- The Commands directory which contains any commands you create in Laravel for the application. These commands can be CLI commands, asynchronous jobs, or simply synchronous tasks to be executed by other portions of the application. In many web applications creating commands may not be necessary
- The Events directory which contains classes for your application used in the event manager provided by the Laravel Framework. Laravel supports class-based Events as well as non-class based Events and this directory is designed to house any Event classes you create.
- The Handlers directory is a counterpart to the Events directory described above. Its role is to contain classes that handle events that are fired within the application and perform logic based on those events. For example, one could have a new user event class in the Events directory which is fired causing a `HandleNewUser` handler class to be created to handle that event object.
- The Services directory contains any services defined by your application. In Laravel Services are helpers that allow you to keep core functionality of your application abstract for ease of reuse. For example, one could create a service class to handle user

management that is used by other parts of the application. Another example would be the creation of a service to interface with another web site, etc.

- The Exceptions directory as one might expect houses any custom application exception classes to be thrown in the event of an error.

Note

Most if not all of the classes to be stored in the app directories described can have valid templates for them auto-generated using the Laravel `artisan` CLI command. A complete list of generators can be found by running `php artisan list make` from the root directory of your Laravel application.

As indicated in this chapter we will be building out a basic web-based IMAP client to demonstrate the use of both Laravel and the IMAP functions of PHP. As we go through this exercise, many of the concepts and directories introduced in this section will be revisited in practical demonstration.

The Laravel Request Cycle and MVC Pattern

Now that we have introduced the basic structure of an application let's take some time to explore how these directories and files within them are put together and used from start to finish. Every time a Laravel application wants to do something, be that serve a request made by a browser or execute a CLI command via the `artisan` command, the first code to be executed is the `public/index.php` script. By itself, it is a very simple script that only serves to kick-start the rest of the framework and thus application. It is responsible for things like creating an instance of the primary Laravel application class of the framework, loading any auto-loaders provided by Composer, and other low-level tasks. Ultimately, this script hands off execution to the Framework at large through the fundamental Laravel application class

Application Kernels

Depending on the nature of the request (was it initiated by a web request via HTTP, or by the execution of a command in the console) the request is handled by its corresponding application framework kernel. For our purposes we will only be focusing on the HTTP kernel which is the single point-of-entry of a HTTP request into the rest of the framework and your application. The HTTP kernel is responsible for defining the majority of core needs of every request such as logging, error handling, etc. and can be modified as necessary to suit the needs of your application.

The application kernel, in addition to setting up core tools and requirements of the request, is also responsible for defining any so-called application middleware that all requests must execute before being accepted by the real application. In the context of an HTTP request examples of middleware include handling HTTP session data, processing cross-site request forgery tokens in form submissions, and various other tasks that logically must supersede

any application logic. The application kernel is also responsible for the loading and handling of the Service Providers of the application discussed in the following section.

In practical terms, the application's HTTP kernel class can be found in the `app/Http/Kernel.php` file, extending from the framework's `Illuminate\Foundation\Http\Kernel` class. This provides an access point to you as the developer to customize the use of middleware, logging, error handling, etc. easily as necessary.

Service Providers

The next step of the framework in handling a request is to initialize and register within itself any defined Service Providers. Service Providers are a critical component of the overall Laravel framework, providing most if not all the useful functionality of the framework ultimately to your application. The purpose of Service Providers is to expose functionality or customize existing functionality and they are defined in your application's configuration, specifically in the `config/app.php` configuration file under the providers key. By default, a new Laravel application defines providers that expose functionality for everything from encryption to pagination.

For those who may be interested in writing libraries for Laravel applications that can be consumed by others, Service Providers are the entry point of how to do so. For example, if you wished to develop a Laravel component that provided access to Twitter, you would write your library and provide a Service Provider to expose that library and its functionality within Laravel. Then, in your application, you would list that Service Provider in your configuration, which would enable you to access it throughout.

Even if you are not interested in the creation of component libraries, Service Providers are a key aspect of any Laravel application. By default, a new Laravel application defines various providers in the `app\Providers` directory of your application to initialize various components. These providers (and others you create if so needed) allow your application to initialize any resources you might need prior to handing the request off to the logic of your application.

Understanding the Laravel Model, View, and Controller Classes

Once Service Providers have been registered and initialized, the request will enter the Model, View, and Controller (MVC) aspect of the framework where the majority of your application logic will reside. Specifically, once the Service Providers have been initialized the framework will pass the request to the Laravel router which is responsible for mapping the content of the request to the appropriate controller within your application.

The Laravel Router

Compared to other frameworks, the Laravel Router is one of the easiest mechanisms to use when it comes to getting a request to the appropriate controller within your application. In general, routes are defined in the `app/Http/routes.php` file for HTTP requests. This file is

simply a PHP script that should contain as many static calls to the Laravel Route class as needed to define, in this case, how specific paths and protocols map to their respective controllers.

In Laravel there are basic route methods mapping to each of the HTTP methods available that a request can originate from. For example, the below code is a basic route which maps an HTTP GET request for document root (`'/'`) to the simplest controller available in Laravel (a closure) and a second HTTP POST request to /submit to another closure:

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

Route::post('submit', function() {
    return 'You sent us a POST request';
});
```

In addition to the common HTTP GET and POST methods, Laravel also provides `Route::put()`, `Route::delete()`, and more. If you wish to have a single set of code handle multiple HTTP methods this can also be done either by using the `Route::any()` method (which will match the route based on path only regardless of HTTP method used), or you can explicitly define which methods are accepted using the `Route::match()` method as shown:

```
Route::match(['post', 'put'], '/', function() {
    return 'This request was either a HTTP POST or an HTTP PUT';
});

Route::any('universal', function() {
    return 'You made a request to /universal, and we accepted any valid HTTP
method.';
});
```

Route Parameters

In addition to specifying simple paths as routes, Laravel's router also enables you to create routes which contain variables using a simple syntax. For example, if you wanted to create a GET route to retrieve an article with a specific ID specified in the path (i.e. `/article/1234`) you could do so as follows:

```
Route::get('article/{id}', function($id) {
    return 'You wanted an article with the ID of ' . $id;
});
```

If the parameter for a given route is optional, you can add `?` to the end of a variable name in the route definition to indicate this:

```
Route::get('articles/{id?}', function($id) {
    if(is_null($id)) {
```

```
        return 'You wanted all the articles';
    } else {
        return 'You wanted an article with the ID of ' . $id;
    }
});
```

When specifying routes with optional parameters, a default value for the parameter can be defined simply by setting a default value for the parameter when the function or method is declared:

```
Route::get('articles/{id?}', function($id = 1) {
    return 'You want an article with the ID of ' . $id . ', which defaults to 1 if
unspecified.';
});
```

Often, when using parameters in your routes, you will desire to have more control over what is an acceptable parameter. In the examples shown thus far, the `{id}` parameter could have been a number or a string which is in many cases contrary to the design of your application. To enforce a specific format for a given route parameter, once defined you can use the `Route::where()` method to impose restrictions using a regular expression on the value of a given parameter in order to match the route as shown:

```
Route::get('articles/{id}', function($id) {
    return 'You provided a numeric value for ID of ' . $id;
})->where('id', '[0-9]+');

Route::get('articles/{id}', function($id) {
    return 'You provided an alphabetical value for ID of ' . $id;
})->where('id', '[A-Za-z]+');
```

In the above example we define two distinct routes using the same endpoint of `/articles/<ID>`, the first will handle any values of ID which are numerical and the second will handle any values of ID which are alphabetical. If the value of ID fails to meet either criteria no route will be matched and a 404 error will be returned.

It is also worthwhile to note that routes can contain any number of parameters in the same format, and in that case can be restrained on their values by passing an array of regular expressions to the `Route::where()` method instead as shown:

```
Route::get('articles/{section}/{id}', function($section, $id) {
    return "You requested an article in section ID of $section and ID of $id";
})->where(['section' => '[A-Za-z]+', 'id' => '[0-9]+']);
```

The above example would match routes in the form of `articles/history/1234` but not `articles/1234/history`.

Note

If desired it is possible to define global constraints for a specific parameter across all routes by using the `Route::pattern()` method. To do so, you would need to modify your `RouteServiceProvider::boot()` method and define the pattern for a route:

```
$router->pattern('id', '[0-9]+');
```

Route Groups

Often when defining routes it is desirable to be able to define global behaviors for one subset or another. For example, some routes may only make sense if the user has been previously authenticated while other routes may all exist under a single base path such as authenticated/view and authenticated/create. For these sorts of purposes Laravel supports Route groups.

To group a set of routes together we will make use of the `Route::group()` method. This method takes as its first parameter a series of definitions common to all of the members of the group and as its second parameter a closure which contains the individual routes that belong to that group. Using our previous example if we wanted to create a set of routes which all existed under the 'authenticated' path we could do as follows:

```
Route::group(['prefix' => 'authenticated'], function() {
    Route::get('view', function() {
        return 'This is the authenticated\view route';
    });

    Route::get('create', function() {
        return 'This is the authenticated\create route';
    });
});
```

The Laravel Router supports many different definitions for route groups such as the prefix one used above. Below is a sample of the various types of group definitions that can be created.

- prefix – Defines the path to prefix to all routes within this group.
- middleware – Defines an array of middleware components that apply to all routes in this group.
- namespace – Defines the namespace under which all referenced controllers are located in this group
- domain – Allows you to route based on the sub-domain of the hostname in the request provided, giving you that sub-domain as a parameter. For example, [`'domain' => '{account}.example.com'`] would match a request to http://myaccount.example.com/ and the first parameter of the controller would be passed with the value `'myaccount'`.

For more examples of route definitions and their various configurations please consult the Laravel manual on routing at: https://laravel.com/docs/5.2/routing

Using Controllers

In the previous section on Routers all logic for a given route was contained within a closure function. While valid this is generally not the most common way to define the logic for a route. Rather than defining the logic within a closure, a method within a controller class is generally used.

Controllers are useful for organizing the logic of your application, grouping related code together into a single class located in the `app/Http/Controllers` directory of your application. A new Laravel project ships with the `App\Http\Controllers\Controller` class which should serve as a base class for your own controllers. For example, consider the following Controller example (stored as `App\Http\Controllers\MyController.php`):

```
namespace App\Http\Controllers;


class MyController extends Controller
{
    public function myAction()
    {
        return "This is our first action";
    }
}
```

To use this controller as the end point for a given route we have multiple approaches. The two most common approaches are to add routes to your application as in the following example for HTTP GET requests:

```
Route::get('/', 'MyController@myAction');
```

Which uses the format `<Controller>@<method>`, where `<Controller>` is assumed to be a class in the `App\Http\Controllers` namespace unless explicitly specified otherwise and `<method>` is a public method within that controller. Alternatively a more verbose approach can be used if specifying additional Route definitions:

```
Route::get('/', [
    'uses' => 'MyController@myAction'
]);
```

This approach using the `'uses'` key allows you to also specify middleware or other relevant attributes to the route as well.

As was the case in the section on Routers where closures were used rather than controllers, the same approach to route parameters mapping to method parameters applies. It simply is done as a method of the class instead of as a closure.

While it is reasonable to do things like specify the middleware for a given route when defining the route, using controllers middleware specifically can also be specified in the

constructor for the controller class. This may be preferable depending on how your code base is organized. For example, to add the `auth` middleware to our controller we could update our example as follows:

```
namespace App\Http\Controllers;

class MyController extends Controller
{

    public function __construct()
    {
        $this->middleware('auth');
    }

    public function myAction()
    {
        return "This is our first action";
    }
}
```

Another powerful feature of controllers is the ability to inject dependencies. For this example, let's assume you have created a `ServiceProvider` that registers a new service `MyTool` as an object available to your application. If you wanted to inject automatically an instance of `MyTool` into your controllers, you need to only specify it via type-hinting when the controller is constructed:

```
namespace App\Http\Controllers;

class MyController extends Controller
{
    protected $_myTool;

    public function __construct(MyTool $foo)
    {
        $this->middleware('auth');
        $this->_myTool = $foo;
    }

    public function myAction()
    {
        return "This is our first action";
    }
}
```

This injection ability of Laravel controllers extends to the method level as well. For example, if you would like access to the `Request` object (which contains all the data available for the current request such as HTTP GET/POST data, route parameters, etc.) it need only be specified in the method declaration:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class MyController extends Controller
{

    public function __construct()
    {
        $this->middleware('auth');
    }

    public function myAction(Request $request)
    {
        $name = $request->input('name');
        return "This is our first action, we were provided a name: $name";
    }
}
```

Accessing Request Data

As alluded to in the previous example, in practical usage all data regarding a HTTP request made to the application is contained within an instance of the `Illuminate\Http\Request` object which is available to controllers using the injection mechanisms. This class is comprehensive of all data available for the request. Some of the more common methods available within it will be introduced below however a complete list is available in the Laravel documentation.

The most common use of the Request object in a controller is accessing data associated with the request such as GET query parameters or POST data. In Laravel, all input access is consolidated into a single method Request::input() taking two parameters. The first is the key by which the data should be accessed (such as the variable name) and the second a default value if that key does not exist. For example, consider the following hypothetical request to http://www.example.com/myaction?name=Joe, the data for the variable name could be accessed as follows:

```
public function myAction(Request $request)
{
    $name = $request->input('name', 'John');
    return "The name I was given was '$name' (default is John)";
}
```

When accessing data using the `Request::input()` method much more than the simple variable name can be used. For example, if the input is a complex data type (such as an array), a unique dot-notation can be used to resolve a value deep within an array. For example the below is identical to the value stored in the hypothetical PHP array `$myarray['mykey'][0]['name']`:

```
public function myAction(Request $request)
{
    $name = $request->input('myarray.mykey.0.name', 'John');
}
```

Note this syntax can even be used when the input data is in JSON format, however in this case it is important the request specify a Content-Type header with the value of `'application/json'`.

If you would like to test to see if an input variable exists (rather than specifying a default value), the Request::has() method is what you seek:

```
public function myAction(Request $request)
{
    if(!$request->has('name')) {
        return "You didn't specify a name";
    }

    $name = $request->input('name');
    return "The name you specified was $name";
}
```

Finally, you can return an associative array of all input data in various ways. The most complete is the `Request::all()` method which will return all input data. If you would like to return a subset of data, `Request::only()` will only return specific listed variables while `Request::except()` will return all variables not explicitly excluded as shown below:

```
public function myAction(Request $request)
{
    $allInput = $request->all();
    $onlyNameAndPhone = $request->only(['name', 'phone']);
    $allButPassword = $request->except('password');
}
```

The final usage of the Request object we will examine is dealing with input data in the form of an uploaded file. Accessing an uploaded file from a request is done primarily through the `Request::file()` method, which returns an instance of `Symfony\Component\HttpFoundation\File\UploadedFile` providing further helpers for working with the file. Two common requirements when working with a file upload are to determine if the uploaded file is valid uploaded file and secondly moving that file from its temporary upload storage to a longer-term destination on the file system. These, combined

with the `Request::hasFile()` method are used below in a demonstration of working with file uploads:

```
public function myAction(Request $request)
{
    if(!$request->hasFile('photo')) {
        return "No file uploaded";
    }

    $file = $request->file('photo');

    if(!$file->isValid()) {
        return "Invalid File";
    }

    $file->move('/destination/path/for/photos/on/server');

    return "Upload accepted";
}
```

Using Views

As has been demonstrated in previous sections of this chapter, one can take advantage of Laravel using simply controllers to contain logic (and returning the output of that controller as a string). In a web application however, the separation of presentation logic is a critical aspect of good code organization. For these purposes the View component exists which allows you to generate complex output (such as HTML) easily using data retrieved or calculated by a controller.

The View component of Laravel is a stand-alone tool, and can be used anywhere generating things like HTML would be valuable. It is used in the Laravel mail component, for example, to generate the email being sent to a user. In its simplest form implementing a view can be done by first creating a view template in the `resources/view` directory and then using the View component of Laravel to render an instance of that view with specific values for any variables contained within it. For example, consider the following simple HTML template, stored in the `resources/view/welcome.php` file:

```
<html>
    <head>
        <title><?=$pageTitle?></title>
    </head>
    <body>Hello, this is a view template</body>
</html>
```

To render this view as a string, one simply needs to use the `view()` helper, providing as the first parameter the name of the view, and in the second parameter an array of `key/value` pairs of values to make available to the view:

```
<?php

Route::get('/', function() {
    return view('welcome', ['pageTitle' => 'Welcome to using views!']);
});

?>
```

Note that the name of the view being rendered is essentially the name of the file in the resources/views directory omitting the `.php` extension. View templates are typically organized into directories within the over-arching `resources/views` directory of the application in some sort of logical way. One classic organizational technique is to store views in directories corresponding to the controllers they are used in conjunction with. Then, to specify a view in a sub-directory simply use a straightforward dot-notation. This example would reference a view template found in the file `resources/views/mycontroller/index.php`:

```
<?php

Route::get('/', function() {
    return view('mycontroller.index', ['somevariable' => 'somevalue']);
});

?>
```

Note

Typically Views are designed only have access to data that is provided to them through the use of the `view()` helper. However, it is also possible to provide "global variables" within a view that are available to all views by default. This can be done in the following way:

```
view()->share('key', 'value');
```

In order to ensure these "global view variables" are accessible from all views, it is best to put them for example in one of your application's Service Providers (typically in the `boot()` method). For more information please see the Laravel documentation on shared view variables.

Blade Templates

An optional but recommend aspect of the View component within Laravel is a templating engine for Views known as Blade. Although it is entirely acceptable to use pure PHP scripting in a Laravel view template, the Blade template engine does provide a number of powerful organizational features that allow you to create templates in a logical way without having to build a lot of actual logic into the view itself. Note, a Blade template can contain

PHP code just like any other view as well. The two major benefits to Blade are the ability to inherit templates and define sections, which we will discuss below.

Blade templates are created just like any other view, with the exception that the templates are stored with the `.blade.php` extension instead of just the `.php` extension. To demonstrate how Blade templates work, let's consider a common problem in a web application – providing a consistent look and feel to your user experience.

Where using pure PHP templates would require the use of many calls to `view()` (or perhaps `include` statements) to load common features of your user experience, the Blade template engine allows you to not only define high-level layouts of your views but then define sections within that layout that can be overwritten or extended as needed by other child views. For example, let's define a simple HTML layout we will store in the `app/resources/views/layout.blade.php` file:

```
<html>
    <head>
        <title>@yield('title')</title>
        @section('stylesheets')
            <link href="/path/to/stylesheet.css" rel="stylesheet"/>
        @show
    </head>
    <body>
        <div class="container">
        @section('sidebar')
            <div class="col-md-4">
                <!-- Sidebar Content -->
            </div>
        @show

        @yield('content')
    </body>
</html>
```

In this Blade template we introduce you to two primary directives, the `@yield` directive, which outputs the content stored under a given string identifier and `@section`, which defines an extendable or replicable section of the layout. The intention is that a layout such as above is never directly referenced when rendering a view. Rather, within the view that is being rendered you indicate that this view extends from the layout and define the necessary values and sections. For example, consider the layout `resources/views/welcome.blade.php` that extends our layout:

```
@extends('layout')

@yield('title', 'My Page Title')
```

```
@section('stylesheets')
@parent
<link href="/path/to/another/stylesheet.css" rel="stylesheet"/>
@stop

@section('sidebar')
<div class="col-md-4">
    <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/account">My Account</a></li>
    </ul>
</div>
@stop

@section('content')
Hello World!
@stop
```

As was used in previous examples, this child blade template `resources/views/welcome.blade.php` would be referenced using the `view()` helper from a controller. Though the use of the `@extends` Blade directive we can see that it is a child of the `resources/views/layout.blade.php` template. Thus, this view when rendered will contain all the content of the Layout, replacing sections and yield values with the values provided by the child template (such as page title). In some situations, such as the `'stylesheets'` section, we don't want to wholesale replace the section but rather add additional stylesheets relevant to this page to it. Thus, within the section we use the `@parent` Blade directive to inject the parent's content for that section before adding our own. Because we don't want these sections to render immediately (only replace the content in sections of the same name defined in the layout), we use `@stop` instead of `@show` to end the section.

As a final note on Blade templates it is important to discuss how variables passed into the template from a controller are accessed. As was previously demonstrated you can always use raw PHP to print a template variable. The recommended approach, however, is to use the syntax provided by Blade which is to encompass the variable in `{{` and `}}`. For example:

```
Hello, {{ $name }}, we are using Blade templates!
```

By default, using the `{{ }}` syntax for variables will automatically escape the content for safe display in an HTML document through the use of the `htmlentities()` PHP function. If you wish to display the content of a variable literally with no escaping this is also possible by using a slightly different `{!! !!}` syntax:

```
Hello, {!! $name !!}, your name was not HTML escaped so potentially could cause an
XSS attack.
```

These are only some of the features available as part of the Laravel View Component and Blade template engine. However, these core basics are sufficient for our purposes to build our Web-based email client. If you would like to learn more about the View component, please refer to the Laravel documentation.

Laravel Models

No discussion of the MVC design pattern would be complete without a discussion of Models, or the data layer of the pattern. Fundamentally for web applications, the concept of a model in MVC is an object which has the ability to retrieve and store data within it and doesn't necessary have any specific design beyond that. In Laravel, models typically are implementing using something in Laravel called Eloquent, a robust approach to another design patterned called Object-Relational-Mapping (otherwise known as ORM).

ORMs are some of the more complicated code you'll ever find in a framework, but their purpose is remarkably simple. Instead of your web application writing SQL queries to store and retrieve data from your relational database, the framework does all the SQL generation automatically and instead presents you as the developer with an object-oriented approach to database interaction.

To get started with Laravel Elqouent models first you will need a database (we'll be using MySQL) and to appropriately configure a connection to it within your Laravel application. These configuration settings can be found in the `config\database.php` file of your application under the `'connections'` key of the array contained within. Here's an example of a valid configuration to a MySQL server residing on localhost:

```
…
    'connections' => [
        'mysql' => [
            'driver' => 'mysql',
            'host' => env('DB_HOST', 'localhost'),
            'database' => env('DB_DATABASE', 'chap29'),
            'username' => env('DB_USERNAME', 'myuser'),
            'password' => env('DB_PASSWORD', 'mypass'),
            'charset' => 'UTF-8',
            'collation' => 'utf8_unicode_ci',
            'prefix' => '',
            'strict' => false
        ]
]
…
```

Note the use of the `env()` helper function which is used to provide the ability to overwrite things like the DB username and password through the use of a system-level environment variable. This 'connections' array can contain as many references to various database servers as you wish, and the key (in this case `'mysql'`, not the same as the value for the `'driver'` key) is determined by you. In situations where you may have multiple database

connections you simply need to specify a default connection (see the `'default'` key in this configuration file).

Now that we have set up our MySQL connection, let's take a look at how you can build Eloquent models that use it. For the purposes of example, let's say you have a MySQL database with two tables, books and authors. The actual columns available in these tables are largely irrelevant to this discussion, simply understand that there is a 1-N relationship between authors and the number of books they have written. Thus, one of the columns within the books table must be a foreign reference to a row in the author table.

Defining these sorts of schema definitions in Laravel is done through a database migration workflow available to the Laravel application. This workflow begins with creating a migration class by using the Laravel artisan command line tool as shown:

```
$ php artisan migrate:make create_author_and_books_schema
```

This will create a migration PHP script located in the `app\database\migrations` directory and prefixed with a timestamp. Opening this script you will note it contains a class extending the `Illuminate\Database\Migrations\Migration` class and implements two methods: `up()` and `down()`. These methods are called when applying and reverting a given migration as necessary and should contain any logic needed to do so. In a database migration, typically you will work with the Laravel Schema class, which contains the necessary tools to create, alter, drop, or otherwise modify tables within your database. Below is an example of a basic migration class that will create the schema we previously discussed:

```
<?php

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

class CreateAuthorAndBookSchema extends Migration
{
    public function up()
    {
        Schema::create('authors', function(Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email');
            $table->timestamps();
        });

        Schema::create('books', function(Blueprint $table) {
            $table->increments('id');
            $table->integer('author_id')->unsigned();
            $table->string('title');
            $table->timestamps();
```

```
            $table->foreign('author_id')
                      ->references('id')
                      ->on('authors')
                      ->onDelete('cascade');
    }

    public function down()
    {
        Schema::drop('books');
        Schema::drop('authors');
    }
}
```

Examining the `CreateAuthorAndBookSchema` migration class you can see that in the `up()` method we use `Schema::create()` method to create two tables, `'authors'` and `'books'`. Each call to `Schema::create()` includes as one of its parameters a closure passed as an instance of the Blueprint class, which allows you to specify the details of that table such as columns and foreign key references. One thing of note is the call to the `Blueprint::timestamps()` method in each closure, which automatically creates two columns `'created_at'` and `'updated_at'` for the table. This is because, by default when working with Eloquent models in Laravel, Laravel will automatically add and update these fields as records are written as a matter of convenience and, since disabling them is uncommon, we add them to our schema here.

Once we've created our Schema we can apply the new migration to our database by again using the Laravel artisan command:

```
$ php artisan migrate
```

This will examine the `app\database\migrations` directory for any unapplied migrations and execute each in sequence of creation to the database. With our database schema created, we can now move on to building our classes to interface with that schema using Eloquent.

The whole point behind the concept of object-relational-mapping is simplicity. Thus, creating models in Eloquent for database access reflects this. Where model classes are stored inside of the directory structure of your Laravel application is up to you to decide, but the `app/Models` directory is a good choice in most cases. Using this, here is an example of the Author ORM class which we could put in the `app/Models/Author.php` file:

```
<?php

namespace App\Models;

class Author extends \Eloquent
{
```

```
}
?>
```

Surprisingly enough, that is all you need to enable basic access to the `'authors'` table on your MySQL server. Eloquent will automatically deduce the table name based on the name of the class (the plural form of `'author'`, all lower case) and assume the default database connection as where this schema can be found. Now defined, you can easily insert data into this table using the class for example as follows:

```
$myModel = new Author();
$myModel->name = "John Coggeshall";
$myModel->save();
```

Other operations, such as reading and updating are generally equally as straightforward and will be discussed later in this section.

One thing however that is not defined yet in our Eloquent ORM is the second table in our schema `'books'` and the corresponding relationship between books and authors. To do this, we will need to create a second Eloquent class named Book in similar fashion to the Author class first:

```
<?php
namespace App\Models;

class Book extends \Eloquent
{
}
?>
```

Once we have our model class created we can now start defining the relationship between these two classes in the database. To do this, we simply create a method in each representing the name of the other and returning from that method a relationship object. In our case, the relationship between Authors and Books is a one-to-many relationship and thus for the Author class we will create a `books()` method that returns a t relationship and for the Book class we will create an t method that returns a t relationship as shown:

```
App\Models\Author.php
<?php
namespace App\Models;

class Author extends \Eloquent
{
    public function books()
    {
        return $this->hasMany('Book');
    }
}
?>
```

```
App\Models\Book.php
<?php
namespace App\Models;

class Book extends \Eloquent
{
    public function author()
    {
        return $this->belongsTo('Author');
    }
}
?>
```

CRUD Operations in Eloquent

With the Eloquent model classes defined we are now ready to use them to interact with our database. There are many different syntax options and approaches to using the Eloquent ORM, and we will look at some of the most useful in this section.

As previously shown, fundamentally creating a new record in the database using Eloquent is as straightforward as creating an instance of the model in question, populating it with the relevant data, and calling its `save()` method. The same holds true for updating an existing record, in fact the only real difference between a save and an update to Eloquent is if the primary key for that record is set. If the primary key field in the object is null Eloquent will assume it is an insert/create operation while if it is not null an update operation is assumed:

```
<?php

$myModel = new \App\Models\Author();
$myModel->name = "John Coggeshall";
$myModel->save(); // Will insert a new row

$myModel = new \App\Models\Author();
$myModel->id = 2;
$myModel->name = "Diana Coggeshall";
$myModel->save(); // Will update the record with a ID field of 2 instead of
inserting
```

Retrieving data from the database using Eloquent is equally straightforward for most cases as the model class inherits many useful tools for querying the table. For example, the `all()` method will return all rows from the table:

```
<?php

$authors = \App\Models\Author::all();
```

```
foreach($authors as $author) {
     print "The author name is: {$author->name}";
}

?>
```

Let's take a look at the various querying options available to be used when working with Eloquent. In the below list, the generic Eloquent class name Model is used and should be replaced with a specific Eloquent class reference. This is by no means a complete list of methods available to an Eloquent ORM class – please refer to the Laravel documentation for a complete API:

- `Model::find($key)` – Return a record instance by primary key value
- `Model::where($column, $comparison, $value)` – Return one or more records that match the criteria (i.e. `Model::where('name', '=', 'John')`)
- `Model::whereNull($column)` – Return records with the column `$column` set to the null value
- `Model::whereRaw($conditional, $bindings)` – Perform a `raw` conditional query (for complex queries that can't be produced using another Eloquent method) and provide optional value bindings for that conditional.
- `Model::all()` – Return all records

With the exception of `Model::all()` and `Model::find()`, typically Eloquent distinguishes between the creation of the query and the execution of the query. For example calling the `where()` method alone will not return results from the database. This is done so that you may stack multiple conditionals together into a single query before execution. To actually execute the constructed query and retrieve results you can use the `get()` method (to return multiple results) or `first()` method (to return a single result) as shown:

```
<?php

     $query = Author::where('name', 'LIKE', '%ohn%')
                              ->where('name', 'LIKE', %oggeshall%');

    // Get all of the results
    $results = $query->get();

    // Get the first result
    $result = $query->first();
?>
```

Often it is desirable as well to perform queries against the relationships of an Eloquent model. The full functionality of this is outside of the scope of this section and chapter,

however in principal this can also easily be done by calling the relation as a method and using the same query methods previously introduced. In these situations, Eloquent will automatically include the necessary relational portion of the query:

```
<?
     $author = Author::find(1); // Assume row exists

     $PHPBooks = $author->books()->where('title', 'LIKE', '%PHP%')->get();

?>
```

Final Thoughts on the Laravel Framework

Hopefully this chapter has served as a useful primer in getting to know the workings of the major components of the Laravel framework. In truth, the subject of Laravel is expansive enough to fill an entire book of its own, but the concepts and tools we've introduced here should be sufficient for our needs.

As mentioned before, Laravel also benefits from extensive online documentation, upon which you're strongly encouraged to lean on as you build your own Laravel-based applications beyond what is covered in this introduction. You can find this documentation on the Laravel website at http://laravel.com/docs/.