Laravel

Practical 1

James Ooi

Jan 2018

Contents

1 Introducing Laravel, the PHP Framework for Web Artisans

1.1 About Laravel

Laravel is a free, open-source PHP framework intended for the development of web applications (web apps) by adopting the model-view-controller (MVC) architectural pattern. Created by Taylor Otwell in 2011, Laravel features a modular packaging system with a dedicated dependency manager, different ways of accessing relational databases, utilities that aid in application deployment and maintenance, and its orientation toward syntactic sugar.

Laravel promotes rapid application development. Using Laravel, web application developers can have elegant applications delivered at warp speed.

- *Expressive, beautiful syntax.* Laravel value elegance, simplicity and readability. You may check out Laracasts and the documentation at any time for help.
- *Tailored for your team.* Laravel is tailored for a development team of any size, including if you are a solo developer. It keeps everyone in sync using Laravel's database agnostic migrations and schema builder.
- *Modern toolkit, pinch of magic.* Laravel provides an amazing object-relational mapping (ORM), painless routing, powerful queue library and simple authentication. It gives developers the tools needed for modern, maintainable PHP code.

1.2 Web Application Framework

A web application framework (WAF) is a software framework that is used for the development of web apps, including but not limited to web services, web resources and web APIs. A WAF provides a standard way of building and deploying web apps. These frameworks aim to automate the overhead associated with common activities and features of web development, such as database access, templating, session management, authentication and authorization. The use of WAF in web development also promote code reuse.

1.3 Key Features of Laravel

- *Bundles* provides a modular packaging system with bundled features readily available for easy addition to web apps. Laravel also uses *Composer*, a PHP dependency manager to add framework-agnostic and Laravel-specific PHP packages available from the *Packagist* repository.

- *Eloquent ORM* (object-relational mapping), an advanced PHP implementation of the active record pattern, provides internal methods for enforcing constraints on the relationships between database objects. Following the active record pattern, Eloquent ORM presents tables as classes and instances of these classes represent the respective rows of tables.
- *Query builder* provides a direct database access alternative to Eloquent ORM. It provides a set of classes and methods to build SQL queries programmatically instead of writing raw SQL statements.
- *Application logic* is an integral part of developed web apps, implemented either using controllers or as part of route declarations.
- *Reverse routing* defines a relationship between the links and routes, making it possible for later changes to routes to be automatically propagated into relevant links. When the links are created using names of existing routes, the appropriate uniform resource identifiers (URIs) are automatically created by Laravel.
- *Restful controllers* provide an optional way for separating the logic behind serving HTTP GET and POST requests.
- *Class auto loading* provides automated loading of PHP classes without the need for manual maintenance of inclusion paths. On-demand loading prevents inclusion of unnecessary components, so only the actually used components are loaded.
- *Blade templating engine* combines one or more templates with a data model to produce resulting views, doing that by transpiling the templates into cached PHP code for improved performance. Blade also provides a set of its own control structures such as conditional statements and loops, which are internally mapped to PHP. Laravel services may also be called from Blade templates, and the templating engine itself can be extended with custom directives.
- *Migrations* provide a version control system for database schemas, making it possible to associate changes in the application's codebase and required changes in the database structure. This provides a simplified deployment and updating of Laravel web apps.
- *Automatic pagination* simplfies the task of implementing pagination, replacing the usual manual implementation approaches with automated methods integrated into Laravel.
- *Form request* serves as the base for form input validation by internally binding event listeners, resulting in automated invoking of form validation methods and generation of the actual form.

1.4 Artisan CLI

Laravel provides *Artisan*, a command-line interface (CLI) that features a number of helpful commands that can assist developers in building web apps. The features of Artisan are mapped to different subcommands of the **artisan** command-line utility, providing functionality that aids in managing and building Laravel web apps. Common uses of Artisan include managing database migrations and seeding and publishing package assets. Artisan is also used for generating biolerplate code for new controllers and migrations. This frees the developer from creating proper code skeletons.

Developers can also create new Artisan functionalities and capabilities by implementing new custom commands, for example, to automate application-specific recurring tasks or to migrate data from an old system to a new system.

2 Getting Started with Laravel

2.1 Laravel Version

In this course, we will begin setting up web applications using Laravel 5.5. While Laravel provide major releases every 6 months in February and August, not all major releases include long *long term support (LTS)*. Laravel 5.5 provides for LTS. The previous Laravel release with LTS is Laravel 5.1. A LTS release means Laravel will provide bug fixes for 2 years and security fixes for 3 years via minor releases. For general releases such as Laravel 5.2, 5.3 and 5.4, bug fixes are only provided for 6 months while security fixes are provided for 1 year.

2.2 Server Requirements

The machine hosting the web server for your Laravel development (and production) must fulfill the following requirements:

- PHP $\geq$ 7.0
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

2.3 Installing Laravel

Laravel utilizes *Composer*, a PHP Dependency Manager to manage its dependencies. Before using Laravel, ensure that you have installed Composer.

Assuming that you want to create your Laravel project directory named **myapp** in **D:\Laravel Practicals**, open a Command Prompt window and change the working directory to **D:\Laravel Practicals**. Issue the Composer create-project command:

```
composer create-project –prefer-dist laravel/laravel myapp ”5.5.*”
```

The above command will download the latest release of Laravel 5.5 and create a Laravel project directory in **D:\Laravel Practicals\myapp**. This process takes several minutes depending on your Internet connection speed, so sit back and relax while waiting the completion of the installation task.

2.4 Configuring Laravel

2.4.1 Web Server Configuration

After installing Laravel, you should configure your web server web root directory to the **public** directory of your Laravel project. The **public** directory contains the **index.php** file which serves as the front controller for all HTTP requests entering your Laravel app.

In our practicals, we will create a virtual host on HTTP port 8088 on your computer's Apache web server as follows:

- Open the **httpd-vhosts.conf** file of your Apache web server and add the following lines at the end of the file:

```
<VirtualHost *:8088>
        ServerName localhost
        DocumentRoot "D:/Laravel Practicals/myapp/public"
        <Directory  "D:/Laravel Practicals/myapp/public">
                Options +Indexes +Includes +FollowSymLinks +MultiViews
                AllowOverride All
                Require local
        </Directory>
</VirtualHost>
```

- Open the **httpd.conf** file of your Apache web server. Search for where the **Listen** directives are placed in the file and add the following lines after the last **Listen** directive in the file:

```
Listen 0.0.0.0:8088
Listen [::0]:8088
```

- Restart your Apache web server for the changes to take effect.
- Open the URL **http://localhost:8088** in your web browser. If you can see a default Laravel app home page loaded, your Laravel installation and web server configuration has been completed successfully.

2.4.2 Configuration Files

All the configuration files of your Laravel app are located in the **config** directory of your Laravel project directory. If you browse the directory, you will notice the following configuration files already pre-created:

- **app.php**
- **auth.php**
- **broadcasting.php**

- **cache.php**
- **app.php**
- **database.php**
- **filesystems.php**
- **mail.php**
- **queue.php**
- **services.php**
- **session.php**
- **view.php**

Each option in each file is documented. You may take a look at the files and get familiar with the options available to you.

2.4.3 Directory Permissions

Ensure that the **storage** and **bootstrap/cache** directories are writable by the web server or Laravel will not run. This is especially true if you are working on Linux systems.

2.4.4 Application Key

You should set an application key for your Laravel project upon installation. Typically, the application key is a 32-character long key and is set in the **.env** environment file located in your Laravel project directory.

This should have already been set for you since you have created your Laravel project using Composer, so you can safely skip this step. You may view the application key by looking at the **.env** environment file as follows:

```
APP_KEY=base64:LFjeRqPithDxBQ6QdxaORwc0Fmfjvz6NVeMoQaeZ/XI=
```

Take note that the value of your **APP_KEY** is very likely to be different that what is shown above.

It is important that the application key is set. Otherwise, your user sessions and other encrypted data will not be secure!

2.4.5 Enabling Pretty URLs

Ensure that the **public/.htaccess** file has been set with following Apache directives:

```
<IfModule mod_rewrite.c>
    <IfModule mod_negotiation.c>
        Options -MultiViews -Indexes
    </IfModule>

    RewriteEngine On

    # Handle Authorization Header
    RewriteCond %{HTTP:Authorization} .
    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]

    # Redirect Trailing Slashes If Not A Folder...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_URI} (.+)/$
    RewriteRule ^ %1 [L,R=301]

    # Handle Front Controller...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]
</IfModule>
```

Also ensure that **mod_rewrite** is installed and enabled in your Apache web server.

2.4.6 Database

As you will be using a relational database management system (RDBMS) in your Laravel project, you need to create a database and configure it in your Laravel project. For all our practical session, we will be using MySQL. To do so, create a database named **laravel_app** using any MySQL client such as PhpMyAdmin. Ensure that you set the database collation to **utf8_general_ci** when creating the database.

Once you have created the database, open the **.env** environment file and modify the following values with your MySQL database settings:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel_app
DB_USERNAME=dbuser
DB_PASSWORD=dbpass
```

Please note that the database username and password may be different in your MySQL installation.

Our application uses the **utf8** character set and the **utf8_general_ci** collation. Open the **database.php** file in the **config** directory and make the following changes:

```
'mysql' => [
    // ...

    'charset' => 'utf8',
    'collation' => 'utf8_general_ci',

    // ...
],
```

3 Your First App

Let's begin some coding tasks before going through the topic on the *Directory Structure* of your Laravel project. Throughout the practicals, we will be developing an app to manage membership registration for Travelife Club, a travel club with divisions at various cities within Malaysia. A member joins the travel club at a particular division and each member can join one or more interest groups set up within the club. Staff members at the club offices are users of this app to maintain the divisions, groups and membership details.

3.1 Prepping the Database

From the above scenario, we can see that we have 4 entities for this app:

- User
- Division
- Member
- Group

3.1.1 Database Migrations

We shall begin creating the database tables required for this app.

3.1.1.1 The *users* Table

As we are going to allow staff members (users) to maintain the divisions and membership details, we will need a table to store all users. Laravel already ships with a migration to create a basic **users** table, so we do not need to manually generate one. The default migration for the **users** table is the file **2014_10_12_000000_create_users_table.php** located in the **database/migrations** directory.

```
<?php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create(’users’, function (Blueprint $table) {
            $table->increments(’id’);
            $table->string(’name’);
            $table->string(’email’)->unique();
            $table->string(’password’);
            $table->rememberToken();
            $table->timestamps();
        });
    }

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

As we want to use the *InnoDB Storage Engine* for our table, we need to add the code as shown in Line 17 in the code listing below:

```
 1 <?php
 2 
 3 use Illuminate\Support\Facades\Schema;
 4 use Illuminate\Database\Schema\Blueprint;
 5 use Illuminate\Database\Migrations\Migration;
 6 
 7 class CreateUsersTable extends Migration
 8 {
 9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::create(’users’, function (Blueprint $table) {
17             $table->engine = ’InnoDB’;
18             $table->increments(’id’);
19             $table->string(’name’);
20             $table->string(’email’)->unique();
21             $table->string(’password’);
22             $table->rememberToken();
```

```
            $table->timestamps();
        });
    }

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

3.1.1.2 The *divisions* Table

We shall now build a table to hold the *divisions* of the club. We will be using the *Artisan CLI* to generate a template of the migration script for our **divisions** table. Take note that the *Artisan CLI* can be used to generate a variety of classes and will save you a lot of typing as you build your Laravel projects.

To generate a migration for the **divisions** table, issue the following **make:migration** Artisan CLI command:

```
php artisan make:migration create_divisions_table --create=divisions
```

This will create a migration file **2017_12_08_033403_create_divisions_table.php**. Please take note that the name of your migration file will be different as the date and time the migration is created is part of the filename. You will notice that the filename begins with the year, month, and day, separated by underscores, followed by the UTC time in **hhmmss** format before the **create_divisions_table.php** name.

The resulting file is as follows:

```
<?php

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

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

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

You will notice that in the **up** method of the **CreateDivisionsTable** class, three columns have already been defined:

- **id** The *primary key* column which is an auto-increment unsigned integer.
- The **timestamps** method will create two columns named **created_at** and **updated_at** in the table, specifying the date and time a row is created and updated respectively.

Now, we need to build on the **up** method to add more columns required for the **divisions** table. We will add the following columns:

- **code** Code of the division. This will be defined as a char with a length of 3 characters. Equivalent to SQL **CHAR(3)**. This column must be unique.
- **name** Name of the division. This will be defined as a string with a maximum length of 100 characters. Equivalent to SQL **VARCHAR(100)**.
- **address** Street address of the division office. This will be defined as a text, equivalent to Equivalent to SQL **TEXT**.
- **postcode** Postcode of the division office. This will be defined as a string with a maximum length of 5 characters. Equivalent to SQL **VARCHAR(5)**.
- **city** City of the division office. This will be defined as a string with a maximum length of 50 characters. Equivalent to SQL **VARCHAR(50)**.
- **state** State of the division office. This will be defined as a char with a length of 2 characters. Equivalent to SQL **CHAR(2)**. We will use 2-digit code to represent the states and federal territories in Malaysia for this column.

In addition, the **name**, **postcode**, **city** and **state** columns must be indexed for more efficient querying.

First of all, we will be using *InnoDB Storage Engine* for our table. Hence we need to add the code as shown in Line 17 in the code listing below. To define the abovementioned additional columns, we add the code as shown in lines 19 to 24 in the code listing below:

```php
 1 <?php
 2 
 3 use Illuminate\Support\Facades\Schema;
 4 use Illuminate\Database\Schema\Blueprint;
 5 use Illuminate\Database\Migrations\Migration;
 6 
 7 class CreateDivisionsTable extends Migration
 8 {
 9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::create('divisions', function (Blueprint $table) {
17             $table->engine = 'InnoDB';
18             $table->increments('id');
19             $table->char('code', 3)->unique();
20             $table->string('name', 100)->index();
21             $table->text('address')->nullable();
22             $table->string('postcode', 5)->index();
23             $table->string('city', 50)->index();
24             $table->char('state', 2)->index();
25             $table->timestamps();
26         });
27     }
28 
29     /**
30      * Reverse the migrations.
31      *
32      * @return void
33      */
34     public function down()
35     {
36         Schema::dropIfExists('divisions');
37     }
38 }
```

3.1.1.3 The *members* Table

Next, we will create the **members** table. Once again, we will be using the *Artisan CLI* to generate a migration:

```
php artisan make:migration create_members_table --create=members
```

In addition to the primary key and timestamp columns, we need to add the following columns to the **members** table:

- **membership_no** The membership number. This will be defined as a char with a length of 10 characters. Equivalent to SQL **CHAR(10)**. This column must be unique.
- **nric** NRIC number of the member. This will be defined as a char with a length of 12 characters. Equivalent to SQL **CHAR(12)**.
- **name** Name of the member. This will be defined as a string with a maximum length of 100 characters. Equivalent to SQL **VARCHAR(100)**.
- **gender** Gender of the member. This will be defined as a char with a length of 1 character. Equivalent to SQL **CHAR(1)**.
- **address** Street address of the member's home. This will be defined as a text, equivalent to Equivalent to SQL **TEXT**.
- **postcode** Postcode of the member's home. This will be defined as a string with a maximum length of 5 characters. Equivalent to SQL **VARCHAR(5)**.
- **city** City of the member's home. This will be defined as a string with a maximum length of 50 characters. Equivalent to SQL **VARCHAR(50)**.
- **state** State of the member's home. This will be defined as a char with a length of 2 characters. Equivalent to SQL **CHAR(2)**. We will use 2-digit code to represent the states and federal territories in Malaysia for this column.

The **nric**, **name**, **gender**, **postcode**, **city** and **state** columns must be indexed for more efficient querying.

Once again, we will be using the *InnoDB Storage Engine*. We add code as shown in Line 17 and Lines 19 to 25 as shown in the code listing below.

```
 1 <?php
 2 
 3 use Illuminate\Support\Facades\Schema;
 4 use Illuminate\Database\Schema\Blueprint;
 5 use Illuminate\Database\Migrations\Migration;
 6 
 7 class CreateMembersTable extends Migration
 8 {
 9     /**
10      * Run the migrations.
11      *
12      * @return void
```

```
     */
    public function up()
    {
        Schema::create('members', function (Blueprint $table) {
            $table->engine = 'InnoDB';
            $table->increments('id');
            $table->char('membership_no', 10)->unique();
            $table->char('nric', 12)->index();
            $table->string('name', 100)->index();
            $table->text('address')->nullable();
            $table->string('postcode', 5)->index();
            $table->string('city', 50)->index();
            $table->char('state', 2)->index();
            $table->timestamps();
        });
    }

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

3.1.1.4 Running the Migration

We will put aside the **groups** table for the time being and will be revisiting it in later practical session.

Now, we are ready to run the migration to create the 3 tables: **users**, **divisions** and **groups** in our database. Once again, we will be using the Artisan CLI, this time to run the migration.

Enter and run the following command:

```
php artisan migrate
```

This will run the migration. When the migration process is completed, you will notice the following tables created:

- **migrations** This table is used by the Laravel migration process to maintain the migrations history.
- **users** This table is created from the migration for **users** table discussed earlier.

- **password_resets** This table is created from the migration for **password_resets** which is shipped with Laravel. We will revisit the usage of this migration in later practical sessions.
- **divisions** This table is created from the migration for **divisions** table discussed earlier.
- **members** This table is created from the migration for **members** table discussed earlier.

3.2 Defining the Models

Eloquent is Laravel's default object-relational mapper (ORM). Eloquent makes it easy to retrieve and store data in the database using clearly defined *models*. Usually, each Eloquent model corresponds directly with a single table in the database.

In this practical session, we will be focusing on creating the models for the **divisions** and **members** table. The model classes will be named **Division** and **Member** respectively. Do take note that in Laravel, by default, table names take the plural form while model class names take the singular form. This, however, can be easily changed.

For consistency, we shall stick to the default Laravel convention.

3.2.1 The *User* model

Laravel already ships with a **User** model, so we do not need to generate one manually.

3.2.2 The *Division* model

Now, you need to define a **Division** model that corresponds to the **divisions** table that was created via the migration process earlier. Once again, we will be using the Artisan CLI to generate this model using the **make:model** command.

```
php artisan make:model Division
```

The model will be placed in the **app** directory of your application. By default, the model class is empty. We do not have to explicitly tell the Eloquent model which table it corresponds to because it will assume the database table is the plural form of the model name. So, in this case, the **Division** model corresponds with the **divisions** table.

We will be adding a few lines of code in the **Division** model. For a start, we will define the attributes that are *mass-assignable* as shown in Lines 9 to 21 in the code listing below:

```
1 <?php
2
3 namespace App;
4
5 use Illuminate\Database\Eloquent\Model;
```

```

class Division extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = [
        ’code’,
        ’name’,
        ’address’,
        ’postcode’,
        ’city’,
        ’state’,
    ];
}
```

3.2.2.1 What is *Mass-assignable*?

Mass assignable attributes are attributes that you can set in a single statement. We will be revisitting this topic in later sections.

3.2.3 The *Member* model

Next, you need to define a **Member** model that corresponds to the **members** table.

Again, we will use the Artisan CLI to enter the following command:

```
php artisan make:model Member
```

The code listing below shows the **Member** class with Lines 9 to 22 added manually to define the mass-assignable attributes.

```
 1 <?php
 2
 3 namespace App;
 4
 5 use Illuminate\Database\Eloquent\Model;
 6
 7 class Member extends Model
 8 {
 9     /**
10      * The attributes that are mass assignable.
11      *
12      * @var array
13      */
14     protected $fillable = [
        'membership_no',
        'nric',
        'name',
        'address',
        'postcode',
        'city',
        'state',
    ];
}
```

3.3 Defining Controllers

By now, you would have figured out that you will need to retrieve and store data about divisions and members. We will need to create two controllers, namely the **DivisionController** and **MemberController**. Using the Artisan CLI, enter the following commands to generate both controllers:

```
php artisan make:controller DivisionController

php artisan make:controller MemberController
```

The controller files are generated in the **app/Http/Controllers** directory.

We shall begin working on the **DivisionController**, i.e. the file **DivisionController.php**. You will notice that just like the models, the controller file is empty when generated.

For a start, we will add functionality to our **DivisionController** to display an HTML form to accept user input and to store the input into the database. Create the following two methods in the **DivisionController**.

- **create** Displays an HTML form to accept user input.
- **store** Stores the user input into the database.

The code listing below shows the **DivisionController** with the abovementioned methods which are now empty.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class DivisionController extends Controller
{
    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {

    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     *
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {

    }
}
```

3.4 Defining Routes

Now, we need to add routes to our application. Routes are used to point URLs to controllers methods that should be executed when a user accesses a given URL. By default, all routes for web access are defined in the **routes/web.php** file that is included in every Laravel project. There are 3 other routing files that we will discuss in further practical sessions.

We now need to define two routes, one for each of the methods in the **DivisionController** - **create** and **store**. This is to allow the user to access to the methods in the controller by specifying a URL in the web client.

We will need to append the following statements to the **routes/web.php** routing file to define two new routes to our application:

```
Route::get('/division/create', 'DivisionController@create')
        ->name('division.create');
Route::post('/division/store', 'DivisionController@store')
        ->name('division.store');
```

The first route defines that the URL **/division/create** is an HTTP GET request that is mapped to the **create** method of the **DivisionController**. The route is also assigned with the name **division.create** which is handy for us to refer to it in our code later on.

The second route defines that the URL **/division/store** is an HTTP POST request that is mapped to the **store** method of the **DivisionController**. The route is assigned with the name **division.store**.

3.5 Constructing Layouts & Views

At this point, we are focusing on building functionality to allow user to input new records for divisions. Of course we will eventually include functionalities to edit, delete and list existing division records as well as similar functioanlities for the other entities. In this section, we will be looking at creating *views* for our application. We will be creating a *view* that includes an *HTML Form* which will be displayed when the user access the application using the URL **/division/create**. Upon submitting the form, the URL **/division/store** will be invoked to insert a new division row into the table, using the **Division** model that we created earlier.

3.5.1 Blade Layouts

Before creating the view file to display a form, it is important to know that most, if not all web applications share the same layout across all pages. Typical components in a layout include company logo, navigation bars, headers, sidebars and footers. Fortunately, Laravel makes it easy for us to provide a standard layout across all pages using *Blade Layout*.

All Laravel view files are stored in the directory **resources/views**. A common practice is to put all layout files within a subdirectory named **layouts** in **resources/views**. You will need to create this directory as it is not present by default.

Let's begin by creating a new layout file named **resources/views/layouts/app.blade.php**. The **.blade.php** extension instructs Laravel to use the *Blade templating engine* to render the view. You may use plain PHP templates, however Blade provides convenient short-cuts for writing cleaner, terse templates.

The code listing below shows the **app.blade.php** layout file. Take note that we will adhere to HTML5 in our view and layout templates.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Travelife Club</title>

        <!-- CSS And JavaScript -->
    </head>

    <body>
        <div class="container">
            <nav class="navbar␣navbar-default">
                <!-- Navbar Contents -->
            </nav>
        </div>

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

Obviously, the above code listing is not yet complete. Let's not worry about the header, footer, sidebar and navigation bar for now as our focus is on getting the functionality done first.

Note the**@yield('content')** portion of the layout. This is a special Blade directive that specifies where all child pages that extend the layout can inject their own content.

3.5.2 Child View

3.5.2.1 Laravel Collective Forms & HTML

We can create view files simply using HTML code with Blade directives. However, to ensure consistency of HTML code, we will be using the *Laravel Collective Forms & HTML* package. The *Laravel Collective* (**https://laravelcollective.com/**) is a community organization designed to maintain components that have been removed from the Laravel Framework core, one of which is the *Forms & HTML* package.

Install the *Laravel Collective Forms & HTML* package using *Composer*:

```
composer require "laravelcollective/html":"^5.4.0"
```

Next, you will need to edit your **config/app.php** as follows:

- Add the **Collective\Html\HtmlServiceProvider** to the **providers** array.

```
'providers' => [
    // ...
    Collective\Html\HtmlServiceProvider::class,
    // ...
  ],
```

- Add the class aliases for **Collective\Html\FormFacade** and **Collective\Html\HtmlFacade** to the **aliases** array.

```
'aliases' => [
    // ...
      'Form' => Collective\Html\FormFacade::class,
      'Html' => Collective\Html\HtmlFacade::class,
    // ...
  ],
```

For a complete guide on using the *Laravel Collective Forms & HTML* package, please refer to **https://laravelcollective.com/docs/master/html**.

3.5.2.2 Creating View

We shall now define our child view that contains a form to create a new division. Create a view file in **resources/views/divisions/create.blade.php**. You will need to create the subdirectory

divisions which is not exist yet. Note that we use the convention where the **divisions** directory corresponds to the **DivisionController** while the filename **create.blade.php** corresponds to the **create** method.

The code listing below is our **create.blade.php** view file. We'll skip over some of the Bootstrap CSS boilerplate and only focus on the things that matter.

```
<?php

use App\Common;

?>
@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- New Division Form -->
        {!! Form::model($division, [
            'route' => ['division.store'],
            'class' => 'form-horizontal'
        ]) !!}

            <!-- Code -->
            <div class="form-group␣row">
                {!! Form::label('division-code', 'Code', [
                    'class'         => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::text('code', null, [
                        'id'        => 'division-code',
                        'class'     => 'form-control',
                        'maxlength' => 3,
                    ]) !!}
                </div>
            </div>

            <!-- Name -->
            <div class="form-group␣row">
                {!! Form::label('division-name', 'Name', [
                    'class'         => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::text('name', null, [
                        'id'        => 'division-name',
                        'class'     => 'form-control',
                        'maxlength' => 100,
                    ]) !!}
                </div>
            </div>

            <!-- Address -->
            <div class="form-group␣row">
                {!! Form::label('division-address', 'Address', [
                    'class'          => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::textarea('address', null, [
                        'id'         => 'division-address',
                        'class'      => 'form-control',
                    ]) !!}
                </div>
            </div>

            <!-- Postcode -->
            <div class="form-group␣row">
                {!! Form::label('division-postcode', 'Postcode', [
                    'class'          => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::text('postcode', null, [
                        'id'         => 'division-postcode',
                        'class'      => 'form-control',
                        'maxlength' => 5,
                    ]) !!}
                </div>
            </div>

            <!-- City -->
            <div class="form-group␣row">
                {!! Form::label('division-city', 'City', [
                    'class'          => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::text('city', null, [
                        'id'         => 'division-city',
                        'class'      => 'form-control',
                        'maxlength' => 50,
                    ]) !!}
                </div>
            </div>

            <!-- State -->
            <div class="form-group␣row">
                {!! Form::label('division-state', 'State', [
                    'class'          => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
```

```
 94                     {!! Form::select(’state’, Common::$states, null, [
 95                         ’class’          => ’form-control’,
 96                         ’placeholder’    => ’-␣Select␣State␣-’,
 97                     ]) !!}
 98                 </div>
 99             </div>
100
101             <!-- Submit Button -->
102             <div class="form-group␣row">
103                 <div class="col-sm-offset-3␣col-sm-6">
104                     {!! Form::button(’Save’, [
105                         ’type’           => ’submit’,
106                         ’class’          => ’btn␣btn-primary’,
107                     ]) !!}
108                 </div>
109             </div>
110         {!! Form::close() !!}
111     </div>
112
113 @endsection
```

Note that in the above view file, we have specified to **use App\Common** in Line 3 and have referred to a static property **states** in Line 94. The class **App\Common** is not yet defined. We will define this class in the **app** directory as follows:

```
<?php

namespace App;

class Common
{
    public static $states = [
        ’01’    => ’Johor’,
        ’02’    => ’Kedah’,
        ’03’    => ’Kelantan’,
        ’14’    => ’Kuala␣Lumpur’,
        ’15’    => ’Labuan’,
        ’04’    => ’Melaka’,
        ’05’    => ’Negeri␣Sembilan’,
        ’06’    => ’Pahang’,
        ’07’    => ’Penang’,
        ’08’    => ’Perak’,
        ’09’    => ’Perlis’,
        ’16’    => ’Putrajaya’,
        ’12’    => ’Sabah’,
        ’13’    => ’Sarawak’,
        ’10’    => ’Selangor’,
        ’11’    => ’Terengganu’,
    ];
}
```

In many applications, we will encounter situations where users will be asked to make selections from radio buttons, checkboxes and dropdown lists. In some cases, the data for the selections will be extracted from a database table or file. In other cases, the selections will be hardcoded, particularly values that are fixed, such as gender. Values for states of a country or a complete list of countries seldom change, and only do so when there are major political developments. In the case of Malaysia, since the federation was established in September 1963, the list of states have only changed three times. They are the result of the creation of three Federal Territories, namely Kuala Lumpur (February 1974), Labuan (April 1984) and Putrajaya (February 2001).

A common practice to hardcode such values is to define them in a file as arrays. In our case, we define a class **use App\Common** with the value for states in a static array named **states**.

3.5.3 Rendering the View

Now that we have completed the Blade view templates, we need to render it. In the **Division-Controller.php** file, add the following **use** statement after the last **use** statement as we will be referring to the **app\Division** class in code.

```
use App\Division;
```

Well, in fact, the order doesn't matter, but it is recommended to group all the **use** statements together in your code. If there are many **use** statements, you may want to have them sorted in a certain way, but let's not worry about that for now.

Now, we modify the code for the **create** method as follows:

```
/**
 * Show the form for creating a new resource.
 *
 * @return \Illuminate\Http\Response
 */
public function create()
{
    $division = new Division();

    return view('divisions.create', [
        'division'     => $division,
    ]);
}
```

Note that in the **create** method, we instantiate an instance of the **Division** model class and inject it to the view we defined earlier using the **view** method. Returning it causes the output to be sent to the client for rendering. The first parameter of the **view** method specifies the view file. In this case, **division.create** means render the view **create.blade.php** located in the **divisions** directory. The second parameter specifies an array of variables to be injected to the view, in this case the variable **$division**. The key **division** means we can access it within the view using the variable **$division** as ween in Line 14 of the **create.blade.php** view file we defined earlier.

3.5.3.1 Displaying Page

By now, if you have followed every instructions given, you should be able to open the web page that displays the abovementioned view within the layout specified. Open your web browser and enter the URL **http://localhost:8088/division/create**. This should render the web page that includes the form we defined.

3.6 Storing User Input

Remember that in our view **create.blade.php**, we have defined the HTML form where the *action* points to the route **division.store**.

```
{!! Form::model($division, [
    'route' => ['division.store'],
    'class' => 'form-horizontal'
]) !!}
```

Now, in the **store** method of our **DivisionController** we write the following code to insert a new row and redirect the user to the route **division.index**. Note that we have not defined the route **division.index**. We will add that after we confirm that the row insertion works. Also take note that we have yet to implement data validation to validate user input.

```
/**
 * Store a newly created resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 *
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $division = new Division;
    $division->fill($request->all());
    $division->save();

    return redirect()->route('division.index');
}
```

Now, open your web browser and enter the URL **http://localhost:8088/division/create**, enter data to all the input fields and click on the **Save** button. You should be getting an **InvalidArgumentException: Route [division.index] not defined** error. Don't worry about this error for the time being as we have not yet define the route **division.index**. Go to your MySQL client and verify that the data you have entered is successfully saved in the **divisions** table. Do take note that both the timestamp columns, **created_at** and **updated_at** are automatically populated based on the server's clock in UTC time zone.

Once you have completed the above steps, try entering another four records for divisions before proceeding.

3.7 Listing & Viewing

In this section, we will be building functionalities for listing all divisions and showing data of a single division. Before proceeding, we need to add the following routes to the **routes/web.php** file:

```
1 Route::get('/division', 'DivisionController@index')->name('division.index');
2 Route::get('/division/show/{id}',
3     'DivisionController@show')->name('division.show');
```

In lines 2-3, a parameter **division** is specified to pass the **id** (primary key) of the division as a URL parameter.

3.7.1 Listing

We shall now define a method named **index** in our **DivisionController**. In this method, we retrieve all the divisions by **name** column in ascending order and inject the model instances to the **divisions.index** view file.

```
/**
 * Display a listing of the resource.
 *
 * @return \Illuminate\Http\Response
 */
public function index()
{
    $divisions = Division::orderBy('name', 'asc')->get();

    return view('divisions.index', [
        'divisions' => $divisions
    ]);
}
```

Now, create the view file **resources/views/divisions/index.blade.php** and enter the following:

```
<?php

use App\Common;

?>
@extends('layouts.app')

@section('content')
    <!-- Bootstrap Boilerplate... -->
    <div class="panel-body">
        @if (count($divisions) > 0)
            <table class="table␣table-striped␣task-table">
                <!-- Table Headings -->
                <thead>
                    <tr>
                        <th>No.</th>
                        <th>Code</th>
                                <th>Name</th>
                                <th>City</th>
                                <th>State</th>
                                <th>Created</th>
                        <th>Actions</th>
                    </tr>
                </thead>

                <!-- Table Body -->
                <tbody>
                    @foreach ($divisions as $i => $division)
                        <tr>
                            <td class="table-text">
                                <div>{{ $i+1 }}</div>
                            </td>
                            <td class="table-text">
                                <div>
                           {!! link_to_route(
                                        'division.show',
                                        $title = $division->code,
                                        $parameters = [
                                            'id' => $division->id,
                                        ]
                                    ) !!}
                                </div>
                            </td>
                            <td class="table-text">
                                <div>{{ $division->name }}</div>
                            </td>
                            <td class="table-text">
                                <div>{{ $division->city }}</div>
                            </td>
                            <td class="table-text">
                                <div>{{ Common::$states[$division->state] }}</div>
                            </td>
                            <td class="table-text">
                                <div>{{ $division->created_at }}</div>
                            </td>
                            <td class="table-text">
                                <div>
                                    {!! link_to_route(
                                            'division.edit',
                                            $title = 'Edit',
                                            $parameters = [
                                                'id' => $division->id,
63                                              ]
64                                      ) !!}
65                                  </div>
66                              </td>
67                          </tr>
68                      @endforeach
69                  </tbody>
70              </table>
71          @else
72              <div>
73                  No records found
74              </div>
75          @endif
76      </div>
77  @endsection
```

In lines 35-41 and 58-64, the **link_to_route** is a helper function that is provided by the *Laravel Collective Forms & HTML* package that we have installed and used earlier. Note the parameters:

- **division.show**: The route name.
- **$title**: The text that will appear in the hyperlink. In this case, the value provided by **$division->code** is provided.
- **$parameters**: This specifies the URL parameters, if applicable. In this case, we provide the **$division->id**.
- **$attributes**: We can configure HTML attributes of the hyperlink by providing the key-value pairs as an associative array. In our example, we do not use it.

For lines 58-64, the link is provided for editing which will be covered in the next sub-section.

Once you have completed the above, enter the URL enter the URL **http://localhost:8088/division**. You should see a table listing of division records that you have entered.

3.7.2 Viewing

We shall now define a method named **show** in our **DivisionController**. In this method, we retrieve a single division by **id** column and inject the model instance to the **divisions.show** view file.

```

// ...

use Illuminate\Database\Eloquent\ModelNotFoundException;

// ...
```

```

class DivisionController extends Controller
{
    // ...

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     *
     * @return \Illuminate\Http\Response
     */
     public function show($id)
     {
         $division = Division::find($id);
         if(!$division) throw new ModelNotFoundException;

         return view('divisions.show', [
             'division' => $division
         ]);
     }

    // ...
}
```

Now, create the view file **resources/views/divisions/show.blade.php** and enter the following:

```
<?php

use App\Common;

?>
@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <table class="table␣table-striped␣task-table">
            <!-- Table Headings -->
            <thead>
                <tr>
                    <th>Attribute</th>
                    <th>Value</th>
                </tr>
            </thead>
            <!-- Table Body -->
            <tbody>
```

```
                    <tr>
                        <td>Code</td>
                        <td>{{ $division->code }}</td>
                    </tr>
                    <tr>
                        <td>Name</td>
                        <td>{{ $division->name }}</td>
                    </tr>
                    <tr>
                        <td>Address</td>
                        <td>{!! nl2br($division->address) !!}</td>
                    </tr>
                    <tr>
                        <td>Postcode</td>
                        <td>{{ $division->postcode }}</td>
                    </tr>
                    <tr>
                        <td>City</td>
                        <td>{{ $division->city }}</td>
                    </tr>
                    <tr>
                        <td>State</td>
                        <td>{{ Common::$states[$division->state] }}</td>
                    </tr>
                    <tr>
                        <td>Created</td>
                        <td>{{ $division->created_at }}</td>
                    </tr>
                </tbody>
            </table>
        </div>

@endsection
```

Go to the URL **http://localhost:8088/division** and click on the division code hyperlinks to view the record for individual divisions.

3.8 Update Existing Data

What if there are changes to data already entered? We will need to provide users with the ability to update existing data. To begin, we shall add another two routes to our application, one for displaying the editing form and another to execute the update operation.

```
Route::get('/division/edit/{id}',
    'DivisionController@edit')->name('division.edit');
Route::post('/division/update/{id}',
    'DivisionController@update')->name('division.update');
```

3.8.1 Editing Form

We will define a method named **edit** in our **DivisionController**

```
/**
 * Show the form for editing the specified resource.
 *
 * @param  int  $id
 *
 * @return \Illuminate\Http\Response
 */
public function edit($id)
{
    $division = Division::find($id);
    if(!$division) throw new ModelNotFoundException;

    return view('divisions.edit', [
        'division' => $division
    ]);
}
```

Now, create the view file **resources/views/divisions/edit.blade.php** and enter the following:

```
<?php

use App\Common;

?>
@extends('layouts.app')

@section('content')

    <!-- Bootstrap Boilerplate... -->

    <div class="panel-body">
        <!-- New Division Form -->
        {!! Form::model($division, [
            'route' => ['division.update', $division->id],
            'class' => 'form-horizontal'
        ]) !!}

            <!-- Code -->
            <div class="form-group␣row">
                {!! Form::label('division-code', 'Code', [
                    'class'         => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::text('code', $division->code, [
                        'id'        => 'division-code',
                    'class'     => 'form-control',
                    'maxlength' => 3,
                ]) !!}
            </div>
        </div>

        <!-- Name -->
        <div class="form-group␣row">
            {!! Form::label('division-name', 'Name', [
                'class'         => 'control-label␣col-sm-3',
            ]) !!}
            <div class="col-sm-9">
                {!! Form::text('name', $division->name, [
                    'id'        => 'division-name',
                    'class'     => 'form-control',
                    'maxlength' => 100,
                ]) !!}
            </div>
        </div>

        <!-- Address -->
        <div class="form-group␣row">
            {!! Form::label('division-address', 'Address', [
                'class'         => 'control-label␣col-sm-3',
            ]) !!}
            <div class="col-sm-9">
                {!! Form::textarea('address', $division->address, [
                    'id'        => 'division-address',
                    'class'     => 'form-control',
                ]) !!}
            </div>
        </div>

        <!-- Postcode -->
        <div class="form-group␣row">
            {!! Form::label('division-postcode', 'Postcode', [
                'class'         => 'control-label␣col-sm-3',
            ]) !!}
            <div class="col-sm-9">
                {!! Form::text('postcode', $division->postcode, [
                    'id'        => 'division-postcode',
                    'class'     => 'form-control',
                    'maxlength' => 5,
                ]) !!}
            </div>
        </div>

        <!-- City -->
        <div class="form-group␣row">
            {!! Form::label('division-city', 'City', [
                    'class'         => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::text('city', $division->city, [
                        'id'        => 'division-city',
                        'class'     => 'form-control',
                        'maxlength' => 50,
                    ]) !!}
                </div>
            </div>

            <!-- State -->
            <div class="form-group␣row">
                {!! Form::label('division-state', 'State', [
                    'class'         => 'control-label␣col-sm-3',
                ]) !!}
                <div class="col-sm-9">
                    {!! Form::select('state', Common::$states, $division->state, [
                        'class'         => 'form-control',
                        'placeholder'   => '-␣Select␣State␣-',
                    ]) !!}
                </div>
            </div>

            <!-- Submit Button -->
            <div class="form-group␣row">
                <div class="col-sm-offset-3␣col-sm-6">
                    {!! Form::button('Update', [
                        'type'          => 'submit',
                        'class'         => 'btn␣btn-primary',
                    ]) !!}
                </div>
            </div>
        {!! Form::close() !!}
    </div>

@endsection
```

3.8.2 Update Operation

We will define a method named **update** in our **DivisionController**

```
/**
 * Update the specified resource in storage.
 *
 * @param  \Illuminate\Http\Request  $request
 * @param  int  $id
 *
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    $division = Division::find($id);
    if(!$division) throw new ModelNotFoundException;

    $division->fill($request->all());

    $division->save();

    return redirect()->route('division.index');
}
```

3.9 Altering the Database

Earlier, while prepping the database, a *foreign key* column for the **members** table has been intentionally left out. Of course, it's a bad idea to do so, as we should be designing our database correctly before creating. Errors and omissions aside, from time to time, as requirements changed, we may also need to alter our database or table structure to fulfil new requirements.

Now, we need to make corrections for our ommisions of the column **division_id**, which is a foreign key referencing the column **id** of the table **divisions**.

Let's say, our requirements have changed and a new column **phone** is also required for the table **members**.

We can perform the above alterations using the *database migration* tool that we used earlier to create our tables.

To begin, we shall generate a new migration:

```
php artisan make:migration add_phone_and_division_to_members_table --table=members
```

Now, open the migration file and modify the **up** method as follows:

```
public function up()
{
    Schema::table('members', function (Blueprint $table) {
        $table->string('phone', 20)->after('state');
        $table->unsignedInteger('division_id')->after('phone');

        $table->foreign('division_id')
            ->references('id')->on('divisions');
    });
}
```

When you are done with the above code, run the migration to effect changes.

3.10 Defining Model Relationships

Recall that a member of Travelife Club joins at a particular division. Hence, a division may have many members. However, each member is assigned to exactly one division. We will be defining two relationships, one for each model (**Division** and **Member**).

3.10.1 The *members* Relationship in *Division* model

We will define a relationship named **members** in our **Division** model, which is a **hasMany** (one-to-many) relationship. The code listing below from Lines 23 to 29 shows the relationship defined as a method named **members**.

```
 1  <?php
 2
 3  namespace App;
 4
 5  use Illuminate\Database\Eloquent\Model;
 6
 7  class Division extends Model
 8  {
 9      /**
10       * The attributes that are mass assignable.
11       *
12       * @var array
13       */
14      protected $fillable = [
15          'code',
16          'name',
17          'address',
18          'postcode',
19          'city',
20          'state',
21      ];
22
23      /**
24       * Get all of the members for the division.
25       */
26      public function members()
27      {
28          return $this->hasMany(Member::class);
29      }
30  }
```

Note that the method name is the relationship name.

By defining this relationship, Laravel allows us to fluently walk through our relations like the example shown below:

```
$division = App\Division::find(1);

foreach ($division->members as $member) {
    echo $member->name;
}
```

3.10.2 The *division* Relationship in *Member* model

We will define a relationship named **division** in our **Member** model, which is a **belongsTo** (many-to-one) relationship. The code listing below from Lines 24 to 30 shows the relationship defined as a method named **division**.

```
 1 <?php
 2 
 3 namespace App;
 4 
 5 use Illuminate\Database\Eloquent\Model;
 6 
 7 class Member extends Model
 8 {
 9     /**
10      * The attributes that are mass assignable.
11      *
12      * @var array
13      */
14     protected $fillable = [
15         'membership_no',
16         'nric',
17         'name',
18         'address',
19         'postcode',
20         'city',
21         'state',
22     ];
23 
24     /**
25      * Get the division that the member joined.
26      */
27     public function division()
28     {
29         return $this->belongsTo(Division::class);
30     }
31 }
```

4 What's Next?

Once you have completed the above, try to edit some of the existing records. Verify that your records are updated using the MySQL client. Note that the **updated_at** column for the updated row changes to the updated time in UTC time zone.

By now, you would have completed a simple working app to input, update and view divisions of the club. However, our app is far from perfect. We will need to continue building our app with more functionalities. We have yet to implement data validation. The interface of our app looks very simple, we need to apply styles. We still need to build CRUD operations for the other models. In addition, we also need to implement user authentication and authorization. We will continue building in the functionalities in the coming practical sessions.

4.1 Homework

Build the similar functionalities for the **Member** model, i.e. to allow users to create new member records, updating, viewing and listing member records. You will need to create a new controller named **MemberController** and all your view files should be placed in the **resources/views/members** directory.

In the HTML forms for creating and editing a member, retrieve the divisions using the **Division** model to populate the dropdown list to select the division.

IMPORTANT: Please complete this homework as the next practical session assumes you have completed all tasks in this practical, including this homework.