A Very Complete Introduction to Laravel

Summary (v1.0)

http://www.cocotuts.com/

Contents

1 Introduction

1.1 Welcome

See this lesson's presentation slides.

1.2 Installing A Local Server

To start our local development, We'll first first need a to setup a local server. For Windows, Mac and Linux users, there is **WAMP**, **MAMP** and **LAMP** respectively. We can simply google for these to find the official websites. After the installation, we will see some sort of welcome/succes-message if we navigate to `localhost` in our browser (provided that the installation was succesful).

Windows users need to make sure that they add the paths of `php.exe` and `mysql.exe` to their PATH variable (`\wamp\bin\php\phpx.x.x\`; and `\wamp\bin\mysql\mysqlx.x.x\bin\`;), so PHP and MySQL commands can be executed directly from the command line.

If we can execute `php --version` and `mysql --version` from the command line without errors, we're good to go!

1.3 Installing Laravel

Before installing Laravel, we need to make sure that we meet the requirements on our PHP version (we can check the official Laravel documentation for these). If that's OK, we can grab the latest version of Composer (from `http://getcomposer.org/`) by opening up the command line and executing

```
php -r "readfile('https://getcomposer.org/installer');" | php
```

This will download a file called `composer.phar` to the folder where this command was executed (make sure that the openssl extension is enabled in the php configuration file at `\wamp\bin\php\phpx.x.x\php.ini`).

Next, we'll drop that file's extension by renaming it `composer` and then we'll move it to the `\wamp` folder. Next, We'll delete everything inside the `\wamp\www` folder and then we execute the following command to install the framework:

```
php composer create-project laravel/laravel www --prefer-dist
```

This will install the framework to the `\wamp\www` folder. We can also add a version constraint (like `4.2.*`) at the end of the command to install a specific version.

Next, we need to make sure that the `\app\storage` folder (this is relative the `\wamp\www` folder) has write-permission by the server and that the `mod_rewrite` module is loaded in the Apache configuration file (located at `\wamp\bin\apache\apachex.x.x\conf\httpd.conf`). After a server restart we should be able to see the 'You have arrived'-page in the browser when we navigate to `localhost/public`.

To change our server's default folder for serving files from `\wamp\www` to `\wamp\www\public`, we must change the `DocumentRoot` variable accordingly, which is located inside the Apache configuration file (`httpd.conf`) that we used a second ago. This will allow us to visit our application's base URL by just navigating to `localhost`, instead of `localhost/public`.

1.4 Remarks

We should put our application into debugging mode when developing to allow for detailed error messages (this can be done by setting `'debug' => true` inside `\app\config\app.php` inside our `\www` project folder).

If our application is live, it should **NEVER** be in debugging mode for security reasons.

2 Routing

2.1 Basics

Routes are responsible for making sure that the apprapriote response is given when a certain URL is requested. In Laravel, all routes are registered inside the `\app\routes.php` file of our project folder. If we want to return a simple string to the user when he or she requests the base URL of our application, we write the following inside `routes.php`:

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

2.2 Route Parameters

Many times, it'll be interesting to grab some parameter of the requested URL. For instance, if someone visits the `/users/james` URL at our website, we might be interested in grabbing that `james` part. This is easily done by setting up a route with a route parameter, like this:

```
Route::get('users/{name}', function($name) {
   return $name;
};
```

Route parameters can also be defined as optional by writing `'users/{name?}'`. In such cases however, a default value needs to be specified for the route parameter that we pass to the anonymous function in the second argument of `Route::get()`.

3 Views

3.1 Basics Of Blade

A view is a type of response that can be returned to a user when he/she requests something of our application. Laravel offers us a powerful templating engine called Blade, in order to easily work with these views.

Typically, views are stored within the `\app\views` folder. Remember that all views, in which we want to use Blade, **must** have the `.blade.php` extension. We could, for instance, return a view whenever someone visits the `/welcome` URL by adding the following to our `routes.php` file:

```
Route::get('welcome', function() {
   return View::make('hello');
};
```

Within our Blade views, we may use the {{ `$var` }} syntax to echo the value of some PHP variable. For Sublime Text users, a Blade syntax-highlighter can be downloaded via Package Control.

3.2 Passing Data To Views

Instead of creating variables within our views, Laravel lets us pass variables along when we create our views. We always need to specify how we want our variable to be called inside the view, **and** the value that we want our variable to have inside that view. In Laravel, there are two ways of passing variables along with a view.

- ```
  return View::make('hello')
      ->with('var1', $firstVariable)
      ->with('var2', $secondVariable);
  ```

- ```
  return View::make('hello', array(
      'var1' => $firstVariable,
      'var2' => $secondVariable)
    );
  ```

These variables can then be echoed inside the `hello` view, by using the curly braces syntax: {{ `$var1` }}. We may also use magic methods to write `->withVar($myvar)` instead of `->with('var', $myvar)`.

3.3 Conditionals And Loops

Instead of embedding PHP tags in our Blade views to create if-statements, we can also use the following Blade syntax:

```
@if ($sun === 'shining')
   The weather is nice!
@endif
```

There is also expressive syntax for creating loops:

```
@for ($i = 0; $i < 10; $i++)
   This is iteration number {{ $i }}.  <br>
@endfor
```

The syntax for `foreach` and `while` loops is similar.

3.4 Extending Views

Blade allows us to easily create master/parent pages. To create such a page, we just create a new Blade view inside our `\app\views` folder and fill it with the desired 'skeleton', or basic structure. We can then add 'entry'-points for inheriting children-views by calling on `@yield('content')`.

In a different view, we may inherit from this page by just calling `@extends('filename-of-master-page')` at the top. We can then fill the created entry points as follows:

```
@section('content')
   <p>I'm a paragraph!</p>
@stop
```

In our master view, we may also create an 'entry'-point, with some default code inside. The syntax is as follows:

```
@section('nav')
   default code ...
@show
```

We can override this default code in child-views by just calling

```
@section('nav')
   this will override my parent's default code ...
@stop
```

Alternatively, we can use the parent's default code by calling `@parent`:

```
@section('nav')
   @parent
   this will be appended to my parent's default code ...
@stop
```

3.5 HTMLBuilder

HTMLBuilder is a class that Laravel offers us to easily construct certain HTML elements. For instance, `HTML::link('http://www.google.com/', 'Google')` will return the string `'<a href='http://www.google.com/'>Google</a>'`. Of course, we can also call on these HTMLBuilder methods from within our views, by embedding them between {{ curly braces }}.

For more information, or a complete list of all available methods, visit the official documentation on HTMLBuilder.

4 Forms

4.1 GET And POST Requests

Whenever creating an HTML form, we need to specify the action attribute (the URL to which the form is to be submitted) and the **method** attribute.

Whenever we try to access some URL by just typing it into our browser, we are always using the **GET** method for our request. If a form is submitted to some URL using the GET-method, all form input is contained within the submitted URL (it can be seen in the address bar).

Many times, we'll want to submit our forms in a more secure way. This can be done by setting the method to **POST**. If we want to respond to a POST-request to some URL in our application (for instance `/validate`), we'll need to register a route for it inside our `routes.php` file:

```
Route::post('validate', function()
   return 'Form was submitted!';
};
```

4.2 FormBuilder

The basic idea with FormBuilder is the same as with HTMLBuilder: it's just a class to easily setup our HTML forms. For instance, writing

```
{{ Form::open(array('url' => 'validate')) }}
   {{ Form::submit() }}
{{ Form::close() }}
```

in one of our Blade views will render the following HTML:

```
<form method='POST' action='validate'>
   <input type='submit'>
</form>
```

We must always specify the action URL in the array that's passed to the `Form::open()` method. We may also specify the desired method in that same array (by adding `'method' => 'GET'` for example): if we don't specify the method, it will default to POST.

5 Database

5.1 Connecting To The DB

To get started, we'll first need to setup some database. It's easiest to setup a MySQL database, and while we're at it, let's also create a users table (and fill it). This can either be done using PHPMyAdmin (just go to `localhost/phpmyadmin` in the browser) or by using the command line:

```
mysql -u root
CREATE DATABASE lrv;
USE lrv;
CREATE TABLE users (
   id INTEGER PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(255) NOT NULL,
   occupation VARCHAR(255) NOT NULL
);
INSERT INTO
   users (name, occupation)
VALUES
   ('Walter', 'Chemist'),
   ('Saul', 'Lawyer'),
   ('Gus', 'Businessman');
```

Next, we'll open up the database configuration file at `\app\config\app.php`. Here we can specify our default database type (`'default' => 'mysql'`), and the credentials (host, database, username, password, etc.) for our database connection.

We can test the database connection by calling

```
DB::connection()->getDatabaseName()
```

somewhere in our application. If the connection was succesful, this will return the name of the database.

5.2 Raw SQL

To execute SELECT-queries against the database, we may use Laravel's `DB` class. For instance, `DB::select('SELECT * FROM users')` will return an array of stdClass-objects, representing the users our database's users table. We can get the column values of these objects by accessing them as properties, e.g: `$name = DB::selectOne('SELECT * FROM users WHERE id = 1')->name;`

For INSERT, UPDATE and DELETE queries, we would use the `insert`, `update` and `delete` methods of the `DB` class respectively (in a similar way). If we want to execute a query that doesn't belong in any of these categories, we can use the `statement` method, for example `DB::statement('ALTER TABLE users ADD email VARCHAR(60)')`.

Note 1. To only grab the first element of an array of results, we can use the `selectOne` method instead of the `select` method. For example: `$user = DB::selectOne('SELECT * FROM users WHERE id = 1');`

Note 2. Especially when working with inserts, updates or deletes, we need to be extra careful for SQL injection. QueryBuilder allows us to use question marks inside our queries, so we can attach those bindings as the next argument to the function. Example:

```
DB::update(
   'UPDATE users SET occupation = ?  WHERE id = ?',
   array('Cook', 1)
);
```

Note 3. When updating or deleting from a database table: do ***not*** forget the where-clause. Otherwise, it's going to mass-update or delete every row in the table.

5.3 QueryBuilder

Laravel offers us the QueryBuilder class so we can construct database queries without writing any SQL. When using QueryBuilder, we **always** specify the name of the table first. Next we chain one or multiple methods, which allows us to build our query step by step.

For instance, `DB::table('users')->get()` will return an array of all users in the database. If we just wanted the first element of the results array, we could have used `->first()` instead of `->get()` at the end. To limit the results with a where-clause, we could chain `->where('id', '<', '3')->get()` or `->whereName('Walter')->first()` to `DB::table('users')` instead (we're using magic methods in that last example).

The syntax for inserting, updating and deleting (respectively) is as follows.

- ```
  DB::table('users')->insert(array(
      'name' => 'Mike',
      'occupation' => 'Bodyguard'
  ));
  ```
- ```
  DB::table('users')->where('name', 'Walter')->update(array(
      'occupation' => 'Businessman'
  ));
  ```
- ```
  DB::table('users')->whereId(2)->delete();
  ```

The main idea behind QueryBuilder is that we don't have to embed any SQL. Laravel just checks which kind of database we have setup (MySQL, SQLite, PostgreSQL, SQL Server, etc.) and it handles the actual queries that need to be executed for us. This gives us the freedom to switch between databases anytime we like without breaking our application because of syntax differences between different database types (provided that we've been using QueryBuilder everywhere in our application).

Also with QueryBuilder: consult the official Laravel documentation for more information, and don't forget the where-clause when updating or deleting!

6 Project 1 Fundraising Website

See the source code for this project.
  ```

7 MVC And Rest Theory

7.1 MVC

In the Model-View-Controller (or MVC) design pattern, we divide our application up into three components.

- **Models** represent the data in our application. They are part of the backend and they contain our application's business logic.
- **Views** are for presenting data to our users. They should containt little to no logic (apart maybe from some basic logic structures like conditionals and loops), as they're just used for *presentation*.
- **Controllers** are kind of in the middle between our models and views. When requests to the application are made, they are received and interpreted by the controller. Then, our controller can either create and return a view directly to the user, or it can first communicate with one (or multiple) models before returning a view. In a web application, controllers contain the application's routing logic.

7.2 REST

See this lesson's presentation slides.

8 Controllers

8.1 Basic Controllers

Instead of defining all our routing logic within a single `routes.php` file, Laravel offers us controllers. These are stored within the `\app\controllers` folder and they're defined as classes which must extend the `BaseController` class.

Suppose that we've setup a `HomeController` that contains two public functions: `showWelcome()` and `showAbout()`. Then, inside our `routes.php` file, instead of passing an anonymous function as the second argument to the `Route::get()` (or `Route::post()`) function, we may also pass a controller function like this: `Route::get('/', 'HomeController@showWelcome');` or `Route::get('about', 'HomeController@showAbout');`

Controller functions also allow for (optional) route parameters. Inside our `routes.php` we could grab the following URL:

```
Route::get('users/{name?}', 'HomeController@showProfile');
```

Then, inside our `HomeController`, we could define that `showProfile` function as follows:

```
public function showProfile($name = null) {
   if ($name === null) return 'Showing a list of all users...';
   return 'Showing profile page for '.$name;
}
```

8.2 RESTful Controllers

RESTful controllers are useful if we want to handle the routes to all subURLs of a given URL in a single file. For instance, in our `routes.php` file we can register a RESTful controller by calling `Route::controller('portfolio', 'PortfolioController');` Now, whenever somebody requests a subURL of `/portfolio`, we can handle that within the `PortfolioController` (which also must extend the `BaseController`).

When setting up the controller functions of a RESTful controller, we need to be very careful with the function names. For instance, if we want to setup a route for GET requests to the `/portfolio/paintings` URL, we would have to create a function called `getPaintings()` inside our `PortfolioController`. If we wanted a route for POST requests to the `/portfolio/process` URL, a function called `postProcess()` would be needed.

8.3 Resource Controllers

Many times, we'll be dealing with *resources* in our applications. We might have a user resource, in case of a social network app, or a book resource in case of a library app, and so on. Many times, the ways in which we would like (our users) to interact with these resources is kind of the same, no matter what the resource actually is: many times we want (some of our users) to be able to

- view all resources
- view a specific resource
- create a new resource
- edit/delete an existing resource

Laravel offers us a structured all-in-one solution for dealing with the routes for these actions by giving us **resource controllers**. As an example, let's go with a recipe resource. We may register a resourceful controller inside our `routes.php` file as follows:

```
Route::resource('recipes', 'RecipeController');
```

By declaring the `RecipeController` a resource controller, it is now forced to implement seven methods. We could either implement these ourselves, or we could use the command line to auto-generate everything for us. If we execute

```
php artisan controller:make RecipeController
```

from the command line, it's going to create that `RecipeController.php` file for us inside the `\app\controllers` folder, and define those seven functions inside it. We can check the table below to find out which URL we must visit (using which HTTP verb) to fire which function inside our `RecipeController`.

verb	URL	function
GET	/recipes	index()
GET	/recipes/create	create()
POST	/recipes	store()
GET	/recipes/{id}	show($id)
GET	/recipes/{id}/edit	edit($id)
PUT	/recipes/{id}	update($id)
DELETE	/recipes/{id}	destroy($id)

9 Advanced Database

9.1 Migrations

Laravel gives us migrations in order to version control the state of our database. Migration files are stored within the \app\database\migrations folder. Instead of creating such a migration file ourselves, it's easier to auto generate it by executing

```
php artisan migrate:make create_recipes_table
```

from the command line. If we take a look at the generated file inside our migrations folder, we will notice a `public function up()` and a `public function down()`.

In the `up()` function, we write the code for setting up the recipes table:

```
DB::statement(
   'CREATE TABLE recipes (
      id INTEGER PRIMARY KEY AUTO_INCREMENT,
      name VARCHAR(255) UNIQUE NOT NULL,
      body TEXT NOT NULL'
);
```

In the `down()` function, we would write the code for **undoing** whatever we wrote in the `up()` function: `DB::statement('DROP TABLE recipes');`

Now, we can easily migrate the database by running `php artisan migrate` from the command line. This will fire the `up()` method in our migration. If we wish to undo the creation of that recipes table, we simply call `php artisan migrate:rollback`, after dumping Composer's autoload (this can be done by calling `php ../composer dump-autoload` from the root folder of our application, assuming that our `composer` file is located one directory higher).

9.2 SchemaBuilder

The main idea with SchemaBuilder is the same as with QueryBuilder: by using it, we'll make our application independent of the underlying database, because Laravel will handle the actual queries that need to be executed against the database that we've setup. The only difference is that SchemaBuilder is for creating and modifying our database tables, whereas QueryBuilder was made for querying existing tables in our database.

For example, to create a books table using SchemaBuilder, we would create a `create_books_table` migration file from the command line, and fill the bodies of the `up()` and `down()` functions with

```
Schema::create('books', function($table) {
   $table->increments('id');   // integer id-field primary key AI
   $table->string('title');    // varchar title-field not null
   $table->string('author');   // varchar author-field not null
   $table->text('summary')->nullable();   // text summary-field
});
```

and

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

respectively. Later in time (after migrating the database with `php artisan migrate`), we could easily make some adjustments to this books table (without rolling back the previous migration) by creating a new migration called something like `adjust_books_table`, and filling the `up()` and `down()` functions on that migration with

```
Schema::table('books', function($table) {
   $table->string('genre');   // adding a genre column
   $table->timestamps();      // adding some timestamps
   $table->unique('title');   // declaring the title-field unique
});
```

and

```
Schema::table('books', function($table) {
   $table->dropColumn('genre');
   $table->dropColumn(array('updated_at', 'created_at'));
   $table->dropUnique('books_title_unique');
};
```

respectively. Executing `php artisan migrate` will only run the latest migration then. For a list of all available column types and some general information, take a look at the official Laravel documentation on SchemaBuilder.

9.3 Seeding

If we want to fill (or *seed*) our database, we may write the necessary code for doing so inside a php file in our `\app\database\seeds` folder. By default, we will find a `DatabaseSeeder.php` file there, containing a `DatabaseSeeder` class.

Suppose we have a `books` table with a `title` and an `author` column. In the body of the `run()` function of the `DatabaseSeeder` class, we may call another seeder class from somewhere in our seeds folder like this:

```
$this->call('BooksTableSeeder');
```

At the bottom of that same file, we could define that `BooksTableSeeder` as follows:

```
class BooksTableSeeder extends Seeder {
   public function run() {
      for ($i = 0; $i < 100; $i++) {
         DB::table('books')->insert(array(
            'title' => 'This is book '.$i,
            'author' => 'Author '.$i
         ));
      }
   }
}
```

As can be seen from this example, all seeder classes extend `Seeder` and contain a `public function run()`. The books table is now easily filled with a hundred books by executing `php artisan db:seed` from the command line.

9.4 Models

Models contain our application's business logic. They are stored inside the `\app\models` folder. For example, a blogging application could have an Article model, which would abstractly represent the articles on the website by containing various article related methods and properties. An example Article model could look something like this:

```
class Article {
   public $title;
   public $body;

   public function __construct($title, $body) {
      $this->title = $title;
      $this->body = $body;
   }
   public function getArticleLength() {
      return str_word_count($this->body);
   }
}
```

9.5 Eloquent Part 1

Eloquent is the name of Laravel's ORM, which stands for Object Relational Mapping and which basically creates a connection between our model and our database table. To create such a link on one of our models, the Article model for instance, we could define the Article class as follows:

```
class Article extends Eloquent {

   public function getArticleLength() {
      return str_word_count($this->body);
   }

}
```

There are some differences between this lesson and the previous lesson in the way that we're defining this Article class. First of all, we've dropped those Article properties and that Article constructor. Secondly, our model now extends `Eloquent`, which has a few implications:

- Laravel will assume (by convention) that we've got a table called `articles` in our database, as this is the plural form of our class name. To override this convention, we can simply specify a `protected $table` property on our Article class, whose string value equals the desired table name.
- Laravel will assume that the database table which is linked to this model contains timestamps (i.e.: `created_at` and `updated_at` columns). To disable this default behaviour, create a `public $timestamps` property on the Article class and set its vale equal to `false`.

If we've got an `articles` table, with a title field, a body field and some timestamps, we can now use Eloquent to insert new users into that table by one of the following two ways:

```
$article = Article::create(array(
   'title' => 'How To Peel Potatoes',
   'body' => 'This is some body ...'
));
```

or

```
$article = new Article();
$article->title = 'How to be succesful!';
$article->body = 'This is some other body ...';
$article->save();
```

The first way will likely give us a mass-assignment error: by default, Laravel disallows us to simultaneously update multiple columns in the database when using Eloquent syntax. We can specify the columns that we *want* to be mass assignable inside a `protected $fillable` array on our Article model. For example: `protected $fillable = array('title', 'body');`

Eloquent allows us to access column values as the properties of our Article objects. For instance, we could get the id of a freshly inserted article by calling `$article->id` in the example above.

9.6 Eloquent Part 2

Of course, we can also retrieve, update and delete articles using expressive Eloquent syntax. We use `Article::all()` to get an array of all articles in the database. To limit the result with a where-clause, we can simply use the syntax that we've learned from QuerBuilder, e.g.: `Article::where('id', '<' 3)->get()` or `Article::whereTitle('How to be succesful!')->first()`. We can use `Article::find($id)` to grab an article by its id.

To update a single article or mass update multiple articles, we could use

```
$article = Article::find(2);
$article->title = 'How to drive a car';
$article->save();
```

or

```
Article::where('id', '<', 4)->update(array(
   'body' => 'test'
));
```

respectively. To delete a single article by its id or mass delete multiple articles at once, we could use `$article->destroy($id)` or `Article::where('id', '>', 2)->delete()` respectively. There is also a lot of documentation on Eloquent on the official website.

9.7 Relationships

Most of the time, two or more tables in our database are going to be related to each other in some way. We could think of an articles table and a comments table for instance. Each comment in the comments table must belong to some article in the articles table, and each article could have one or multiple comments. Such a database relation is called a **one-to-many** relation from articles to comments.

Suppose we have an articles table, and a comments table containing an `article_id` foreign key column which references the `id` column on the articles table. After setting up an article and a comment model (both of which must extend `Eloquent` of course), we can define a one-to-many relation as follows:

```
// inside Article.php
public function comments() {
   return $this->hasMany('Comment');
}

// inside Comment.php
public function article() {
   return $this->belongsTo('Article');
}
```

Now, we can grab ourselves an array of all comments on a particular article by calling `Article::find($article_id)->comments()->get()`, or `Article::find($article_id)->comments` using magic properties. Similarly, we can get the article to which a particular comment belongs by calling `$article = Comment::find($comment_id)->article`. We could then get that article's title of course by calling `$article->title`.

10 Miscellaneous

10.1 Filters

Filters are used to restrict access to some parts of our website, based on some condition. Filters are registered within `\app\filters.php` and an example would be a filter for restricting access to the website on sundays. We could register this sunday-filter by writing the following inside `filters.php`:

```
Route::filter('sunday', function() {
   if (date('l') === 'Sunday') // l = lowercase L
      return 'Please come back tomorrow.';
});
```

To apply this filter to one of our routes, we open up `routes.php`. To prevent users from visiting our homepage, we could apply the sunday-filter to our base URL route like this:

```
Route::get('/', array('before' => 'sunday', function() {
   return View::make('hello');
});
```

Filters may also be defined on controllers by using that controllers constructor. For instance, if our `HomeController` has functions `showWelcome()`, `showAbout()` and `showContact()`, we can add our filter to all those functions by creating the following `HomeController` constructor:

```
public function __construct() {
   $this->beforeFilter('sunday');
}
```

If we only want to run this filter on some of the functions in the `HomeController`, we might do something like this:

```
public function __construct() {
   $this->beforeFilter(
      'sunday',
      array('only' => array('showWelcome', 'showAbout'))
   );
}
```

Instead of `'only'` filtering some `HomeController` functions, we can also filter them all, `'except'` for a few ones specified.

10.2 Validation

Laravel offers us a very easy way to validate form input and other data. Suppose we have an array of data which looks like the following (and many times, such an array would come from a submitted form in the form of `Input::all()`):

```
$data = array(
   'name' => 'Billy',
   'link' => 'http://www.cocotuts.com/',
   'password' => 'milkshake',
   'password-repeat' => 'mjlkshake'
);
```

Before the actual validation, we need to specify an array of validation rules, which could look something like this:

```
$rules = array(
   'name' => 'required',
   'link' => 'required|url',
   'password' => 'required|min:6',
   'password-repeat' => 'required|same:password'
);
```

As can be seen from this example, multiple validation rules are seperated by a pipe-character. Now, we're ready to create a validator object: `$validator = Validator::make($data, $rules);`

If the data came from a form and we wanted to return to that form page if the validation failed, we could do something like this:

```
if ($validator->fails()) {
   return Redirect::to('form')
      ->withErrors($validator->messages())
      ->withInput();
}
```

Inside the view for displaying the form, we will now have an `$errors` variable. We can get an array of all validation errors related to some input field by calling `$errors->get('password')` within that view, or we can get an array of all errors for all fields by calling `$errors->all()`. Also, because we chained that `->withInput()` method at the end, all form fields will be re-populated with the data that the user entered before submitting the form.

In this example, the validation of the `$data` would have failed because the `password` and `password-repeat` fields didn't match.

10.3 The Artisan File And 404s

The artisan file is located at the root of our application and is used to interact with our application via the command line. Some artisan commands that we've already come across include `controller:make`, `migrate`, `migrate:make` and `db:seed`. Two other interesting commands are `php artisan down` for putting our application into maintenance mode, and `php artisan serve` for serving a Laravel application at `localhost:8000`, even though that application's root isn't located in the `\wamp\www` folder. To get a list of all available artisan commands, simply run `php artisan`.

To change the default 'Be right back!'-message when our application is in maintenance mode, or to handle 404 errors, which occur when we try to access an unregistered URL, we open up `\app\start\global.php`. Here we can modify the `App::down()` function as follows to return some view instead of just a message:

```
App::down(function() {
   return Response::view('maintenance', array(), 503);
});
```

For returning a custom 404 page, we need to add

```
App::missing(function($exception) {
   return Response::view('404page', array(), 404);
});
```

Note how we're using `Response::view()` instead of `View::make()` here: this is because if we use the `Response` class, we can also send the right HTTP status code to the browser (for example the infamous 404: the page/resource could not be found, or 503: service unavailable). If we use `View::make()`, it'll always send 200 (Request Succesful) to the browser. For more information, take a look at `http://en.wikipedia.org/wiki/List_of_HTTP_status_codes`.

10.4 Way's Generators

A Laravel expert called Jeffrey Way has developed a set of generator functions for easily generating views, migrations, models and many more things from the command line. To include his package in our project, we need to open up `composer.json` at the root of our application. In here, we need to require his package by adding the following lines:

```
"require-dev":  {
   "way/generators":  "2.*"
}
```

Next, we run `php ../composer update --dev` in the command line at the root of our application (assuming our `composer` file is located one directory higher) to actually pull in his package. Once that's finished, we just need to register the service provider. Open up `\app\config\app.php`, and scroll down until you find the `providers` array. At the bottom of that array we add

```
'Way\Generators\GeneratorsServiceProvider'
```

Now, if we execute `php artisan`, we should see a handful of new generator commands. Using these, it's very easy for example to setup a database with just 2 commands, without ever touching the text editor:

```
1| php artisan generate:migration create_users_table
      --fields="name:string, age:integer:nullable"
2| php artisan migrate
```

For a detailed explanation of how these commands work, take a look at the way/generators GitHub page.

10.5 Seeding With Faker

Faker is a very popular package by fzaninotto to generate realistic first names, last names, usernames, e-mails, adresses, and many more things. To require it, we add it to our `composer.json` file (below the line where we require the Laravel framework itself):

```
"require": {
   "laravel/framework": "4.2.*",
   "fzaninotto/faker": "1.*"
}
```

After that, we can pull it in by running `php ../composer update` or `php ../composer update fzaninotto/faker` from the command line. Once that's finished, we're basically ready to use it for seeding our database for example. Suppose we're connected to a database with a `users` table which has a column for `first_name`, `last_name` and `address`. We could then open up `\app\database\seeds\DatabaseSeeder.php` and define a UsersTableSeeder at the bottom of that file:

```
class UsersTableSeeder extends Seeder {
   public function run() {
      $faker = Faker\Factory::create();
      for ($i = 0; $i < 100; $i++) {
         User::create(array(
            'first_name' => $faker->firstName,
            'last_name' => $faker->lastName,
            'address' => $faker->address
         ));
      }
   }
}
```

Note how we're using Eloquent syntax here. That's because, by default, Laravel gives us a User model (in the `\app\models` folder) which already extends

Eloquent. After calling `$this->call('UsersTableSeeder');` in the body of the `run()` function of the `DatabaseSeeder` class, we're ready to seed by calling `php artisan db:seed` from the command line.

For a detailed explanation of this package and its many features, take a look at the fzaninotto/faker GitHub page.

11 Project 2 Registration And Login System

See the source code for this project.