

Jamal Armel

Web application development with Laravel PHP Framework version 4

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

11 April 2014

Author(s) Title Number of Pages Date	Jamal Armel Web application development with Laravel PHP Framework version 4 53 pages 11 April 2014
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	.NET application development and Hybrid Media
Instructor(s)	Aarne Klemetti, Senior Lecturer
<p>The purpose of this thesis work was to learn a new PHP framework and use it efficiently to build an eCommerce web application for a small start-up freelancing company that will let potential customers check products by category and pass orders securely. To fulfil this set of requirements, a system consisting of a web application with a backend was designed and implemented using built in Laravel features such as Composer, Eloquent, Blade and Artisan and a WAMP stack.</p> <p>The web application was built using the Laravel framework version 4, a modern PHP framework that aims at making PHP development easier, faster and more intuitive. The web application was built following the MVC architecture pattern. Admin panels were created for easily updating and managing the categories and products and uploading product images as well. A public interface was made available also to let registered users to log in and add orders to their carts and proceed to check out using <i>PayPal</i>.</p> <p>The application is easily expandable and features can be added or removed effortlessly thanks to the Laravel's ability to manage packages through Composer's Packagist online repository.</p> <p>The results proved that Laravel 4 is effectively a premium choice for a PHP framework that helps developers rapidly build secure, upgradable web applications.</p>	
Keywords	PHP, Laravel 4, MVC, Database, eCommerce

Contents

List of Abbreviations

1	Introduction	1
2	Laravel's main features	2
2.1	Architecture	2
2.2	MVC	4
2.2.1	Model	4
2.2.2	Views	4
2.2.3	Control	5
2.2.4	Database	5
2.3	Composer	7
2.4	Artisan	10
3	Creating the workflow and configuring our environment	11
3.1	Operating system	11
3.2	Terminal	11
3.3	Text editor	11
3.4	Bootstrap as the HTML5/CSS3/Javascript framework	12
3.5	Apache–MySQL–PHP package	13
3.6	Installing Composer	14
3.7	Installing Laravel 4	15
3.8	Database	16
4	Building the application with Laravel 4	20
4.1	Designing our application	20
4.1.1	The Idea	20
4.1.2	Entities, relationships and attributes	20

4.1.3	Map of the application	21
4.2	Creating a new application	23
4.2.1	Creating a main view	23
4.2.2	Creating the Eloquent models and their respective schemas	25
4.2.3	Image managing as an example of dependency management	30
4.2.4	Creating the Controllers and their respective Routes	32
4.2.5	Creating the views	38
4.3	Authentication and security	44
4.3.1	Authenticating users	44
4.3.2	Securing the application	48
5	Conclusion	50
	References	52

List of abbreviations

MVC	Model, View and Control
WAMP	Windows, Apache, MySQL, and PHP
PHP	Personal Home Page
DBMS	Database Management System
SQL	Structured Query Language
MySQL	My Structured Query Language
ORM	Object Relational Mapper
Apache	Apache HTTP Server
HTTP	HyperText Transfer Protocol
CRUD	Create, Read, Update and Delete
CSRF	Cross-Site Request Forgery
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
XML	Extensible Markup Language
API	Application Programming Interface
UI	User Interface

1 Introduction

The purpose of this thesis work is to learn a new PHP framework and use it efficiently to build an eCommerce web application for a small start-up freelancing company that will let potential customers check products by category and pass orders securely. To fulfil this set of requirements, a system consisting of a web application with a backend will be designed and implemented using a modern MVC framework.

It is worthwhile considering the use of a PHP framework when time is a limitation and the developer's PHP coding skills do not match the high level demanded to build a complex application. Frameworks handle all the repetitive basic tasks of a PHP project, letting the developer concentrate her/his efforts on the business logic and the general structure of the project as a whole, in doing so, frameworks are becoming an ideal tool used by said developers to rapidly build complex operational prototypes in a matter of hours with minimal time spent on coding. Frameworks offer also whole range of ready-made utilities and libraries.

The use of a robust framework is recommended when the security of the web application is an essential requirement. It even becomes a necessity when the developer lacks the necessary know-how to prevent security breaches from happening. Most of the modern frameworks have built-in security features that range from input sanitising to automatic cookie encryption.

Organised structure of the project as a whole, clear and clean code are required when working in an organisation or co-developing an application in a team of developers. Frameworks permit the organisation of said code into a logical architecture, thus facilitating its maintainability and expandability. To achieve this, modern PHP frameworks follow the Model-View-Controller (MVC) architecture pattern.

Among the highly popular PHP frameworks, Laravel stands out with its claim in its ability to produce a development process that is agreeable for the developer without losing the application's functionality. That is one of the many reasons it was chosen as the framework of choice for building an eCommerce web application for Armel Solutions freelance start-up. This thesis work will study if Laravel lives up to its claim by evaluating its ability in building an up and running secure eCommerce web application in minimal time.

2 Laravel's main features

This study will focus only on the features used during the building of the eCommerce web application, otherwise this work will not be large enough to cover the entirety of the features of the whole Laravel 4 framework.

2.1 Architecture

Laravel is a web application framework that tries to ease the development process by simplifying repetitive tasks used in most of today's web applications, including but not limited to routing, authentication, caching and sessions. [1]

Since it manages to do all essential tasks ranging from web serving and database management right to HTML generation, Laravel is called a full stack framework. This vertically integrated web development environment is meant to offer an improved and smooth workflow for the developer. [2]

Unlike other vertically integrated environments, Laravel is unique in its way of prioritizing convention over configuration. In fact, while many PHP frameworks demand a heavy-duty XML configuration before starting the actual project, Laravel needs only a few lines of PHP code to be edited and it becomes ready to use. Avoiding or using a minimum amount of configuration files gives all Laravel web applications a similar code structure which is very characteristic and identifiable. This might be considered at first glance as serious constraint on how a developer might wish to organize the structure of her/his own web application. However, these constraints make it actually a lot easier to build web applications. [2]

All the new Laravel projects come out of the box equipped with a full directory tree and also many placeholder files resulting in a structure permitting a quick start of the actual development process. This structure is nevertheless fully customizable. Here in the following figure is shown what such a structure looks like: [3, 16.]

ArmelSolutions	
app	Laravel application
commands	Command line scripts
config	Configuration files
packages	
testing	
controllers	Controllers
database	Database
migrations	Migrations
seeds	Seeders
lang	
en	Localisation variables
models	Classes used to represent entitites
start	Startup scripts
storage	
cache	Cache and logs directory
logs	
meta	
sessions	
views	
tests	Test cases
views	
emails	Templates that are rendered to HTML
auth	
bootstrap	Application bootstrapping scripts
public	Document root
packages	
vendor	Third-party dependencies installed through composer
.gitattributes	
.gitignore	
artisan	Artisan command line utility
composer.json	Project dependencies
composer.lock	
CONTRIBUTING.md	
phpunit	Test configuration file for PHPUnit
readme.md	
server	Local development server

Figure 1. A New Laravel 4 project directory structure

2.2 MVC

The term MVC was briefly mentioned earlier in this work and it is worthwhile mentioning now that Laravel is actually a fully-fledged MVC framework. MVC rapidly became the industry's standard practice used in every modern development environment. Many frameworks such as Ruby on Rails, ASP.NET, CakePHP and CodeIgniter make use of it to separate the logic behind the application from the representation layer. [4, 8.]

An MVC architecture pattern let the web application have many different views of a single common model. That is, in our current context of building an eCommerce web application, a Category page, for example, can have multiple views such as the Product List View or Product Gallery View. In an MVC development environment, one model for the Category table will be created and via that one model multiple views can be created. [4, 8.]

The MVC architecture pattern let the developer write a code that can be divided on the basis of the following three things:

2.2.1 Model

A *Model* is the mode by which the developer can manipulate the data. It consists of a layer residing between the data and the application. The data itself can be stored in various types of database systems such as MySQL or even simple XML or Excel files. [4, 8.]

2.2.2 Views

Views are the visual representation of our web application (presentation layer), they are responsible for presenting the data that the *Controller* received from the *Model* (business logic). They can be easily built using the *Blade* template language that comes with Laravel or simply using plain PHP code. *Blade* is driven by template inheritance and sections. When Laravel renders these *Views* it examines first their file extension, and depending on it being either “.blade.php” or simply “.php”, determines if Laravel treats our *View* as a *Blade* template or not. [3, 14.]

2.2.3 Control

The primary function of a *Controller* is to handle requests and pass data from the *Model* to the *Views*. Thus a *Controller* can be considered as the link between our *Model* and *Views*. [4, 8.]

The developer has the option to write her/his business logic either in *Routers* or *Controllers*. *Routers* can be useful when dealing with a small web application, or in rapidly defining static pages. Writing *Controllers* for every single page of the web application is thus not necessary. [4, 12.]

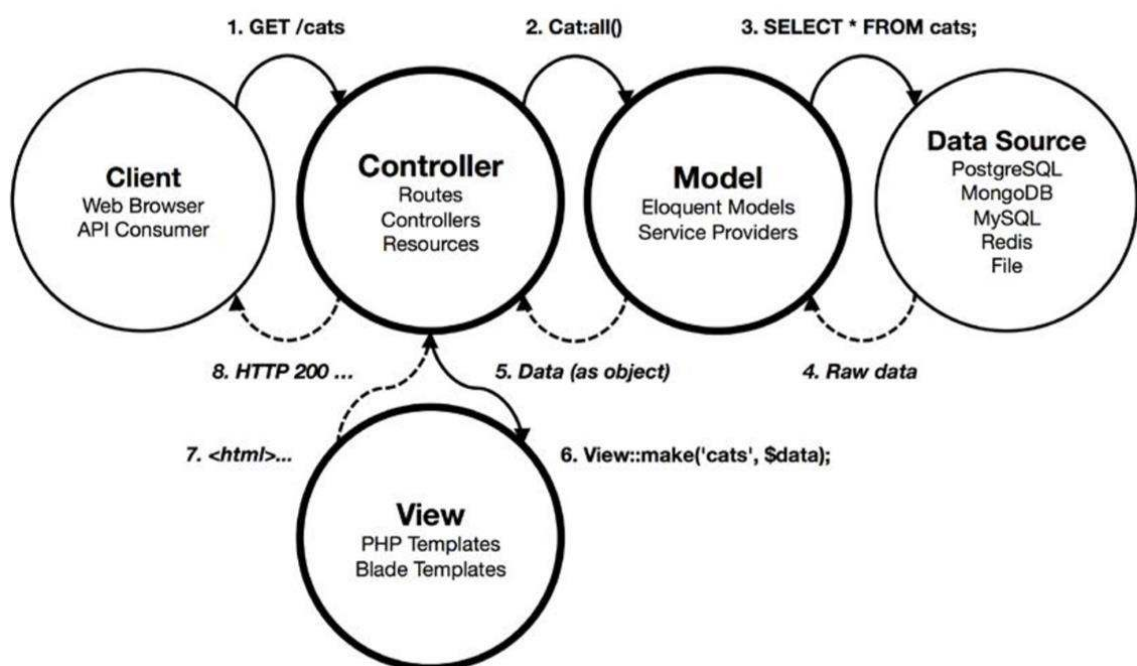


Figure 2. Interactions between all the constituent parts of an MVC architecture pattern. [3, 15]

2.2.4 Database

- Eloquent ORM

The Eloquent ORM provided with Laravel includes a simple PHP ActiveRecord implementation which lets the developer issue database queries with a PHP syntax where instead of writing SQL code, methods are simply chained. Every table in the database possess a corresponding Model through which the developer interact with said table. [5]

- Schema builder

The Laravel Schema class provides a database agnostic (i.e. can function with a multitude of DBMS) way of managing all database related work such as creating or deleting tables and adding fields to an existing table. It works with a multitude of databases systems supported by Laravel and MySQL being the default one. The Schema class has the same API across all of these database systems. [6]

- Managing the database with Migrations

Migrations can be considered as a form of version control for our database. They allow us to change the database schema and describe and record all those specific changes in a migration file. Each Migration is usually associated with a Schema Builder to effortlessly manage our application's database. A migration can also be reverted or “rolled back” using the same said file. [7]

Using our terminal we can issue the following commands to create or drop tables in our database:

Command	Description
\$ php artisan migrate:install	Creates the migration repository
\$ php artisan migrate:make	Creates a new migration file
\$ php artisan migrate:refresh	Resets and reruns all the migrations
\$ php artisan migrate:reset	Rollback all the database migrations
\$ php artisan migrate:rollback	Rollback the last database migration

Table 1. A collection of commands related to migrations. [4, 29]

- Seeders

The *Seeder* class lets us seed data into our tables. This feature is very valuable since the developer may insert data into her/his database's tables every time she/he wants to test the web application. [4, 59.]

When the backend is empty we can populate it with some data by simply issuing the following command in the terminal:

```
$ php artisan db:seed
```

2.3 Composer


Another feature that makes Laravel stand out from the other frameworks is that it is *Composer* ready. In fact Laravel is itself a mixture of different *Composer* components, this adds a much needed interoperability to the framework.

Composer is a dependency management tool for PHP. Essentially, *Composer's* main role in the Laravel framework is that it manages the dependency of our project's dependencies. For example, if one of the libraries we are using in our project is dependent on three other libraries and that there is a need to upgrade all those libraries, then there is no necessity to manually find and update any files. It is possible to update all four libraries via a single command through the command-line, which is, "\$ composer update". [8]

Composer has the ability to manage a dependency up to a given *nth* level, meaning that all dependencies of our project can be managed via a single tool which is a really handy option to have when we are dealing with a multitude of libraries. Another advantage of using *Composer* is that it generates and handles an autoload file at the root of our vendor/ directory, which will contain all the project's dependencies that wires up the autoloading of classes when it is included in a PHP script. In doing so, there is no need from the developer side to remember all dependencies' paths and include each of them on every file of the project, she/he just needs to include the autoload file provided by *Composer*. [4, 11.]

Composer is installed in the form of a PHP executable that is added to the PATH environment variable. A PATH environment variable is the listing of locations that is explored when a command is run in the terminal. When *Composer* is installed properly, the developer can execute it through the command-line from any place in the file system using the “\$ composer” command. The project, and its dependencies, are defined within a JSON file, named *composer.json*. [3, 22.]

Composer is the way the PHP community is heading to, thus there are thousands of thoroughly tested packages already available in the *Composer* package archive. Laravel was designed in such a way that it integrates *Composer* packages easily. All what *Composer* needs to do is to read the contents of the *composer.json* file and connect to *Packagist*, which is an online repository of packages, to resolve all the dependencies, recursively. These dependencies are then downloaded to our local directory called *vendor/*, and then their state is recorded to a file named *composer.lock*. [3, 25.]



Packagist

The PHP package archivist.

[Submit Package](#)

Packagist is the main **Composer** repository. It aggregates all sorts of PHP packages that are installable with Composer.
[Browse packages](#) or [submit your own](#).

Search packages...

Getting Started

Define Your Dependencies

Put a file named `composer.json` at the root of your project, containing your project dependencies:

```
{
  "require": {
    "vendor/package": "1.3.2",
    "vendor/package2": "1.*",
    "vendor/package3": ">=2.0.3"
  }
}
```

Install Composer In Your Project

Run this in your command line:

```
curl -s http://getcomposer.org/installer | php
```

Or [download composer.phar](#) into your project root.

Install Dependencies

Execute this in your project root:

```
php composer.phar install
```

Autoload Dependencies

If all your packages follow the **PSR-0** standard, you can autoload all the dependencies by adding this to your code:

```
require 'vendor/autoload.php';
```

Publishing Packages

Define Your Package

Put a file named `composer.json` at the root of your package, containing this information:

```
{
  "name": "your-vendor-name/package-name",
  "description": "A short description of what your package does",
  "require": {
    "php": ">=5.3.0",
    "another-vendor/package": "1.*"
  }
}
```

This is the strictly minimal information you have to give.

For more details about package naming and the fields you can use to document your package better, see the [about](#) page.

Commit The File

You surely don't need help with that.

Publish It

[Login](#) or [register](#) on this site, then hit the big fat green button above that says [submit](#).

Once you entered your public repository URL in there, your package will be automatically crawled periodically. You just have to make sure you keep the `composer.json` file up to date.

Figure 3. A view from Packagist, the online repository for *Composer* [7]

Laravel combined with the power of *Composer* gives the developer more freedom in choosing what kind of packages she/he would like to use with her/his web application. For example, if she/he do not like the default Mail component that comes with Laravel, which is *Swift Mailer*, and she/he wants to replace it with a more preferred package like the *PHPMailer* component for example, which is likewise *Composer* ready; thus, switching between the two packages would be a very easy task. The developer can replace components at will and with ease when there is need to do so via the *Composer* and *Laravel* configuration. [4, 12.]

2.4 Artisan

A developer would have to usually interact with the Laravel framework using a command-line utility that creates and handles the Laravel project environment. Laravel has a built-in command-line tool called *Artisan*. This tool allows us to perform the majority of those repetitive and tedious programming tasks that most of developers shun to perform manually. [4, 13.]

Artisan can be utilized to create a skeleton code, the database schema and build their migrations which can be very handy to manage our database system or repair it. We may as well create database seeds that will allow us to seed some data initially. It can also be employed to generate the basic Model, View and Controller files right away via the command-line and manage those assets and their respective configurations. [4, 13.]

Artisan let us even create our very own commands and do convenient things with them such as sending pending mails to recipients or anything that can be necessary to properly run our web application. Unit tests for our web application can also be run through *Artisan*. [4, 13.]

3 Creating the workflow and configuring our environment

3.1 Operating system

Laravel is a cross platform framework which is built with interoperability in mind. It can be used on top of a variety of operating systems, including but not limited to Linux, Mac OSX and Windows. The operating system of choice for this project is Microsoft's Windows version 8.1, which is the latest offering from the software giant.

3.2 Terminal

As it was discussed earlier in this work, a developer usually interacts with Laravel framework through a command-line. The Windows operating system comes equipped with two of such command-lines, that is, the *Command prompt* and the *Powershell*. However, for our project we are going to use a popular third party terminal named *Cygwin*. So why make such a choice? The reason is that one of the major inconveniences that a developer can face in a modern development environment is supporting her/his application across heterogeneous platforms. [9]

Cygwin offers a standard UNIX/Linux shell environment, together with many of its greatly handy commands to the Windows platform. By utilizing *Cygwin*, a developer may handle various environments in a reliable and effective way. [9]

To install *Cygwin* we need to download the executable file from the author's website [9] and double click on the downloaded file and just follow the instructions, the installation is done automatically. [9]

3.3 Text editor

For this project the text editor of choice to build our web application will be *Sublime Text* 3. It is a web developer's editor that can do few useful tasks from the editor window itself. Therefore, the developer does not have to constantly switch between windows and run tasks from other applications. Another important aspect of the *Sublime Text* editor is its

Package Control, this package manager allows us to add the package's features. [4, 226]

3.4 Bootstrap as the HTML5/CSS3/Javascript framework

Bootstrap is arguably the industry's most popular frontend web development framework. It offers a full range of user-friendly, cross-platform and tested pieces of code for frequently used standard UI conventions. Bootstrap significantly speeds up the undertaking of building a frontend web interface because of its ready-made, thoroughly tested blend of HTML markup, CSS styles, and JavaScript behaviour. With these essential foundations rapidly set up, we can confidently modify the UI on top of a solid basis. [10, 7.]

There are many ways to download Bootstrap, but not all these ways of downloading Bootstrap are equal. For this project though, and for the sake of rapidity, we will use *Initializr* which generates templates based on HTML5 Boilerplate by permitting the developer to select which components she/he wants from it. [11]

Then after clicking on the download button we will get the following directories and files, logically grouping common assets and providing both compiled and minified variations. The directory and file tree will look like this: [12]

```
bootstrap/
├── css/
│   ├── bootstrap.css
│   ├── bootstrap.min.css
│   ├── bootstrap-theme.css
│   └── bootstrap-theme.min.css
├── js/
│   ├── bootstrap.js
│   └── bootstrap.min.js
└── fonts/
    ├── glyphs-halflings-regular.eot
```

- |— glyphs-halflings-regular.svg
- |— glyphs-halflings-regular.ttf
- |— glyphs-halflings-regular.woff

Please note that this thesis work is mainly focusing on the Laravel framework, therefore the use of the Bootstrap framework or the HTML markup, CSS styling and the JavaScript actions of our web application will not be discussed further than this present chapter.

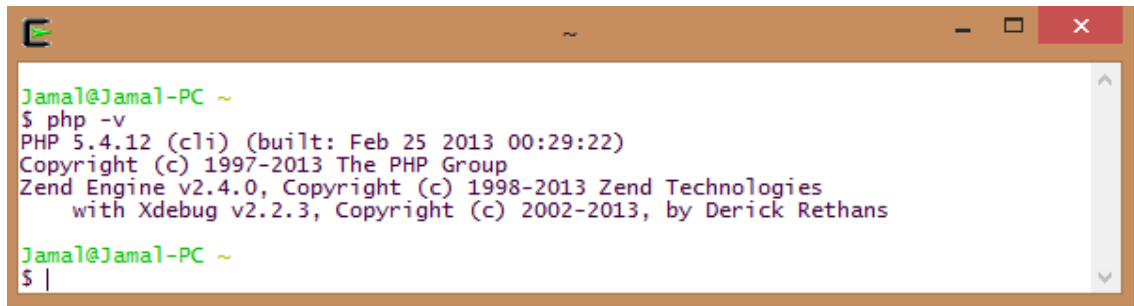
3.5 Apache–MySQL–PHP package

A database is a collection of data with a predefined structure. The set of data it represents can range from as little as a shopping list to a large volume of information in a corporation's network. A database management system is needed to manipulate the data stored in a computer database. [13]

One of the most popular database management systems is MySQL server, it uses the Structured Query Language commonly referred to as SQL. SQL is arguably the most commonly used standardized language for manipulating databases. SQL statements can be used in various ways, they might be entered directly or embedded into a code written in a different language, or use an API that hides the SQL syntax altogether. In our project we will use the second option of embedding the SQL statements into a different programming environment. [13]

The Laravel framework has a very few system requirements, however it explicitly needs PHP version 5.3.7 or above and MCrypt PHP Extension, the latter comes bundled with newer versions of PHP. [15]

Our WAMP stack of choice for this project is WampServer. To install it we need to download the executable file from the author's website [14] and double click on the downloaded file and just follow the instructions, the installation is done automatically. The WampServer package is delivered with the latest releases of Apache, MySQL and PHP, that is, Apache: 2.4.4, MySQL: 5.6.12, PHP: 5.4.12 and PHPMysqlAdmin: 4.0.4. [14]

A screenshot of a terminal window with a brown title bar. The terminal shows the command 'php -v' being executed. The output displays the PHP version as 5.4.12 (cli), built on February 25, 2013, at 00:29:22. It also lists the copyright for The PHP Group (1997-2013) and Zend Technologies (1998-2013), along with the Zend Engine version (v2.4.0) and Xdebug version (v2.2.3, Copyright 2002-2013 by Derick Rethans). The prompt 'Jamal@Jamal-PC ~' is visible at the top and bottom of the terminal output.

```
Jamal@Jamal-PC ~  
$ php -v  
PHP 5.4.12 (cli) (built: Feb 25 2013 00:29:22)  
Copyright (c) 1997-2013 The PHP Group  
Zend Engine v2.4.0, Copyright (c) 1998-2013 Zend Technologies  
    with Xdebug v2.2.3, Copyright (c) 2002-2013, by Derick Rethans  
Jamal@Jamal-PC ~  
$ |
```

Figure 4. The version of the installed PHP

3.6 Installing Composer

As it was mentioned earlier in this work, Laravel framework utilizes *Composer* to manage its dependencies. To install *Composer* on our Windows machine we need to be sure beforehand that we have an appropriate version of PHP installed, then we can get the *Composer* Windows installer from the author's website [8] and download the *Composer-Setup.exe* file. During the process of installation, the installer will ask for the location of the PHP executable in our system, and since we are using WAMP the location is `C:/wamp/bin/php/php5.4.12/php.exe`. The installation will continue automatically then by finalizing the install of *Composer* and adding the `php` and *Composer* commands to our PATH. [3, 24.]

To make sure that *Composer* is installed properly, we open a new terminal window and enter the command “`$ composer -v`”, this command should output the version information message. [3, 24.]

```

Jamal@Jamal-PC ~
$ composer -v

Composer version 70a20ebcc19f1ea8ab0954a4fbdce208b30085e7 2014-03-12 16:07:58

Usage:
[options] command [arguments]

Options:
--help             -h Display this help message.
--quiet           -q Do not output any message.
--verbose         -v|vv|vvv Increase the verbosity of messages: 1 for normal ou
tput, 2 for more verbose output and 3 for debug
--version         -V Display this application version.
--ansi            Force ANSI output.
--no-ansi         Disable ANSI output.
--no-interaction -n Do not ask any interactive question.
--profile         Display timing and memory usage information
--working-dir     -d If specified, use the given directory as working directory
.

Available commands:
about             Short information about Composer
archive          Create an archive of this composer package
config           Set config options
create-project    Create new project from a package into given directory.
depends           Shows which packages depend on the given package
diagnose         Diagnoses the system to identify common errors.
dump-autoload    Dumps the autoloader
dumpautoload     Dumps the autoloader
global           Allows running commands in the global composer dir ($COMPOSER
_HOME).
help             Displays help for a command
init            Creates a basic composer.json file in current directory.
install          Installs the project dependencies from the composer.lock file
if present, or falls back on the composer.json.
licenses         Show information about licenses of dependencies
list            Lists commands
require         Adds required packages to your composer.json and installs the
m
run-script       Run the scripts defined in composer.json.
search          Search for packages
self-update      Updates composer.phar to the latest version.
selfupdate       Updates composer.phar to the latest version.
show            Show information about packages
status          Show a list of locally modified packages
update          Updates your dependencies to the latest version according to
composer.json, and updates the composer.lock file.
validate        Validates a composer.json

Jamal@Jamal-PC ~
$ |

```

Figure 5. Composer version information message

3.7 Installing Laravel 4

We may install Laravel by simply issuing the “\$ composer create-project laravel/laravel” command in our terminal followed by the name of the project, but before that we have to be sure that we change the directory to our development folder. [15]



```
/cygdrive/c/wamp/www
Jamal@Jamal-PC ~
$ cd c:/wamp/www
Jamal@Jamal-PC /cygdrive/c/wamp/www
$ composer create-project laravel/laravel ArmelSolutions
```

Figure 6. Command issued to create a new Laravel project

After a successful installation, Laravel may still require one set of permissions to be configured, that is, the folders within the app/storage directory require write access by the web server. This can be achieved by issuing the command “\$ chmod -R 755” followed by the name of the directory.” [15]



```
/cygdrive/c/wamp/www/ArmelSolutions
Application key [z3ZDxrEWhlsg26eH47tojaeqxEiT1Pdd] set successfully.
Jamal@Jamal-PC /cygdrive/c/wamp/www
$ cd c:/wamp/www/ArmelSolutions
Jamal@Jamal-PC /cygdrive/c/wamp/www/ArmelSolutions
$ chmod -R 755 app/storage
```

Figure 7. Changing the permission to access app/storage directory

3.8 Database

To create the database for our Laravel project, we simply open the phpMyadmin panel in our browser of choice and we proceed to create the database by giving it a name and editing the security credentials.

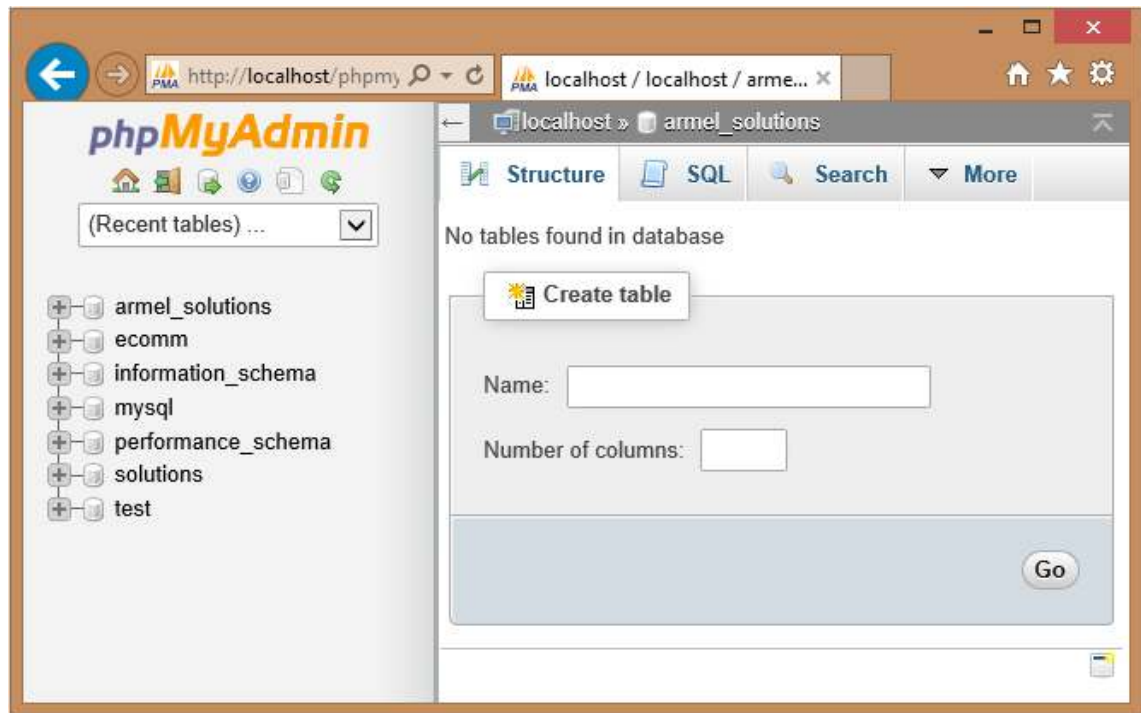


Figure 8. The created database

Before getting started, we need to be sure to configure the database connection in `app/config/database.php` file by editing the lines of PHP code containing the credentials to match our database's credentials. Laravel's default database, which is MySQL will be kept as our database management system for our current project.

To check that our application is up and running and that our Laravel installation is done properly we open our web browser and we enter the following URL: `http://localhost:8000/`. We should be greeted with Laravel's welcome message.

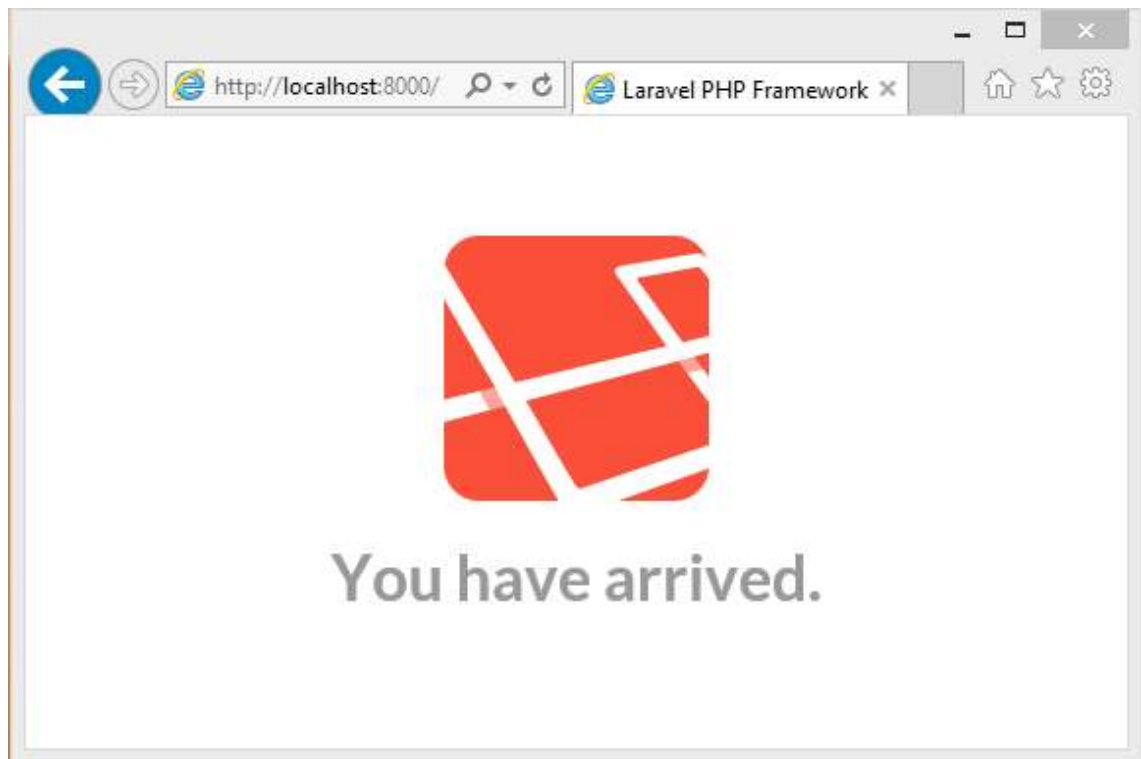


Figure 11. Screenshot of a successful Laravel installation

This ends the first half of this work where we introduced the Laravel framework and created and configured a development environment for our eCommerce project. Next, the building process of our web application will be covered fully.

4 Building the application with Laravel 4

4.1 Designing our application

4.1.1 The Idea

The aim of this project is to build a browsable database of categories and products. An administrator will be able to create pages for her/his categories and products and input simple information such as the name, availability, and image for each product. Our web application will support the basic Create-Retrieve-Update-Delete operations (CRUD). We will create also pages available for the public with the option to filter products by category and letting the users log in and pass their order and proceed to checkout using *PayPal*. All of the security, authentication, and permission features will be covered in more details in the chapter “Authentication and security”.

4.1.2 Entities, relationships and attributes

Initially, the application’s entities must be defined. An entity is a place, a thing or a single person about which data can be stored or classified and have stated relationships to other entities. From the initial requirements, we can define the following entities and their respective attributes: [3, 30.]

- Categories, which have an identifier and a name.
- Products, which have a numeric identifier, a title, a description, a price, an availability and an image.
- Users, which have a numeric identifier, a first name, a last name, an email and a telephone and whether or not he’s an admin (default is not an admin)

This information is essential in assisting us with building our database schema that will store the predefined entities, relationships, and their attributes and *Models*, that is, the PHP classes representing the objects in our database. [3, 30.]

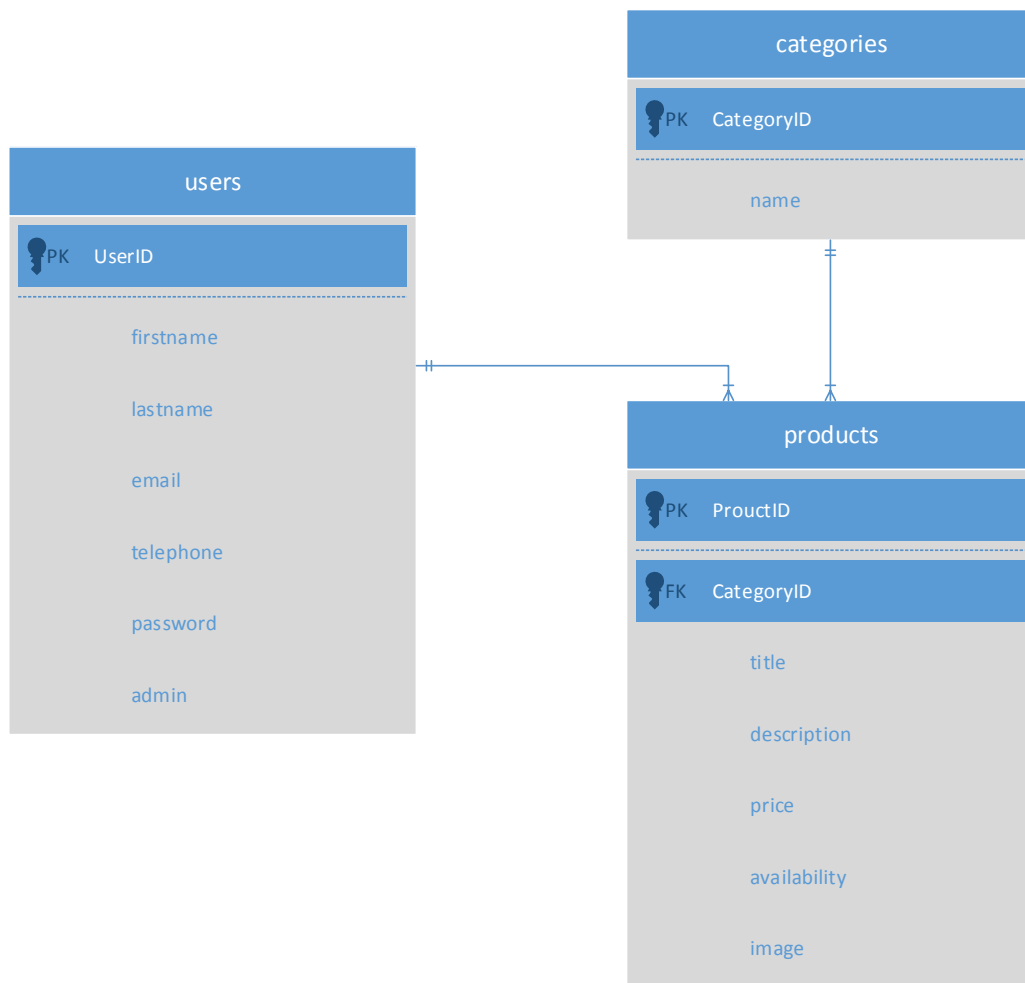


Figure 12. Relationship between constituents of the database

NB: PK->Primary Key and FK->Foreign Key.

From the diagram we can see that categories will have a “hasMany” relationship with the products and products will have a “belongsTo” relationship with categories. This will be discussed further more when we will build our *Models*.

4.1.3 Map of the application

Presently, the URL structure of our web application must be defined. There are many advantages in making sure to have expressive and clean URLs. From the usability point of view, navigating our web application will be done with ease and appear less daunting

to the visitor. They, the URLs, will also usually rank high in search engine results especially if they have appropriate keywords. We are going to utilize the following routes [18] in our web application to fulfil the initial set of requirements: [3, 31.]

Method	Route	Description
GET	/	Index
GET	admin/categories/index	Overview page
GET	admin/categories/create	Form to create a new category page
GET	admin/categories/destroy	Form to confirm deletion of page
GET	admin/products/index	Overview page
GET	admin/products/create	Form to create a new product page
PUT	admin/products/index	Form to update a product page
GET	admin/products/destroy	Form to confirm deletion of page
GET	store/category/id	Overview of single category
GET	store/view/id	Overview of single product
GET	store/cart	Overview page
GET	store/contact	Overview page

Table 2. The application's set of routes

4.2 Creating a new application

4.2.1 Creating a main view

As explained earlier in this work, *Blade* templating let us create hierarchical layouts by letting the templates be nested and/or extended. [3, 40]

The procedure is quite straight forward, that is, we copy the “index.html” file that comes with our Bootstrap installation and we save it as app/views/layouts/main.blade.php.

We will use Laravel helpers instead of regular HTML code, which will help us write more concise code and also escape from any HTML entities. We do these changes following these examples: [16]

For styles: `<link rel="stylesheet" href="css/main.css">` becomes

```
{{ HTML::style('css/main.css') }}
```

For scripts: `<script src="js/vendor/modernizr-2.6.2.min.js"></script>` becomes

```
{{ HTML::script('js/vendor/modernizr-2.6.2.min.js') }}
```

For images: `` becomes

```
{{ HTML::image('img/user-icon.gif', 'Sign In') }}
```

To define a main Blade template we use the following basic structure: [17]

```

<html>
  <body>
    @section('sidebar')
      This is the main sidebar.
    @show

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

```

Then we can use the main Blade template from within another View by using this basic structure:[17]

```

@extends('layouts.main')

@section('sidebar')
  @parent

  <p>This is appended to the main sidebar.</p>
@stop

@section('content')
  <p>This is the body content.</p>
@stop

```

The “@yield” command is a placeholder for the many sections that a nested view can fill and override. While the “@section” and “@stop” commands both define the blocks of content that are going to be injected into the main template. A schematization of this whole process can be seen in the following diagram:[3,40.]

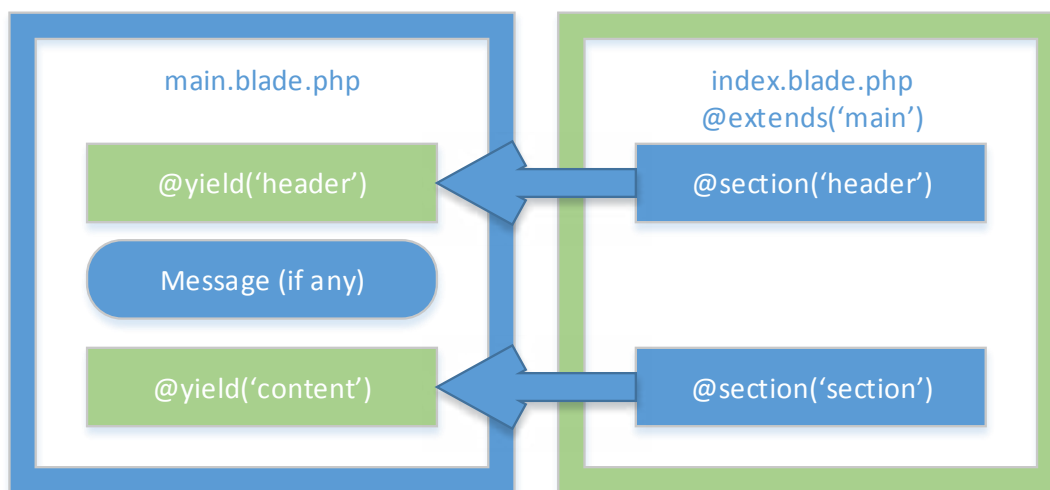


Figure 13. Blocks of content being injected into main template [3, 40]

Practically we empty our main content section in the `main.blade.php` file and replace it with `@yield('content')`. The resulting code will be the "main" template that each of our views in our web application will use. [3, 40.]

A notification area between the header and the page content has been prepared in case there is a need to inform the user about the outcome of certain actions. This flash message originates from the Session object.

The next step is to bring the other resources for our main *View*, to do so we copy all the `css/js/img/fonts` assets that come with our Bootstrap installation and we place them inside our `app/public` directory. [3, 40.]

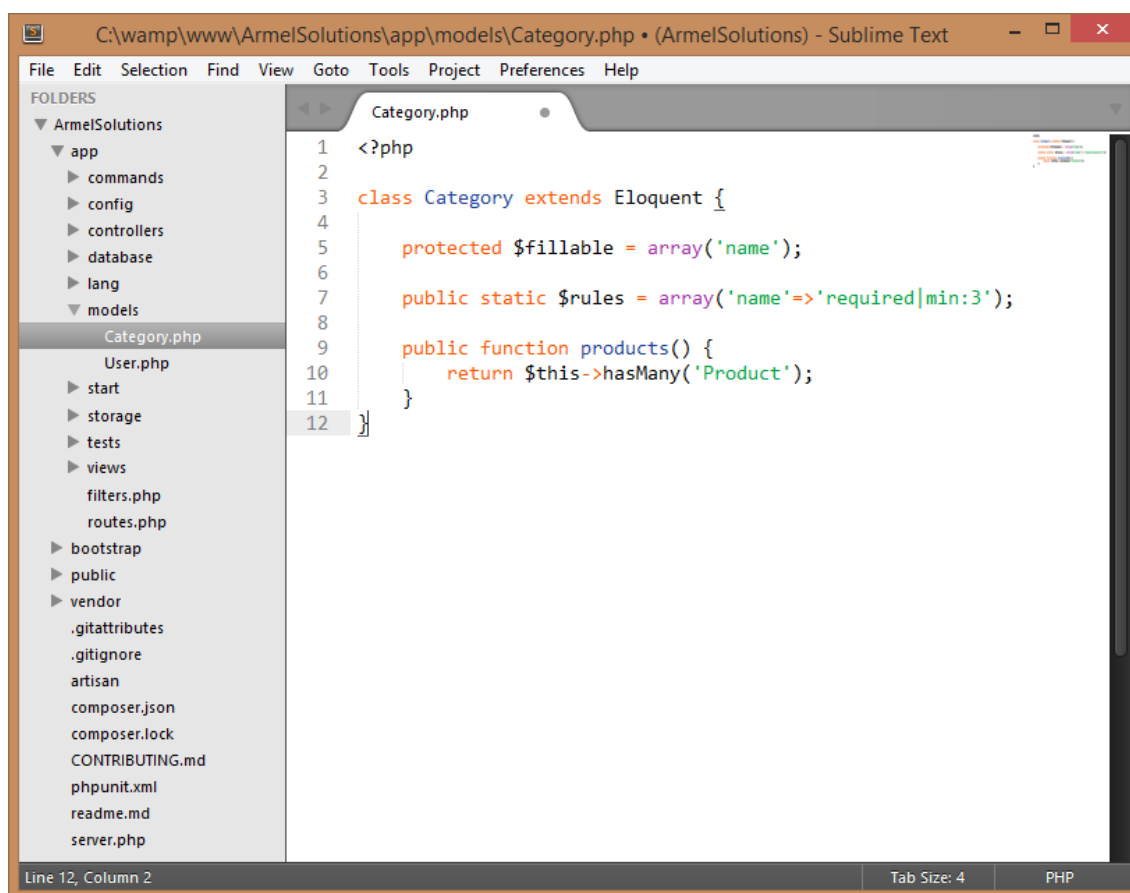
The creation of the individual views for each section of our web application will be covered in full details later in the "Creating views" section of this work.

4.2.2 Creating the Eloquent models and their respective schemas

As we have previously seen, Laravel 4 comes bundled with an ORM of its own named *Eloquent*, this powerful tool will let us define our entities, map them to their respective database tables, and manipulate them by simply using PHP methods instead of SQL syntax.

We begin with defining the models with which our application is going to interact. We previously recognised three main entities, categories, products and users. It is a convention among Laravel developers to write a model's name in the singular form; a model named Product will map to the products table in the database, and the Category model will map to the categories.

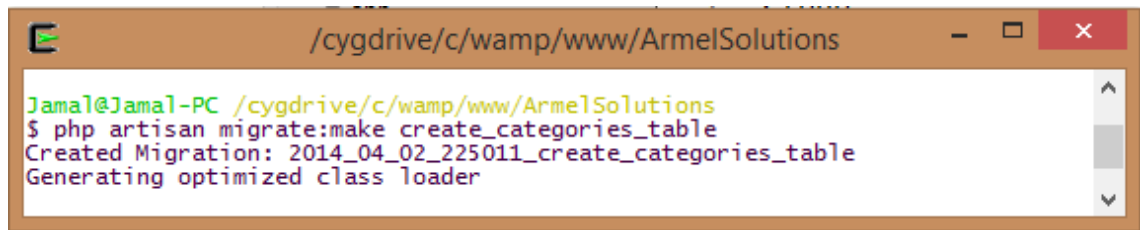
The Category model, saved inside `app/models/Category.php`, will have a “hasMany” relationship with the Product model [21]



```
1 <?php
2
3 class Category extends Eloquent {
4
5     protected $fillable = array('name');
6
7     public static $rules = array('name'=>'required|min:3');
8
9     public function products() {
10         return $this->hasMany('Product');
11     }
12 }
```

Figure 14. Category model

To create the migration for this model we issue the command “\$ php artisan migrate:make” followed by the name of the migration.



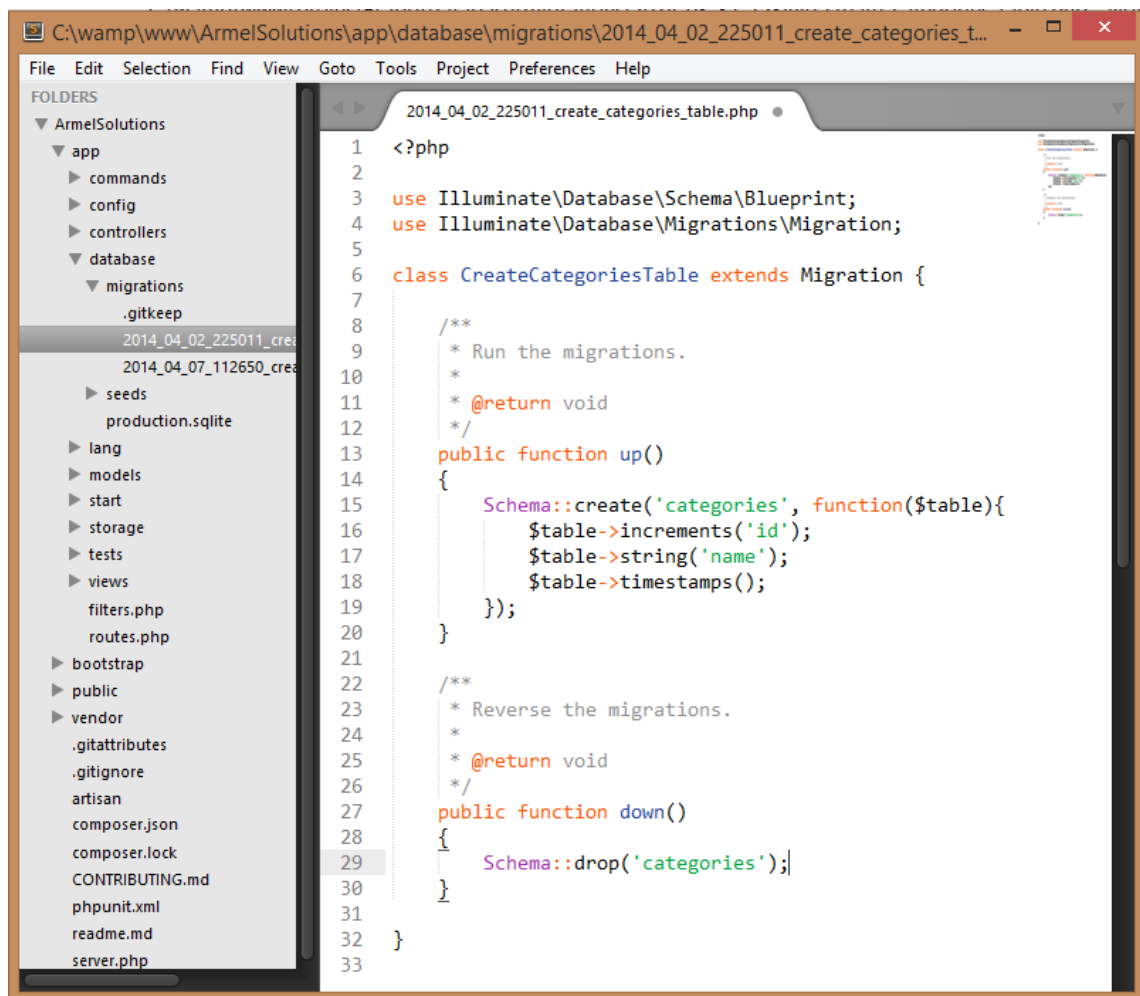
```

/cygdrive/c/wamp/www/ArmelSolutions
Jamal@Jamal-PC /cygdrive/c/wamp/www/ArmelSolutions
$ php artisan migrate:make create_categories_table
Created Migration: 2014_04_02_225011_create_categories_table
Generating optimized class loader

```

Figure 15. Migration creation

We then open the migration file inside `app/database/migrations` and we write the schema using the Schema class. [21]



```

C:\wamp\www\ArmelSolutions\app\database\migrations\2014_04_02_225011_create_categories_t...
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  ▼ ArmelSolutions
    ▼ app
      ▶ commands
      ▶ config
      ▶ controllers
      ▼ database
        ▼ migrations
          .gitkeep
          2014_04_02_225011_create_categories_table.php
          2014_04_07_112650_create_categories_table.php
        ▶ seeds
          production.sqlite
      ▶ lang
      ▶ models
      ▶ start
      ▶ storage
      ▶ tests
      ▶ views
        filters.php
        routes.php
      ▶ bootstrap
      ▶ public
      ▶ vendor
        .gitattributes
        .gitignore
        artisan
        composer.json
        composer.lock
        CONTRIBUTING.md
        phpunit.xml
        readme.md
        server.php
2014_04_02_225011_create_categories_table.php
1  <?php
2
3  use Illuminate\Database\Schema\Blueprint;
4  use Illuminate\Database\Migrations\Migration;
5
6  class CreateCategoriesTable extends Migration {
7
8      /**
9       * Run the migrations.
10      *
11      * @return void
12      */
13      public function up()
14      {
15          Schema::create('categories', function($table){
16              $table->increments('id');
17              $table->string('name');
18              $table->timestamps();
19          });
20      }
21
22      /**
23       * Reverse the migrations.
24      *
25      * @return void
26      */
27      public function down()
28      {
29          Schema::drop('categories');
30      }
31
32 }
33

```

Figure 16. Schema builder with *create* and *drop* methods

To create the table in the database all we have to do now is to issue the command “\$ php artisan migrate”.


```

/cygdrive/c/wamp/www/ArmelSolutions
Jamal@Jamal-PC /cygdrive/c/wamp/www/ArmelSolutions
$ php artisan migrate
Migration table created successfully.
Migrated: 2014_04_02_225011_create_categories_table

```

Figure 17. Artisan migrate to create the table in the database

#	Name	Type	Collation	Attributes	Null	Default	Extra
<input type="checkbox"/> 1	id	int(10)		UNSIGNED	No	None	AUTO_INCREMENT
<input type="checkbox"/> 2	name	varchar(255)	utf8_unicode_ci		No	None	
<input type="checkbox"/> 3	created_at	timestamp			No	0000-00-00 00:00:00	
<input type="checkbox"/> 4	updated_at	timestamp			No	0000-00-00 00:00:00	

Figure 18. Categories table created successfully

We follow the same previous steps to create the Product model, which is saved inside app/models/Product.php, and it will have a “belongsTo” relationship with the Category model. [21]

```

C:\wamp\www\ArmelSolutions\app\models\Product.php • (ArmelSolutions) - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
  ▼ ArmelSolutions
    ▼ app
      ▶ commands
      ▶ config
      ▶ controllers
      ▶ database
      ▶ lang
      ▼ models
        Category.php
        Product.php
        User.php
      ▶ start
      ▶ storage
      ▶ tests
      ▶ views
      filters.php
      routes.php
    ▶ bootstrap
    ▶ public
    ▶ vendor
    .gitattributes
    .gitignore
    artisan
    composer.json
    composer.lock
    CONTRIBUTING.md
    phpunit.xml
    readme.md
    server.php
Product.php
1  <?php
2
3  class Product extends Eloquent {
4
5      protected $fillable = array('category_id', 'title'
6          , 'description', 'price', 'availability', '
7          image');
8
9      public static $rules = array(
10         'category_id'=>'required|integer',
11         'title'=>'required|min:2',
12         'description'=>'required|min:20',
13         'price'=>'required|numeric',
14         'availability'=>'integer',
15         'image'=>'required|image|mimes:jpeg,jpg,bmp,
16         png,gif'
17     );
18
19     public function category() {
20         return $this->belongsTo('Category');
21     }
22 }
Line 19, Column 2
Tab Size: 4
PHP

```

Figure 19. Product model

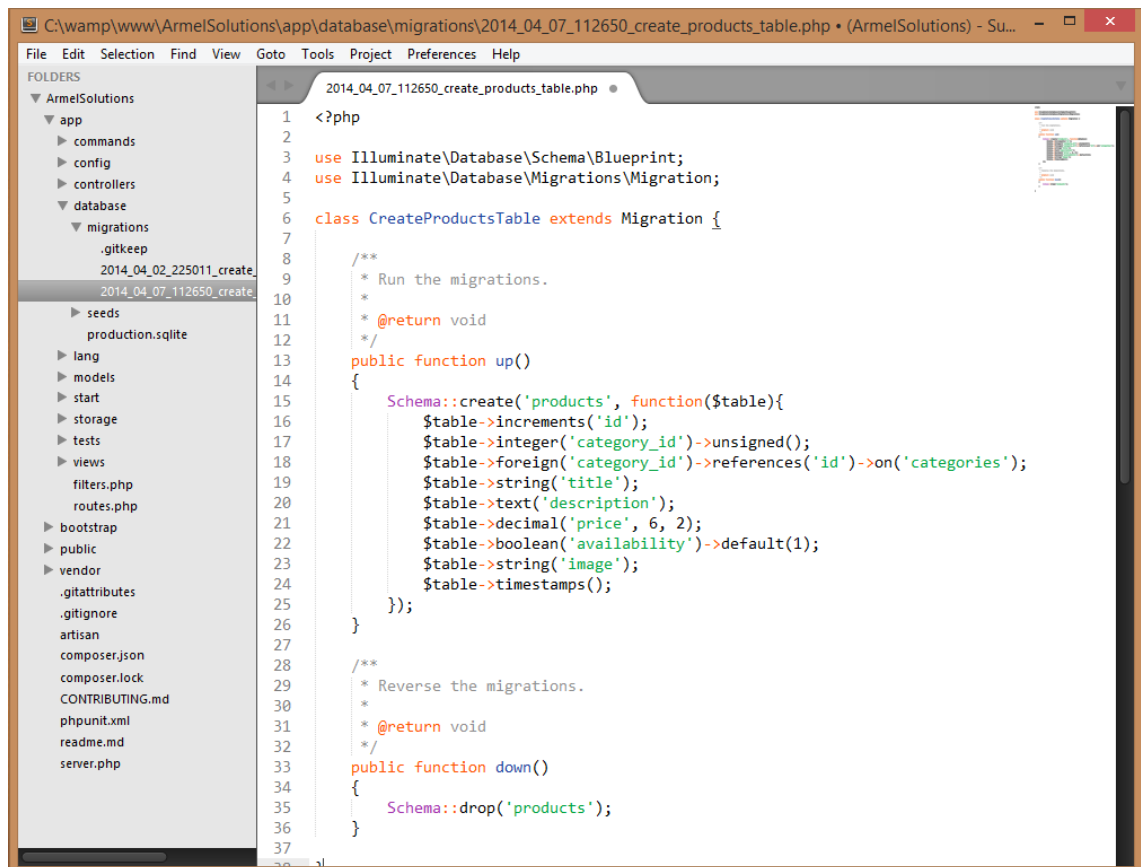


Figure 20. Product schema

The types must be always the same. So we need to be careful to always make the foreign key column unsigned when it references an incrementing integer. [6]

#	Name	Type	Collation	Attributes	Null	Default	Extra
<input type="checkbox"/> 1	id	int(10)		UNSIGNED	No	None	AUTO_INCREMENT
<input type="checkbox"/> 2	category_id	int(10)		UNSIGNED	No	None	
<input type="checkbox"/> 3	title	varchar(255)	utf8_unicode_ci		No	None	
<input type="checkbox"/> 4	description	text	utf8_unicode_ci		No	None	
<input type="checkbox"/> 5	price	decimal(6,2)			No	None	
<input type="checkbox"/> 6	availability	tinyint(1)			No	1	
<input type="checkbox"/> 7	image	varchar(255)	utf8_unicode_ci		No	None	
<input type="checkbox"/> 8	created_at	timestamp			No	0000-00-00 00:00:00	
<input type="checkbox"/> 9	updated_at	timestamp			No	0000-00-00 00:00:00	

Figure 21. Products table created successfully

The User model is a special case because of its security implications, it will be covered later in the “Authenticating users” section of this work.

4.2.3 Image managing as an example of dependency management

For our products viewed in the store we need to upload images and resize them to be able to preview them as thumbnails. To do so, we need to utilize an external package named *Intervention/image*. To add this new dependency, we must install it through *Composer*. We head to our *composer.json* file and add the following highlighted line: [4, 109.]

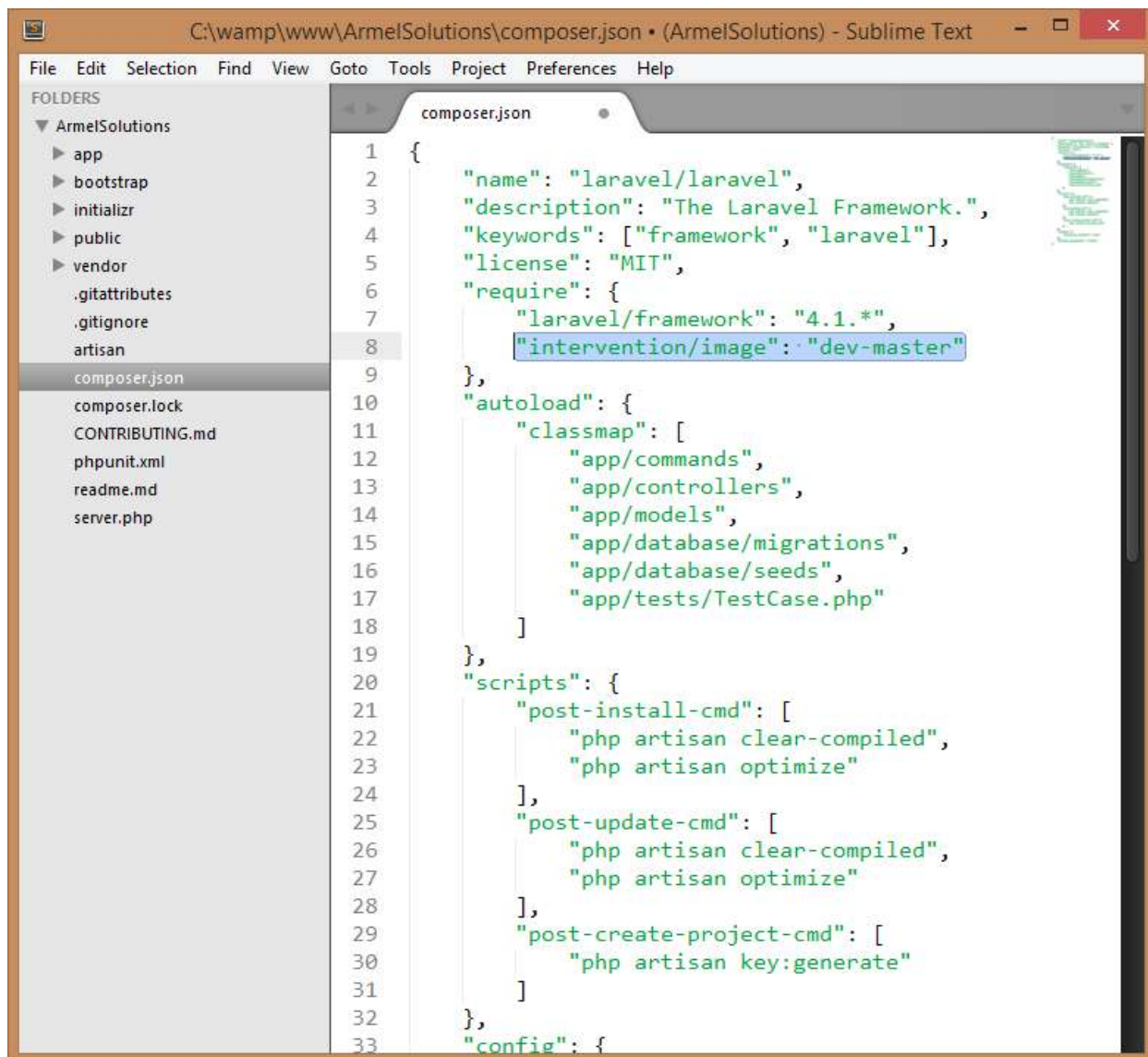


Figure 22. Adding a new dependency through Composer

And then we run the following *Composer* command in our terminal: “\$ composer update”.

This will install the *Intervention* package in the vendor directory. To make sure that Laravel autoloads it we also need to add the service provider of the class. We need to

go to `app/config/app.php` in the service provider array section and add the following highlighted line for *Intervention*:

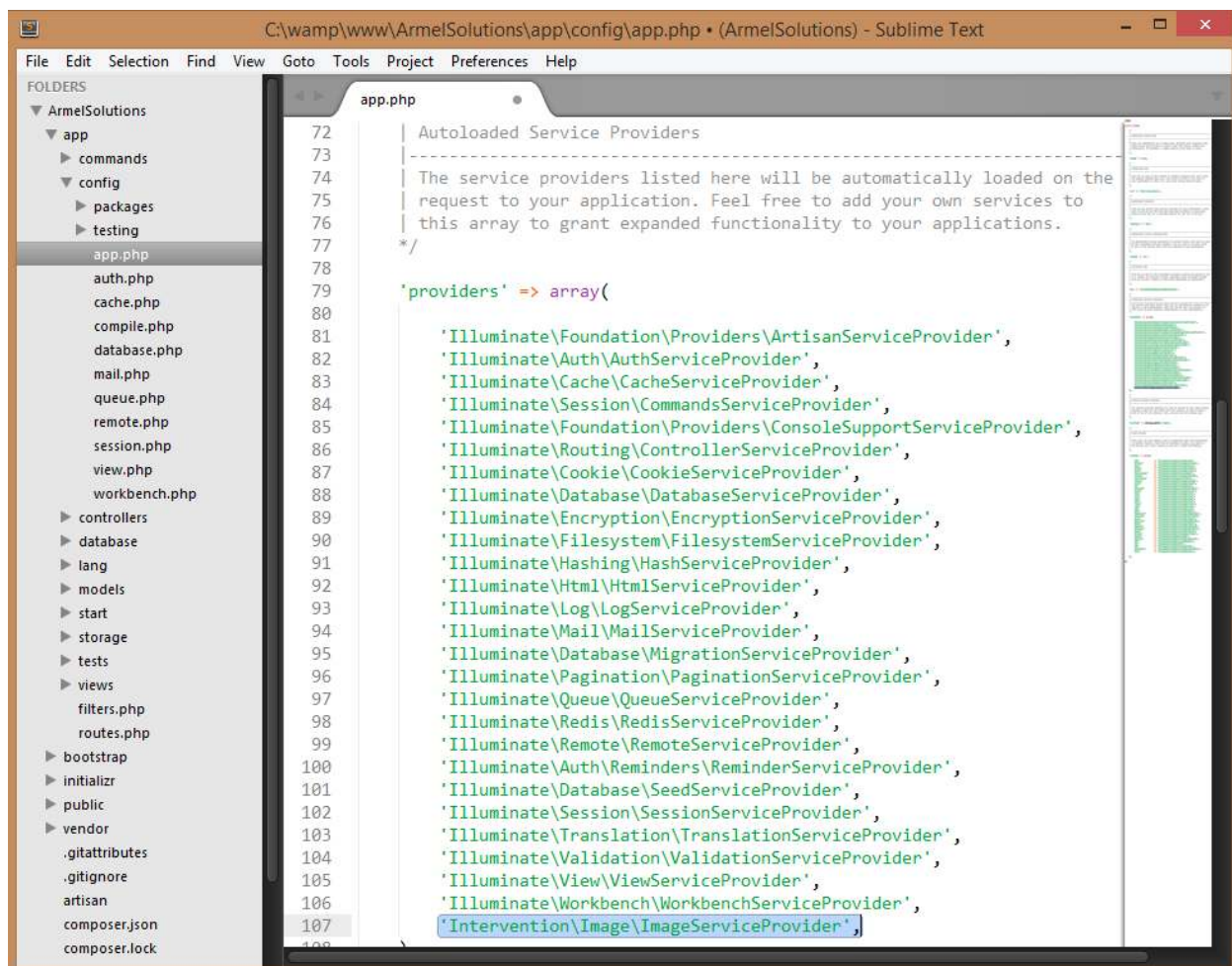


Figure 23. Binding the service provider with the Laravel setup

After finishing the previous step, we can have access to the *Intervention* library through the image alias autoloaded by Laravel. To do so, and in the class aliases section of the same file we need to add the Facades [19], so we can simply access it via an alias in our alias array: [4, 109.]

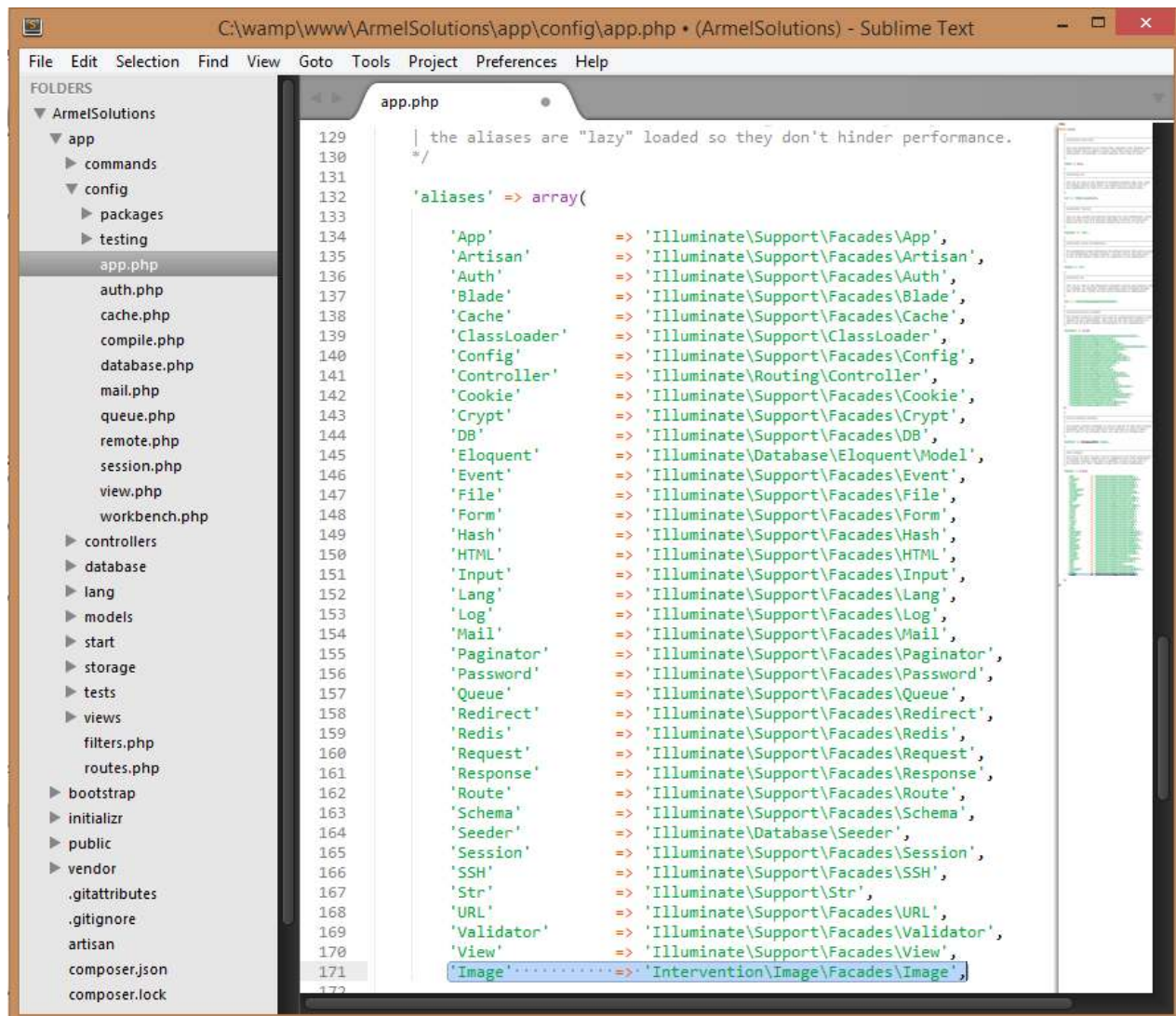


Figure 24. Adding the Facade for *Intervention*

The last step is to create a location to store our uploaded images. We will add a folder named “products” to the directory public/image/.

The image save and resize methods will be used in our products controller.

4.2.4 Creating the Controllers and their respective Routes

As we’ve seen earlier in this work, the primary function of a *Controller* is to handle requests and pass data from the *Model* to the *Views*. Thus a *Controller* can be considered as the link between our *Model* and *Views*.

- Categories controller

To create the Categories Controller, inside app/controllers directory we add the following CategoriesController.php file [21]

```

3  class CategoriesController extends BaseController {
4
5      public function __construct() {
6          $this->beforeFilter('csrf', array('on'=>'post'));
7      }
8
9      public function getIndex() {
10         return View::make('categories.index')
11             ->with('categories', Category::all());
12     }
13
14     public function postCreate() {
15         $validator = Validator::make(Input::all(), Category::$rules);
16
17         if ($validator->passes()) {
18             $category = new Category;
19             $category->name = Input::get('name');
20             $category->save();
21
22             return Redirect::to('admin/categories/index')
23                 ->with('message', 'Category Created');
24         }
25
26         return Redirect::to('admin/categories/index')
27             ->with('message', 'Something went wrong')
28             ->withErrors($validator)
29             ->withInput();
30     }
31
32     public function postDestroy() {
33         $category = Category::find(Input::get('id'));
34
35         if ($category) {
36             $category->delete();
37             return Redirect::to('admin/categories/index')
38                 ->with('message', 'Category Deleted');
39         }
40
41         return Redirect::to('admin/categories/index')
42             ->with('message', 'Something went wrong, please try again');
43     }
44 }

```

Figure 25. Categories Controller

Then we can register the corresponding route in app/routes.php file. To do so we add the following highlighted line to the aforementioned file which will indicate the URI:

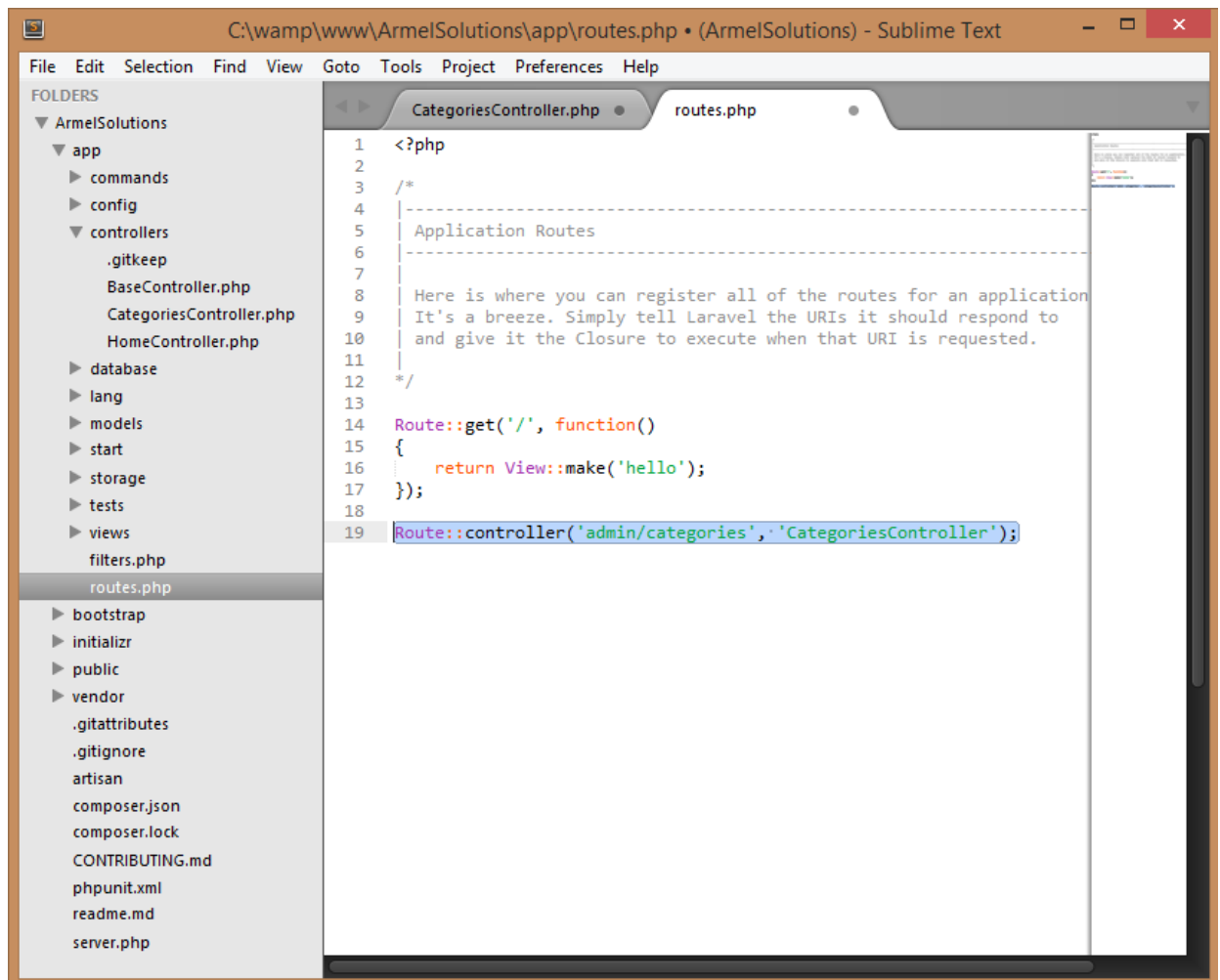


Figure 26. Tying the categories Controller's action to a set of routes

The `app/routes.php` file will be edited later to include more routes from other Controllers.

- Products controller

Similarly we create the products Controller with the following code inside `app/controllers/ProductsController.php`:

```

<?php

class ProductsController extends BaseController {

    public function __construct() {
        $this->beforeFilter('csrf', array('on'=>'post'));
    }

    public function getIndex() {
        $categories = array();

        foreach(Category::all() as $category) {
            $categories[$category->id] = $category->name;
        }

        return View::make('products.index')
            ->with('products', Product::all())
            ->with('categories', $categories);
    }

    public function postCreate() {
        $validator = Validator::make(Input::all(), Product::$rules);

        if ($validator->passes()) {
            $product = new Product;
            $product->category_id = Input::get('category id');
            $product->title = Input::get('title');
            $product->description = Input::get('description');
            $product->price = Input::get('price');

            $image = Input::file('image');
            $filename = time()."-".$image->getClientOriginalName();
            Image::make($image->getRealPath()->resize(468,249)->save(public_path().'/img/products/'.$filename);
            $product->image = 'img/products/'.$filename;
            $product->save();

            return Redirect::to('admin/products/index')
                ->with('message', 'Product Created');
        }

        return Redirect::to('admin/products/index')
            ->with('message', 'Something went wrong')
            ->withErrors($validator)
            ->withInput();
    }

    public function postDestroy() {
        $product = Product::find(Input::get('id'));

        if ($product) {
            File::delete('public/'.$product->image);
            $product->delete();
            return Redirect::to('admin/products/index')
                ->with('message', 'Product Deleted');
        }
        return Redirect::to('admin/products/index')
            ->with('message', 'Something went wrong, please try again');
    }

    public function postToggleAvailability() {
        $product = Product::find(Input::get('id'));
        if ($product) {
            $product->availability = Input::get('availability');
            $product->save();
            return Redirect::to('admin/products/index')->with('message', 'Product Updated');
        }

        return Redirect::to('admin/products/index')->with('message', 'Invalid Product');
    }
}

```


Then we update again our set of routes by adding the following line to app/routes.php:

```
Route::controller('admin/products', 'ProductsController');
```

- Store controller

```
<?php

class StoreController extends BaseController {

    public function __construct() {
        $this->beforeFilter('csrf', array('on'=>'post'));
    }

    public function getIndex() {
        return View::make('store.index')
            ->with('products', Product::take(4)->orderBy('created_at', 'DESC')->get());
    }

    public function getView($id) {
        return View::make('store.view')->with('product', Product::find($id));
    }

    public function getCategory($cat_id) {
        return View::make('store.category')
            ->with('products', Product::where('category_id', '=', $cat_id)->paginate(6))
            ->with('category', Category::find($cat_id));
    }

    public function getSearch() {
        $keyword = Input::get('keyword');

        return View::make('store.search')
            ->with('products', Product::where('title', 'LIKE', '%'.$keyword.'%')->get())
            ->with('keyword', $keyword);
    }

    public function postAddtocart() {
        $product = Product::find(Input::get('id'));
        $quantity = Input::get('quantity');

        Cart::insert(array(
            'id'=>$product->id,
            'name'=>$product->title,
            'price'=>$product->price,
            'quantity'=>$quantity,
            'image'=>$product->image
        ));

        return Redirect::to('store/cart');
    }

    public function getCart() {
        return View::make('store.cart')->with('products', Cart::contents());
    }

    public function getRemoveitem($identifier) {
        $item = Cart::item($identifier);
        $item->remove();
        return Redirect::to('store/cart');
    }

    public function getContact() {
        return View::make('store.contact');
    }
}
```

- Base Controller

To be able to view our products by category. We will start by updating the dropdown menu so that it actually uses our categories from the database instead of the static links that our layout is using. Our entire web application uses that categories dropdown menu so that means that when we go to populate it, all our Views need to have access to that categories data. In order to do this, we need to set up a before filter in our Base Controller's constructor to ensure that all of our Controllers inherit it and thus all the Views site wide will share that same categories data. [21]



Figure 27. Base Controller

Now we need to update all our controllers to instead of overwriting the Base Controller's constructor they just append to it. We add the following highlighted line to all our controllers:

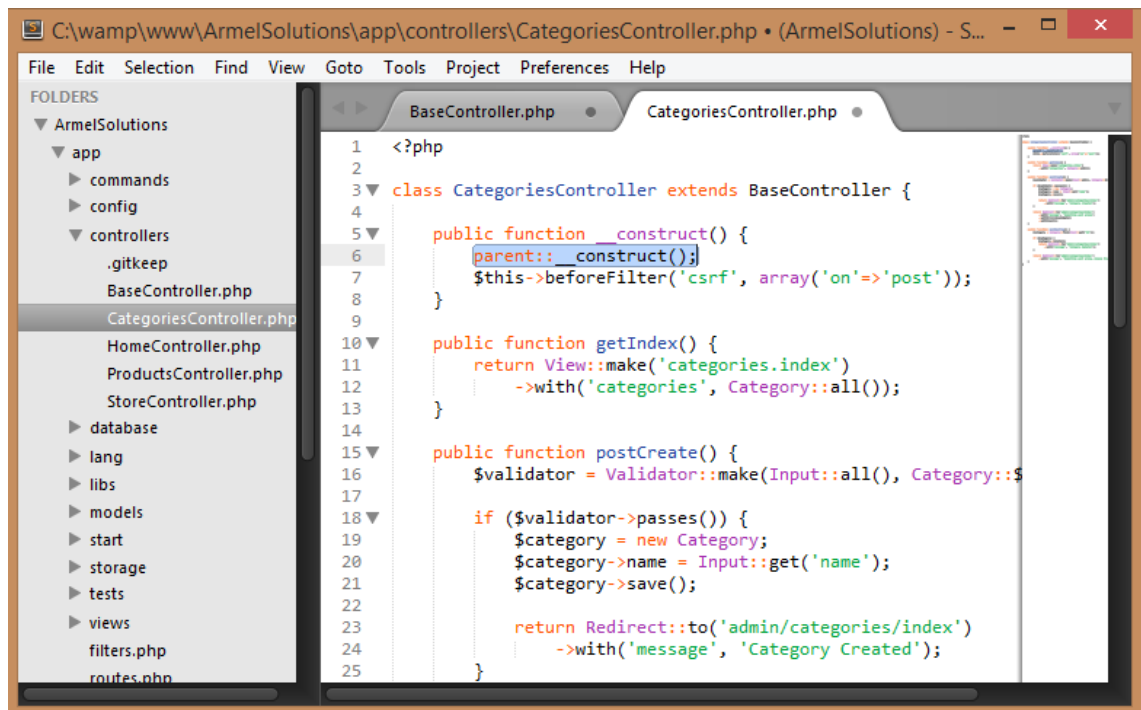


Figure 28. Appending to the Base Controller

- Users controller

The making of this Controller will be covered in full details in the “Authenticating users” section.

4.2.5 Creating the views

As was explained before in this work, *Views* receive data from a *Controller* (or *Router*) and inject it into a template, therefore, helping us to separate the business logic from the presentation layer in your web application.

- Categories View

To add our first *View*, that is, the categories view, we simply create a file called `index.blade.php` inside `app/views/categories` and add the following content to it:

This is the View used by the admin to manage the categories:

```

@extends('layouts.main')

@section('content')

    <div id="admin">

        <h1>Categories Admin Panel</h1><hr>

        <p>Here you can view, delete, and create new categories.</p>

        <h2>Categories</h2><hr>

        <ul>
            @foreach($categories as $category)
                <li>
                    {{ $category->name }} -
                    {{ Form::open(array('url'=>'admin/categories/destroy',
                    'class'=>'form-inline')) }}
                    {{ Form::hidden('id', $category->id) }}
                    {{ Form::submit('delete') }}
                    {{ Form::close() }}
                </li>
            @endforeach
        </ul>

        <h2>Create New Category</h2><hr>

        @if($errors->has())
            <div id="form-errors">
                <p>The following errors have occurred:</p>

                <ul>
                    @foreach($errors->all() as $error)
                        <li>{{ $error }}</li>
                    @endforeach
                </ul>
            </div><!-- end form-errors -->
        @endif

        {{ Form::open(array('url'=>'admin/categories/create')) }}
        <p>
            {{ Form::label('name') }}
            {{ Form::text('name') }}
        </p>
        {{ Form::submit('Create Category', array('class'=>'secondary-cart-
        btn')) }}
        {{ Form::close() }}
    </div><!-- end admin -->

@stop

```

- Products View

This is the View used by the admin to manage the products.

```

@extends('layouts.main')

@section('content')

    <div id="admin">
        <h1>Products Admin Panel</h1><hr>
        <p>Here you can view, delete, and create new products.</p>
        <h2>Products</h2><hr>
        <ul>
            @foreach($products as $product)
                <li>
                    {{ HTML::image($product->image, $product->title, array('width'=>'50')) }}
                    {{ $product->title }} -
                    {{ Form::open(array('url'=>'admin/products/destroy', 'class'=>'form-in-
line')) }}

                    {{ Form::hidden('id', $product->id) }}
                    {{ Form::submit('delete') }}

                    {{ Form::close() }} -

                    {{ Form::open(array('url'=>'admin/products/toggle-availability',
'class'=>'form-inline')) }}
                    {{ Form::hidden('id', $product->id) }}
                    {{ Form::select('availability', array('1'=>'In Stock', '0'=>'Out of Stock'),
$product->availability) }}
                    {{ Form::submit('Update') }}
                    {{ Form::close() }}

                </li>
            @endforeach
        </ul>
        <h2>Create New Product</h2><hr>
        @if($errors->has())
            <div id="form-errors">
                <p>The following errors have occurred:</p>

                <ul>
                    @foreach($errors->all() as $error)
                        <li>{{ $error }}</li>
                    @endforeach
                </ul>
            </div><!-- end form-errors -->
        @endif

        {{ Form::open(array('url'=>'admin/products/create', 'files'=>true)) }}
        <p>
            {{ Form::label('category_id', 'Category') }}
            {{ Form::select('category_id', $categories) }}
        </p>
        <p>
            {{ Form::label('title') }}
            {{ Form::text('title') }}
        </p>
        <p>
            {{ Form::label('description') }}
            {{ Form::textarea('description') }}
        </p>
        <p>
            {{ Form::label('price') }}
            {{ Form::text('price', null, array('class'=>'form-price')) }}
        </p>
        <p>
            {{ Form::label('image', 'Choose an image') }}
            {{ Form::file('image') }}
        </p>
        {{ Form::submit('Create Product', array('class'=>'secondary-cart-btn')) }}
        {{ Form::close() }}
    </div><!-- end admin -->
@stop

```

- Store Views

All the Store Views listed here are those Views that a non-admin user can View and interact with. The following code for the Index View is used as the basis for the following Views.

```
@extends('layouts.main')

@section('promo')

    <section id="promo">
        <div id="promo-details">
            <h1>Today's Deals</h1>
            <p>Checkout this section of<br />
            products at a discounted price.</p>
            <a href="#" class="default-btn">Shop Now</a>
        </div><!-- end promo-details -->
        {{ HTML::image('img/promo.png', 'Promotional Ad')}}
    </section><!-- promo -->

@stop

@section('content')

    <h2>New Products</h2>
    <hr>
    <div id="products">
        @foreach($products as $product)
            <div class="product">
                <a href="/store/view/{{ $product->id }}">
                    {{ HTML::image($product->image, $product->title, array('class'=>'feature', 'width'=>'240', 'height'=>'127')) }}
                </a>

                <h3><a href="/store/view/{{ $product->id }}">{{ $product->title
            }}</a></h3>

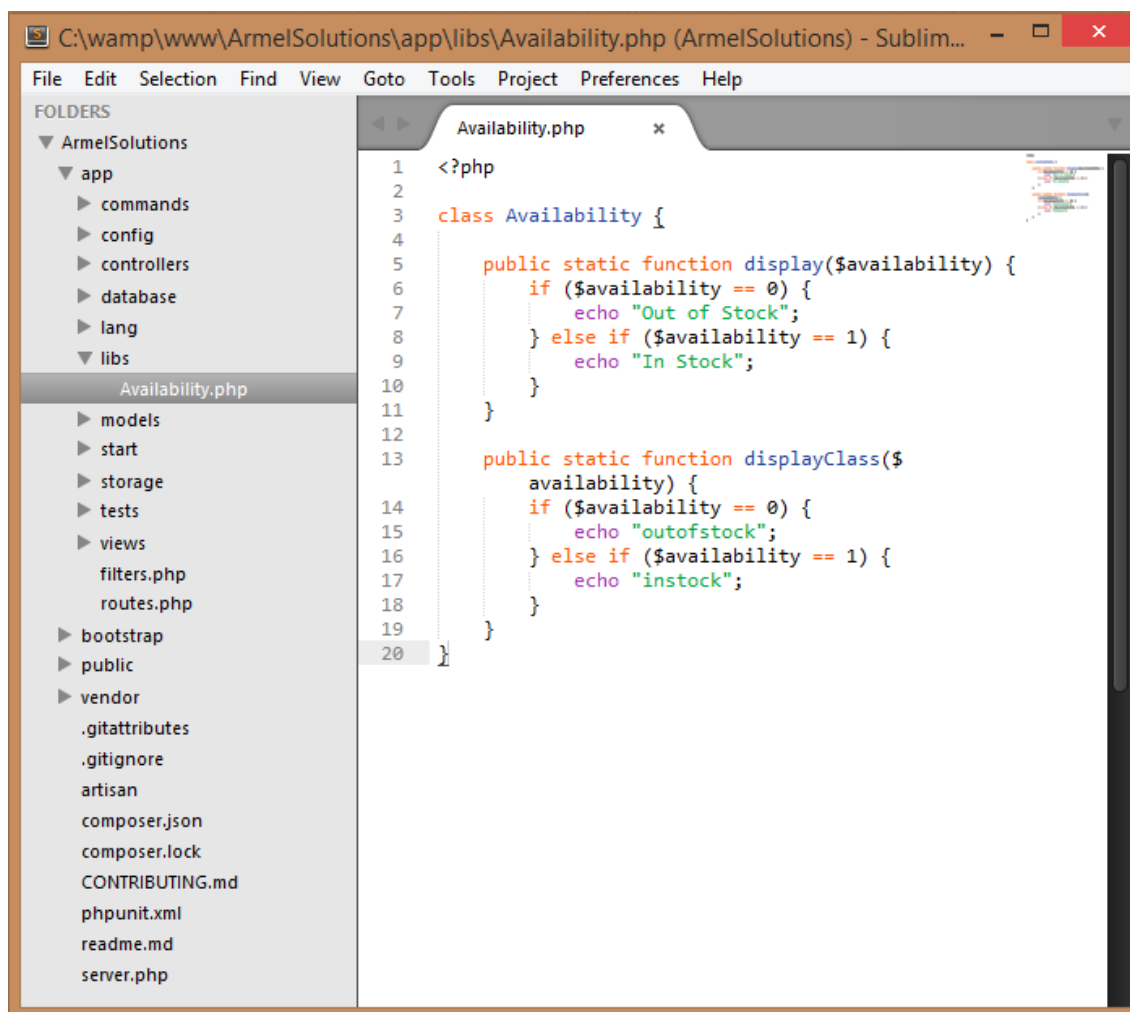
                <p>{{ $product->description }}</p>

                <h5>
                    Availability:
                    <span class="{{ Availability::displayClass($product->availability) }}">
                        {{ Availability::display($product->availability) }}
                    </span>
                </h5>

                <p>
                    {{ Form::open(array('url'=>'store/addtocart')) }}
                    {{ Form::hidden('quantity', 1) }}
                    {{ Form::hidden('id', $product->id) }}
                    <button type="submit" class="cart-btn">
                        <span class="price">{{ $product->price }}</span>
                        {{ HTML::image('img/white-cart.gif', 'Add to Cart') }}
                        ADD TO CART
                    </button>
                    {{ Form::close() }}
                </p>
            </div>
        @endforeach
    </div><!-- end products -->

@stop
```

For the “Availability” class which can be inStock or outOfStock. Our availability filed value in the database is either 0 or 1. So we need to write two helper methods, one which will return either the inStock or outOfStock class name and another one which will return the value In Stock or Out of Stock which we can use to display inside this View. We create a new folder named libs inside the app directory to hold our personal libraries. And we add this Availability file: [21]



```

1 <?php
2
3 class Availability {
4
5     public static function display($availability) {
6         if ($availability == 0) {
7             echo "Out of Stock";
8         } else if ($availability == 1) {
9             echo "In Stock";
10        }
11    }
12
13    public static function displayClass($
14        availability) {
15        if ($availability == 0) {
16            echo "outofstock";
17        } else if ($availability == 1) {
18            echo "instock";
19        }
20    }

```

Figure 29. Availability class with the helper methods

Then we need to make sure that Laravel downloads it for us. To do so we go to app/start/global.php file and we add the following highlighted line:

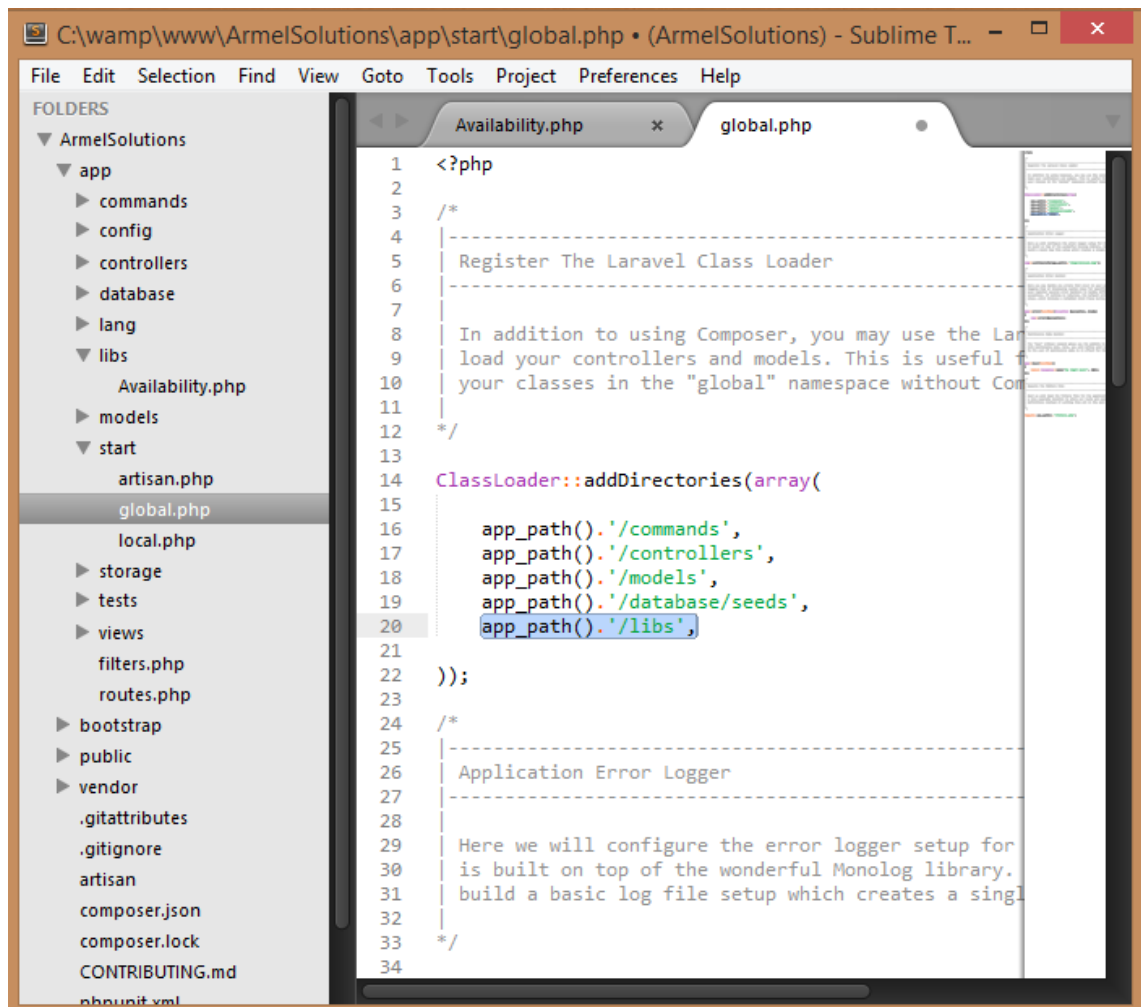


Figure 30. Adding a path to our libraries folder

And similarly, we add the other Views for the Store which include:

- Category: Where the user can view the products by category.
- View: Where the user can view products individually.
- Search: Where the user can search the whole website using a keyword.
- Cart: Where the user can view and edit the details of her/his order before checking out (through molting package and using class Cart).
- Contact: Where the user can find the details for contacting the business.

- Users View

The making of this View will be covered in full details in the “Authenticating users” section.

4.3 Authentication and security

4.3.1 Authenticating users

Now we will work on our authentication system. In order for the customers to place orders and review their previous order history they will need to have an account. First we are going to need a table to store our users' data in.

We create a new migration file by issuing the following command in our terminal:

```
$ php artisan migrate:make create_users_table
```

Then we build our users Model schema by editing the created migration.

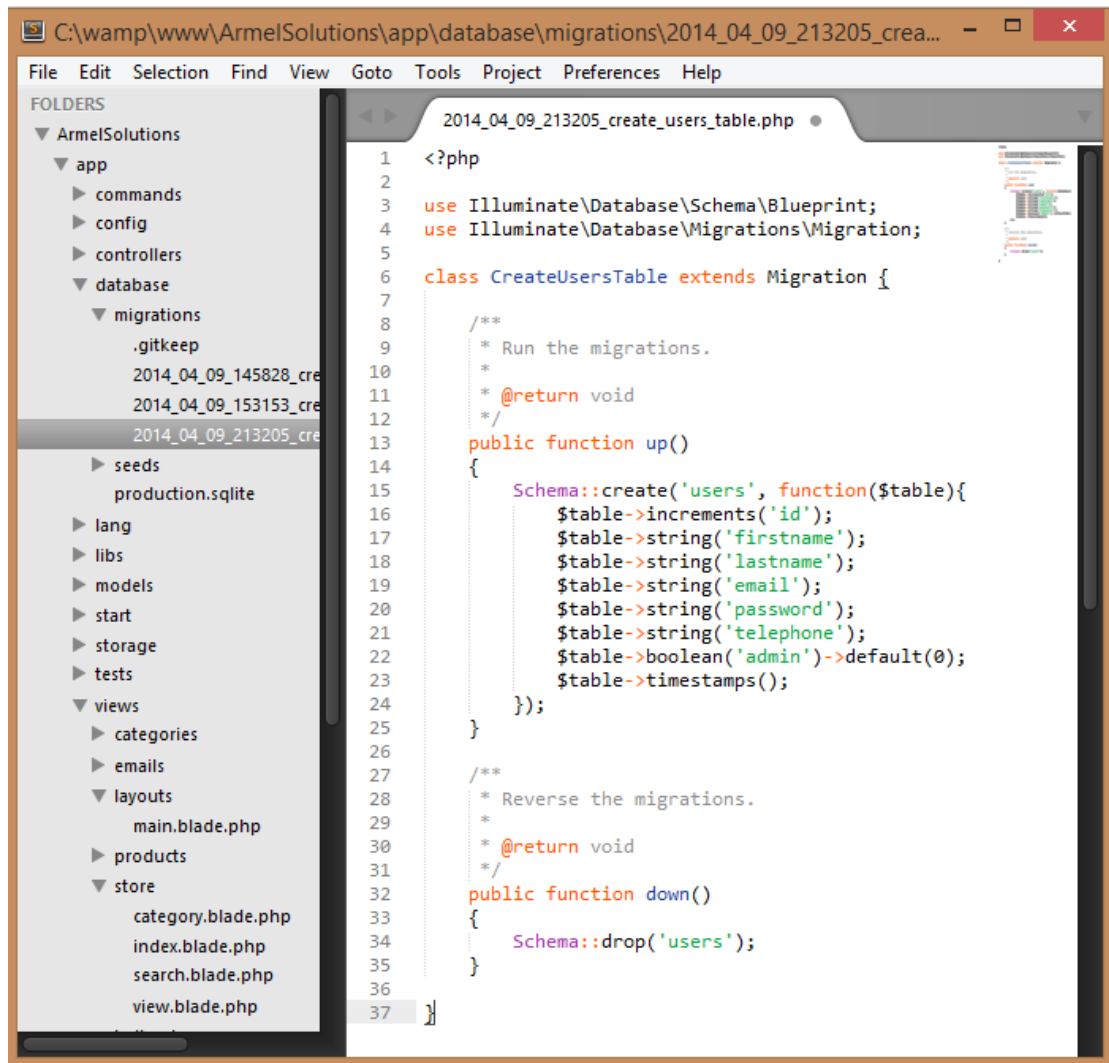


Figure 31. Schema for the users table

As was mentioned earlier in this work, by default users are not administrators (default).

We run the migration to create the table in our database using the following command:

```
$ php artisan migrate
```

Since we do not have any backend data yet, we are going to use a Seeder to populate it with some data. We issue the following command:



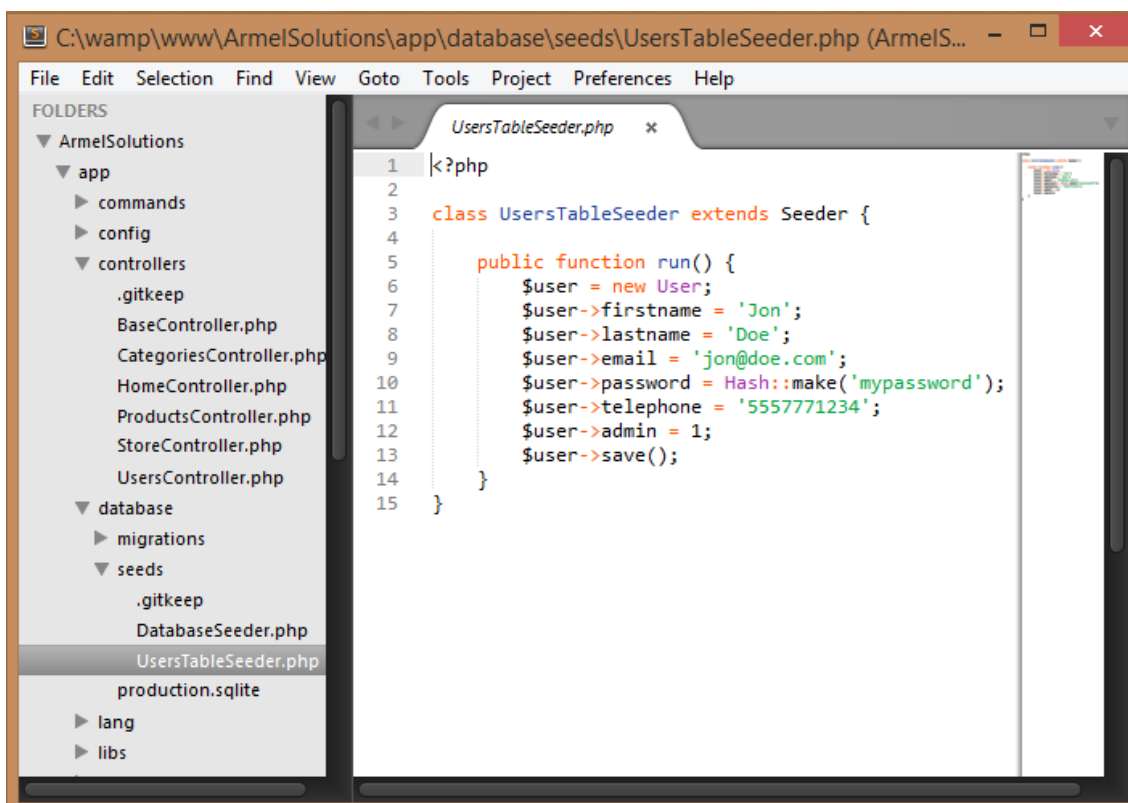
```

/cygdrive/c/wamp/www/ArmelSolutions
Jamal@Jamal-PC /cygdrive/c/wamp/www/ArmelSolutions
$ php artisan db:seed
Seeded: UsersTableSeeder

```

Figure 32. Seeding the database

We then create under app/database/seeds directory the Users Table Seeder file containing the following code:



```

C:\wamp\www\ArmelSolutions\app\database\seeds\UsersTableSeeder.php (ArmelS...
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
▼ ArmelSolutions
  ▼ app
    ► commands
    ► config
    ▼ controllers
      .gitkeep
      BaseController.php
      CategoriesController.php
      HomeController.php
      ProductsController.php
      StoreController.php
      UsersController.php
  ▼ database
    ► migrations
    ▼ seeds
      .gitkeep
      DatabaseSeeder.php
      UsersTableSeeder.php
      production.sqlite
  ► lang
  ► libs
UsersTableSeeder.php x
1 <?php
2
3 class UsersTableSeeder extends Seeder {
4
5     public function run() {
6         $user = new User;
7         $user->firstname = 'Jon';
8         $user->lastname = 'Doe';
9         $user->email = 'jon@doe.com';
10        $user->password = Hash::make('mypassword');
11        $user->telephone = '5557771234';
12        $user->admin = 1;
13        $user->save();
14    }
15 }

```

Figure 33. Adding an admin through the Seeder

Then we make our Categories and Products admin panels accessible only to the logged-in admins. We do so by adding the following highlighted line to both files:

```

1 <?php
2
3 class CategoriesController extends BaseController {
4
5     public function __construct() {
6         parent::__construct();
7         $this->beforeFilter('csrf', array('on'=>'post'));
8         $this->beforeFilter('admin');
9     }
10
11     public function getIndex() {
12         return View::make('categories.index')
13             ->with('categories', Category::all());
14     }
15
16     public function postCreate() {
17         $validator = Validator::make(Input::all(), Category
18
19         if ($validator->passes()) {
20             $category = new Category;
21             $category->name = Input::get('name');
22             $category->save();
23
24             return Redirect::to('admin/categories/index')
25                 ->with('message', 'Category Created');
26         }

```

Figure 34. An admin *before* filter

Then we need to go to the app/filters.php file to add the admin filter's route.

```

47 /*
48 |-----
49 | Guest Filter
50 |-----
51 |
52 | The "guest" filter is the counterpart of the authentication filters as
53 | it simply checks that the current user is not logged in. A redirect
54 | response will be issued if they are, which you may freely change.
55 |
56 */
57
58 Route::filter('guest', function()
59 {
60     if (Auth::check()) return Redirect::to('/');
61 });
62
63 /* Admin Filter */
64
65 Route::filter('admin', function()
66 {
67     if (!Auth::user() || Auth::user()->admin != 1) return Redirect::to('/');
68 });
69
70
71
72 /*

```

Figure 35. Adding the admin filter.

After all the above steps only the logged-in admin can access the categories and products panels. Other users will be redirected to the Store view.

4.3.2 Securing the application

Our web application in its present form has a number of vulnerable endpoints. And they cannot be addressed all in this work but the most serious one will be fixed here. Attacks are conducted by targeting a URL that has side-effects (that is, it is performing an action and not just displaying information). First of all, all the URLs that handle user input are not checking this CSRF token.

To address this Cross-site request forgery (CSRF) we add the following highlighted line to all our Controllers: [3, 58]

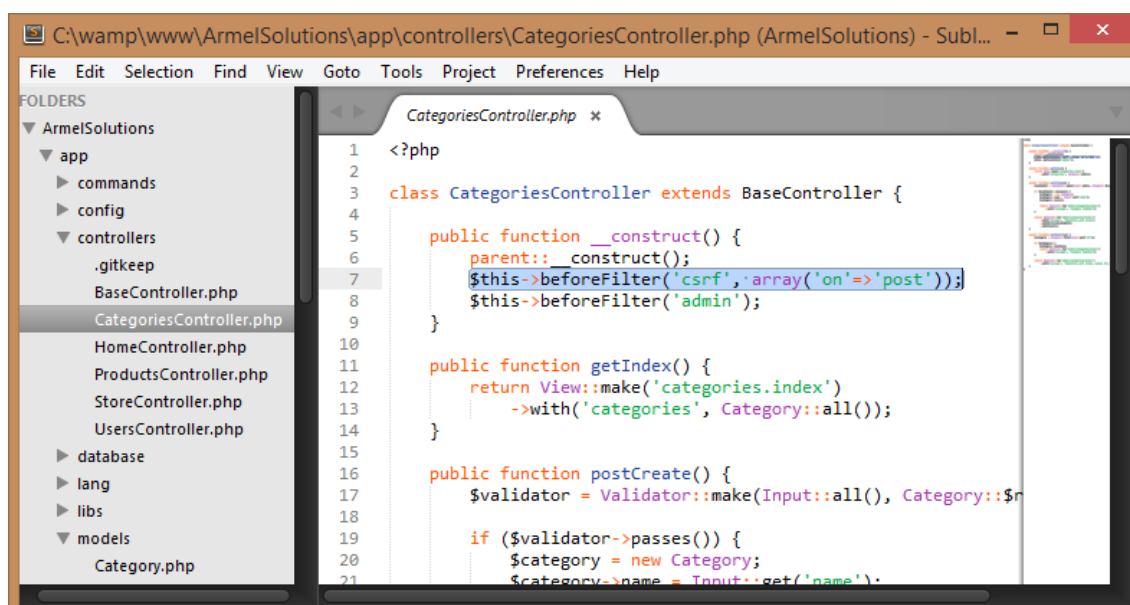
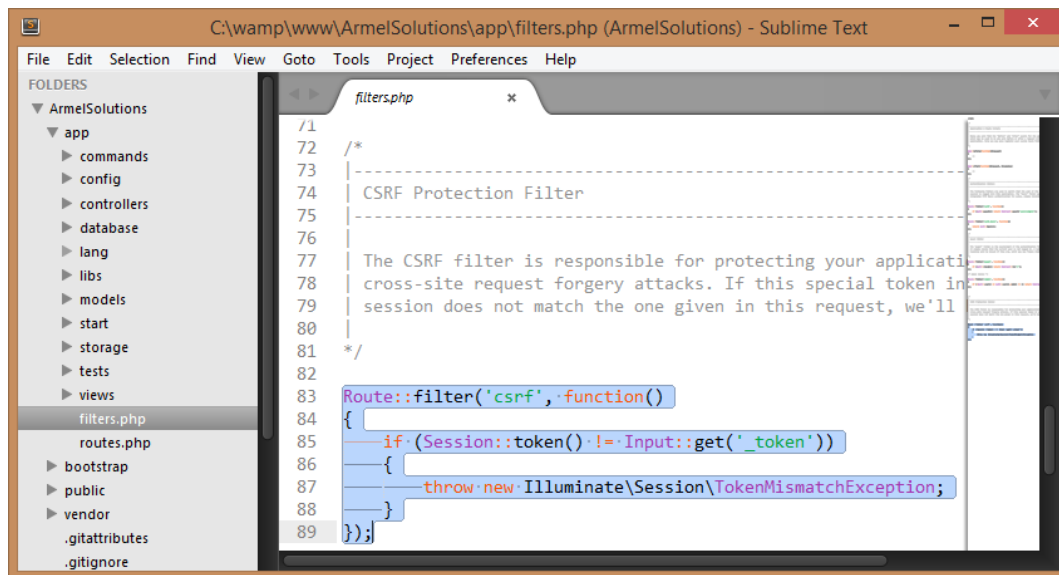


Figure 36. Adding a CSRF *before* filter

Then we need to go to the app/filters.php file to add the CSRF filter's route.



```

C:\wamp\www\ArmelSolutions\app\filters.php (ArmelSolutions) - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
▼ ArmelSolutions
  ▼ app
    ► commands
    ► config
    ► controllers
    ► database
    ► lang
    ► libs
    ► models
    ► start
    ► storage
    ► tests
    ► views
  filters.php
  routes.php
  bootstrap
  public
  vendor
  .gitattributes
  .gitignore

filters.php
71
72 /*
73 |-----
74 | CSRF Protection Filter
75 |-----
76 |
77 | The CSRF filter is responsible for protecting your application
78 | cross-site request forgery attacks. If this special token in
79 | session does not match the one given in this request, we'll
80 |
81 */
82
83 Route::filter('csrf', function()
84 {
85     if (Session::token() != Input::get('_token'))
86     {
87         throw new Illuminate\Session\TokenMismatchException;
88     }
89 });

```

Figure 37. The CSRF filter

And that concludes our process of building an eCommerce web application using Laravel 4 framework.

5 Conclusion

Working on this project I faced a typical PHP developer problem, which is to be able to build a decent looking and feature rich web application in a few days. I needed to find a modern PHP framework that would let rapid developing, while also providing options for expandability on a large scale. After examining different PHP frameworks and comparing their abilities at handling an MVC architecture pattern I came up with the ideal choice for a PHP MVC framework, which is Laravel. At first, learning a new framework might seem an overwhelming task, but it was not the case with Laravel, thanks to its clear and concise documentation, and its developers that make a lively active community. Furthermore, I found a good CRUD web application on GitHub which appeared to be a good introduction to Laravel's world. The said application uses twitter's Bootstrap as well and it was a great help while developing this project. [20]

Early on in the development process with Laravel, one would feel at ease with its simplicity and ease of use. My own experience with another big framework, that is, .NET framework is that one ends up investing an important amount of time struggling with incomprehensible XAML configuration settings, complex syntax, unfinished documentation, and a feeling in the end that the framework's purpose of saving time and effort was not truly achieved. It is the other way around with Laravel, which is actually one of its major strengths. My own experience with Laravel is that it made my development process a more enjoyable experience.

Laravel is lightweight enough not to undermine the project's planning and development process yet it does still offer an adequate structure and balanced amount of built-in features which let one pay more attention to the business logic part of their web application rather than waste too much time with the tedious basics and reinventing the wheel each time when starting a new project. Among these features, we can mention Laravel's very own ORM, named *Eloquent* which is a simple implementation of PHP ActiveRecord, which works in a simple yet effective way. Indeed, the schema for our project was not very complex but not very basic either and yet no problems were encountered. Laravel is also *Composer* ready which comes in handy in managing the dependency of our project's dependencies. Other features worth mentioning are *Artisan*, *Blade*, authentication and security.

The requirements of our project were to create a CRUD eCommerce web application for the Armel Solutions freelance start-up. It required also admin panels for the creation and deletion of new categories and products. Authenticating users and accepting their orders.

I succeeded in building a browsable web application that fulfils all the requirements in a relatively short period of time. The majority of that time was in fact spent on planning the business logic of the application and its data modelling. Minimal time was allocated for the development process itself.

Although developing with Laravel was a great experience, there is still room for improvement for example, when having a closer look at the documentation, the transition between the introductory “getting started” section and the documentation for the API itself is quite abrupt.

Another problem faced is the rarity of academic references for Laravel 4, which might improve with time especially if we take into account the fact that Laravel is a relatively young framework.

References

1 Introduction to Laravel [online].

URL: <http://laravel.com/docs/introduction>

Accessed: 3 April 2014.

2 Architecture of Laravel Applications [online].

URL: <http://laravelbook.com/laravel-architecture/>

Accessed: 3 April 2014.

3 Raphaël S. Getting Started with Laravel 4. Packt Publishing Limited, Birmingham 2014.

4 Hardik D. Learning Laravel 4 application development. Packt Publishing Limited, Birmingham 2013.

5 Eloquent [online]

URL: <http://laravel.com/docs/eloquent>

Accessed: 3 April 2014.

6 Schema Builder [online]

URL: <http://laravel.com/docs/schema>

Accessed: 3 April 2014.

7 The PHP package archivist [online]

URL: <https://packagist.org/>

Accessed: 3 April 2014.

8 Getting started with Composer [online].

URL: <https://getcomposer.org/doc/00-intro.md>

Accessed: 3 April 2014.

9 Cygwin [online]

URL: <http://www.redhat.com/services/custom/cygwin/>

Accessed: 3 April 2014.

10 David C, Ian W. Bootstrap site blueprints. Packt Publishing Limited, Birmingham 2014.

11 Initializr [online]

URL: <http://www.initializr.com>

Accessed: 4 April 2014.

12 Getting started with Bootstrap [online]
URL: <http://getbootstrap.com/getting-started/>
Accessed: 4 April 2014.

13 What is MySQL? [online]
URL: <http://dev.mysql.com/doc/refman/5.6/en/what-is-mysql.html>
Accessed: 4 April 2014.

14 WAMPserver [online]
URL: <http://www.wampserver.com/en/>
Accessed: 4 April 2014.

15 Laravel installation [online]
URL: <http://laravel.com/docs/installation>
Accessed: 5 April 2014.

16 Helper functions [online]
URL: <http://laravel.com/docs/helpers>
Accessed: 5 April 2014.

17 Templates [online]
URL: <http://laravel.com/docs/templates>
Accessed: 5 April 2014.

18 Routing [online]
URL: <http://laravel.com/docs/routing>
Accessed: 5 April 2014.

18 Facades [online]
URL: <http://laravel.com/docs/facades>
Accessed: 6 April 2014.

20 Laravel 4 E-Commerce [online]
URL: <https://medium.com/laravel-4/c5afca925f28>
Accessed: 6 April 2014.

21 Build an eCommerce App in Laravel [online]
URL: <https://tutsplus.com/course/laravel-ecommerce-application/>
Accessed: 6 April 2014.