Getting started with Laravel

after writing almost no PHP code for the last about 10 years, i decided it's time to learn some laravel, to get back on the wagon. These are my notes along the way of learning Laravel ans an old PHP programmer.

As an initial overview I watched the Laravel 6 for Beginners Playlist of The Net Ninja and i have to say, i like the way he structures his tutoral and i also like his accent.. makes me smile all the time ;)

The tutorial was published in January 2020, i am watching it now in November 2020. The tutoral covers Laravel 6, i have now installed Laravel 8 on my machine and they have changed from using bootstrap to using Laravel Jetstream and Tailwind. Bootstrap can still be used which is what i am doing, because i like the bootstrap style and i want to customize some open source scripts which are based on boostrap + laravel. so i don't care about Tailwind and Jetstream for the moment.

as with most successful programming languages and frameworks, Laravel seems to provide a good documentation. just go to the Laravel webpage and click on documentation.. the documentation link contains the version number, hence i won't hard link it here as it seems this would be obsolete in a few months anyway :)

Laravel installation (on Ubuntu 20.04 with Apache2)

I don't care or like the included web-server in laravel for development.. it isn't that much work to set up a proper webserver like you would have on your productive environment, so why not develop with a close to production environment as possible, so you can fix issues beforehand?

installing laravel is pretty straight forward.. I first Installed Adminer on a plain ubuntu installation which depends on everything you need to have a fully working Linux + Apache + PHP + MariaDB (MySQL) Web-Server.

I then installed certbot and used it to configure apache for ssl and also create a set of signed certificates from Let's Encrypt. They provide free SSL certificates which are valid for I think 30 days only but certbot takes care of the autoamtic renewal, so in the end this is a better solution than a long-time certificate which you have to renew manually as soon as you have forgotten what you did when you originally set it up .. with certbot, you set up your SSL certs and the auto-renewal service once and then forget about it forever, and forgetting things is something i'm really good at :)

```
snap install --classic certbot
certbot --apache
```

I want to edit the webpages as a regular user on the server, so i symlinked `/var/www` to my home directory. I also change-owned `/var/www` with all its sub-directories and files to my own user.

after that I've installed composer to install laravel using my regular user:

```
sudo apt install php-xml php-mbstring
cd www
composer global require laravel/installer
nano ~/.profile
```

add this line to the end:

```
PATH="${PATH}:~/.config/composer/vendor/bin"
```

also execute this on the terminal to continue without logging out and in again:

```
export PATH="${PATH}:~/.config/composer/vendor/bin"
```

edit the apache config in `/etc/apache2/sites-enabled/000-default-le-ssl.conf` and add this section under the `DocumentRoot` directive:

```
    <Directory /var/www/mysite/public>
        Options Indexes FollowSymLinks MultiViews
            AllowOverride All
            Order allow,deny
            allow from all
            Require all granted
        </Directory>
```

create first project

```
cd www
laravel new mysite
```

install nvm, npm and node.js as well as yarn **as root**:

```
export XDG_CONFIG_HOME="/opt"
apt install curl gnupg2
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/$(curl -s
https://api.github.com/repos/nvm-sh/nvm/releases/latest | grep "tag_name" |
cut -d : -f 2,3 | tr -d ' ",')/install.sh | bash
echo 'export NVM_DIR="'$XDG_CONFIG_HOME'/nvm"
[ -s "$NVM_DIR/nvm.sh" ] && . "$NVM_DIR/nvm.sh"  # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && . "$NVM_DIR/bash_completion"  # This
loads nvm bash_completion' > /etc/profile.d/nvm.sh
source /etc/profile.d/nvm.sh
nvm install node
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add -
echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee
/etc/apt/sources.list.d/yarn.list
apt update && apt install --no-install-recommends yarn
```

now **back to the regular user** to install bootstarp

```
cd ~/www/mysite
composer require laravel/ui
php artisan ui bootstrap
npm install
npm run dev
```

now comes something that we would not have had to do if we where running the laravel built-in dev server as unprivileged user.. we now run into some privilege issues because the web-server is running as `www-data` and I am using a different user to edit the files in `/var/www`. All files are readable to the web-browser but they aren't writeable and that is a problem when it comes to laravel's log directory and bootstrap's cache directory, so we have to fix the permissions in there:

and fix the permissions (`storage` and `bootstrap/cache` have to be writeable for the web-server)

```
sudo chgrp -R www-data mysite
sudo chmod -R ug+rwx mysite/storage mysite/bootstrap/cache
```

now adjust the `DocumentRoot` in your apache configs to point to `/var/www/mysite/public` systemctl reload apache2

Basics

Laravel follows the Model View Controller design principle.. where the View is basically your GUI, the Webpage in this case, the Model handles storing and reading your data.. so that's your database and some PHP scripts to access the database and retrieve Entries from it, and the Controller is the glue between the two, that's where your actual program logic happens.. the controller does something with the data and passes the result to the View.

artisan is a helper utility, it is a php cli script which is stored in the root of your Laravel project and on linux systems it is executable. so artisan can be called simply by typing `./artisan` in the root directory of the Laravel project. on ther systems one may need to run `php artisan`.

Blades (stored in `resources/views/*.blade.php` by default) are templates for generating the Webpage, or the View. They have their own templating language. The simplest example is

```
{{ $var }}
```

which in PHP would be something like `<?php echo htmlspecialchars($var); ?>` so it html encodes the contents of `$var` and prints it in-place where this expression is written. Blades also have some control structures and much more.

Public folder `public/` is the directory that is set up as your `DocumetnRoot` so that's the only directory that is visible to the outside. This means, that's where all your images and custom css go!

Routes (stored in `routes/*.php` by default) are needed to assign a URL to a View. look at the `web.php` file which contains an example route for a `get` request to `/` and forwards it to the `weclome` blade. the `get` from `Route::get()` actually means that it should handle a GET request.. so that's not a getter function for any type of route!. routes can also contain parameters which are passed as

http parameters (like in `index.php?parameter=abc`) using the function `$myvar = request('parameter')` or parameters can be passed as positional arguments in urls like `/users/1234/comments/233` where 1234 is the user-id and 233 is a comment id. this would be parsed using `Route:get('/users/{uid}/comments/{cid}', function ( $myId,$commentId )….` which passes whatever is behind `users/` in your url as the first argument and whatever is behind `comments/` as your second argument to your controller function. So the order of '{uid}' and '{cid}' is also the order in which the arguments are passed on to the controller function. The names of these placehoders don't matter for this purpose. they only come into play if you want to use some more advanced things like add a condition to the route using `Route::get(..)→where('name','[A-Za-z]+');` for example to filter out invalid URL's even before calling the controller.

Layouts (stored in `resources/views/layouts/*.blade.php` by best practice) basically a blade that contains common elements of multiple other blades is called a layout, like the http header and footer part.. a layout isn't simply included into the other blades.. instead, a blade for a view contains a `@extends(layouts.mylayout)` at the beining which would load the file `resources/views/layouts/mylayout.blade.php`. also the main view template blade now needs to contain sections. a section named “content” would start with `@section('content')` and end with `@endsection`. Now `mylayout` file can insert such a section at a given place by using the `@yield('content')` directive. so it's a little more complicated than a simple include, but also a lot more flexible as a result.

Controllers (stored in `app/Http/Controllers/*.php` by default) are, well, the controllers :) they are usually called by the route script and return an array of data to the blade that is called by the function. A new controller is created using `artisan make:controller MyController` (it is a naming convention, that the controller name should start with a captial letter). To call a function of the Controller, the route file `web.php` for example contains a line like `Route::get('/users/{id}', 'UserController@show');` which calles the show function in user controller and passes a argument `$id=1234` in the above example. the show function in the UserController typically then returns a view just like the `Route::get()` example in the `web.php file`.

database connections are configured in `.env` in your project root directory. by default laravel is configured to use mysql on localhost. so just create a new database and a user for it and adjust the config in `.env`

migrations (stored in `database/migrations/*.php`) define database tables that laravel will create for us. so instead of manually adding new database tables to the database or adding fields to a table, we create or adjust migration php scripts. they have a `up()` and `down()` method. the `up()` method contains the definition of what is to be created when the migration is made, the `donw()` method contains the reverse action, so usually a drop command of some sorts. migrations are created by using `artisan make:migration create_mytable_table` where `mytable` is the name of my new table. this will crate a template migration. to then execute the migration and actually create the database tables you have to run `artisan migrate`. to revert migrations, there are several options which can be seen by running `artisan migrate:` which lists all migrate commands. the most important during development is probably `artisan migrate:fresh` however, whenever a migration is rolled back, it essentially deletes the table, so that also deletes all the data in it. if you don't want to delete the data in the table and simply want to add one or more fields, you don't change the existing migration, instead you create a new migration for example `artisan make:migration add_myfield_to_mytable_table` the `add .. to .. table` portion is important.. artisan will understand tha and create a template for you which does exactly that, so you simply have to edit that

file and add your field definition(s), then run `artisan migrate` and your field is added.

eloquent models (saved in `app/Providers/*.php` by default) contain the definition of the Model in our MVC. An eloquent model extends a class which provides functions like `get()`, `find()` and such to get and manipulate database data. so through these eloquent models you access the database tables. new models can be created using artisan: `artisan make:MyModel` where `MyModel` is the name of the new eloquent model class to be created to access a certain table. The standard naming convention is that a table should be named in the plural of whatever it contains, i.e. “users” and the model should then be named in the singular form with a capital letter, i.e. “User”. Laravel will by default take the Model's name and pluralize it (in english) and convert it to lower case to set the database name. this may not always lead to very readable code. therefore the table name can be overwritten by defining the protected variable `protected $table='myTable;'` in the Model class definition for example.
To use a model in a controller we have to add `use App\MyModel;` at the beginning of our controller. we can then use `$somevar=MyModel::all()` to get all entries from our table for example. `$somevar` will be an array of objects in this case. To search for an entry we can use i.e. `MyModel::where('field','value')→get()` which will return all entries where the field `field` has the value `value`.. notice how we needed to call the `get()` method of the returned object to get our array of objects of entries.

processing post requests first of all we need a form to post some data. we can use a normal html form, but we need to add a directive `@csrf` in the blade just after the form tag. this is to prevent cross site request forging. laravel will automatically take care of providing a token to verify a form from our webpage was used from a valid session, so nobody can to post some date by hijaking a user session of an authenticated user for example. To process the POST request we now need the `post()` method rather than the `get()` method in our route and the controller method that is being called by the route is usually called `store()`. The Controller does not return a view, insead it returns a redirect to a view. this is probably to prevent reload issues. we can pass arguments to a redirect by using session variables. they can be added by using somethig like `return redirect('<url>')→with('sessvar','value')` where `sessvar` is the name of a session variable to be used to pass `value` to the blade. in the blade, use

```
{{ session('sessvar') }}
```

to output our value.
 to access the form data in the controller use the `request('<fieldname>')` function. then just create a new Model object like `$pizza=new Pizza()` and add properties to it like you would to any object: `$pizza→name = request('name');` for example. in the end use the `$pizza→save()` function to save the new entry.

processing delete requests are like posts, but we can't set the form method to DELETE so we need to use a blade function `@method('DELETE')` just after the form tag.

Naming Conventions

I'm going to use the examples of the above mentioned YouTube Series here, which is all about a pizza place :)

- Model Name: Pizza

- Table Name: pizzas
- Controller Name: PizzaController
  - to return the main `/pizzas` view: `index` called from controller@method `PizzaController@index` in the route
  - to return a single pizza entry from a url like `/pizzas/{id}`: view `show` called from controller@method `PizzaController@show` in the route
  - to create a new entry into the pizzas table: view `create` called from controller@method `PizzaController@create` in the route
  - to process a POST request: **no view** (instead use a `return redirect('<url>');` in the controller) in controller@method `PizzaController@store` in the route
    - to show a success message, we can either create a separate view, or, more elegantly, pass an argument to our show or index view and redirect to that view.
  - to process a DELETE request: **no view** in controller@method `PizzaController@destroy` in the route
- The Blades for a given folder in the url should be saved to a sub-folder with the same name
  - the “index” view, i.e. returning all pizzas should therefore be saved to `resources/views/pizzas/index.blade.php` and a view that shows a single pizza should be saved to `resources/views/pizzas/show.blade.php` as data will be provided through the controllers `show()` function.
  - in the `return view()` statement in the controller's function we would then pass `pizzas.index` as an argument. note how the `/` was replaced by a `.` .. this also means, that the sub directories should not contain `.` characters!

Tricks

Create Model + Controller + Migration all in one step

```
artisan make:model Pizza -mc
```

this creates the model, **m**igration and **c**ontroller all at once and puts names according to the above conventions in place.

From:
http://wiki.psuter.ch/ - **pswiki**

Permanent link:
http://wiki.psuter.ch/doku.php?id=getting_started_with_laravel&rev=1605714118

Last update: **18.11.2020 16:41**