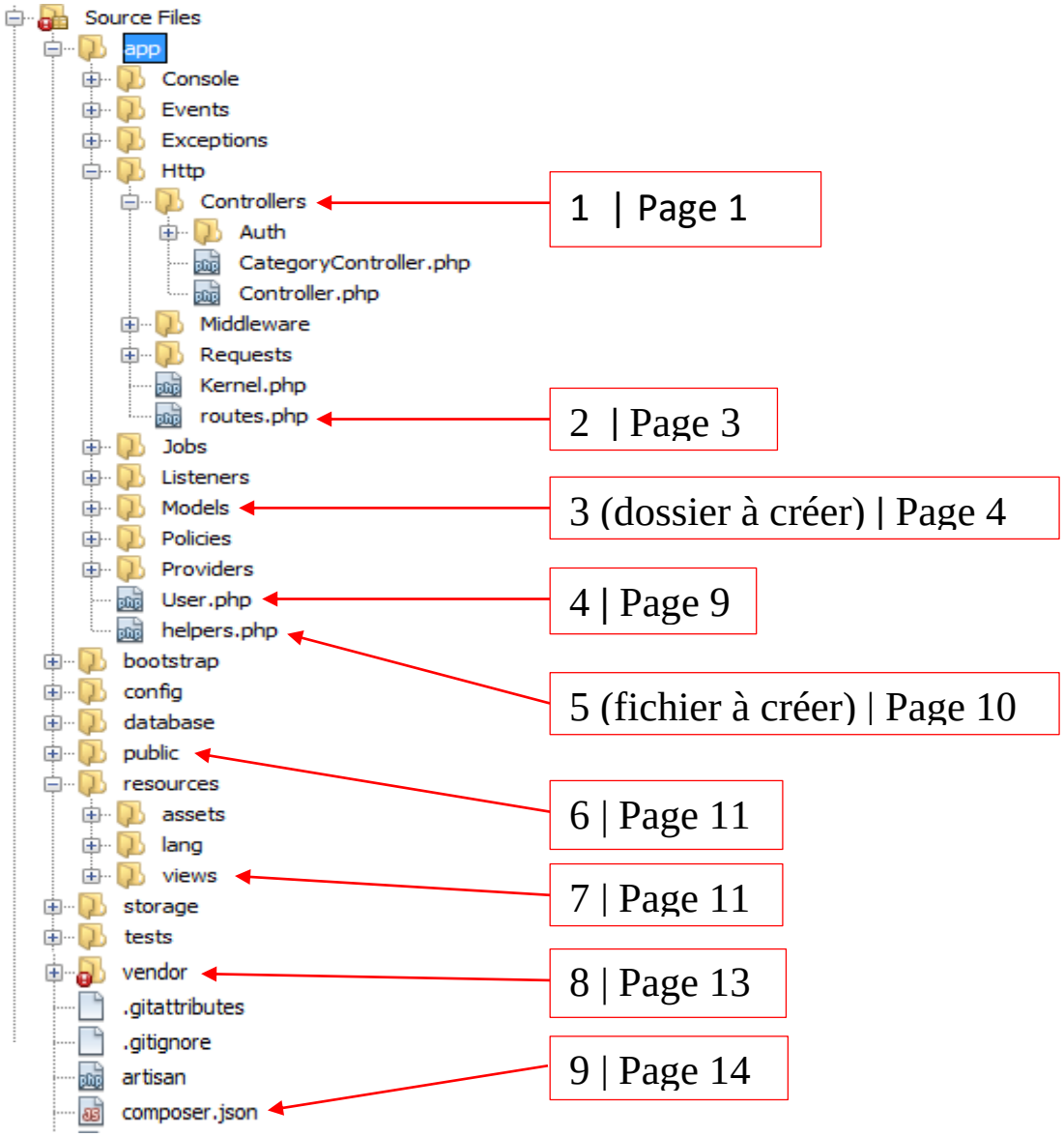
Structure d'un projet Laravel : https://laravel.com/docs/5.2



NB : les éléments numérotés sont ceux que l'on utilisera essentiellement au cours de la réalisation de notre projet.

1) **Le dossier Controllers** https://laravel.com/docs/5.2/controllers

Contient tous les fichiers contrôleurs qui seront appelés à chaque fois qu'une requête http sera réalisée (d'où le nom du dossier http dans lequel il est enregistré).

En règle générale, éviter de mettre trop de méthode dans un contrôleur. Un contrôleur devrait contenir au maximum 10 méthodes elles-mêmes constituées de 20 lignes de code maximum.

Prenons exemple du contrôleur AdvertController.php (relatif aux annonces publiées par les membres) qui contiendrait les méthodes suivantes :

Méthode	Rôle	Url
index	Lister toutes les annonces.	/adverts
create	Afficher le formulaire de création d'une annonce.	/advert/create
store	Sauvegarder une annonce en base de données uniquement lorsque le formulaire est envoyé.	/advert/create
show	Afficher une annonce en fonction de son id	/advert/{id}
edit	Afficher le formulaire d'édition d'une annonce	/advert/{id}/edit
update	Met à jour une annonce après envois du formulaire	/advert/{id}/edit
destroy	Supprime une annonce en fonction de son id	/advert/{id}/delete

NB : Les méthodes de ce contrôleur pourront être appelées via le fichier app/http/ route.php (**cf. partie 2 page suivante**)

Toutes les classes définies dans le dossier Controllers devront avoir le namespace App\Http\Controllers ; et hériter de la classe BaseController défini dans le namespace Illuminate\Routing\.

Astuce : Dans une méthode d'un contrôleur il est possible de définir des arguments à la volée qui seront automatiquement « injectés » par Laravel et disponibles dans la méthode (https://laravel.com/docs/5.2/controllers#dependency-injection-and-controllers).

Par exemple si vous faites à l'intérieur d'une classe d'un contrôleur:

```
1. <?php namespace App\Http\Controllers;
2.
3. use Illuminate\Http\Request;
4. use Illuminate\Routing\Controller as BaseController;
5.
6. class AdvertController extends BaseController
7. {
8.     public function store(Request $request)
9.     {
10.             $name = $request->name;
11.             // renvoi le contenu de $_GET['name'] ou $_POST['name']
12.         }
13.     }
14.
    }
```

Alors la variable $request deviendra automatiquement une instance de la classe Request et permettra entre autre de récupérer le contenu des variables de formulaire ou d'url (alternative plus sécurisée aux $_GET et $_POST).

En savoir plus sur l'objet Request : https://laravel.com/docs/5.2/requests

2) Le fichier routes.php https://laravel.com/docs/5.2/routing

Contient toutes les routes permettant d'appeler un contrôleur et une de ses méthodes selon la requête http effectuée (url tapée dans une barre d'adresse).

Exemple dans le cas du contrôleur UserController :

Si je tape http://localhost/monprojet/public/index.php/user/5 , j'aimerai que la méthode show de la classe UserController soit appelée.

Pour se faire il faut définir la route spécifique dans le fichier route.php :

```
1. Route::group(['middleware' => ['web']], function () {
2.
3.     Route::get('/user/{user}', 'UserController@show')->name('user.show');
       // La méthode name contient le nom de la route
4.
5.     // Toutes les autres routes seront définies ici les unes en dessous des
   autres.
6.
7. }
```

Le premier paramètre de la méthode statique get ('/user/{user}') appliqué sur la classe Route (Route ::get(route,callback)) est l'url. Ici {user} devra être remplacé dans l'url tapée dans la barre d'adresse par l'id d'un utilisateur existant dans la table users (localhost/monsite/public/index.php/user/5).

Le second paramètre ('UserController@show') correspond au nom du contrôleur à appeler (en rouge) ainsi que sa méthode (en vert).

Par exemple dans notre contrôleur UserController on aura :

```
1. <?php namespace App\Http\Controllers ;
2.
3. use App\User ;  // On charge la classe User stockée à la racine de
   dossier app/
4. use Illuminate\Routing\Controller as BaseController;
5.
6. class UserController extends BaseController {
7.
8.         public function show(User $user){
9.
10.             $data['user'] = $user ;
11.
12.            // Si on tape localhost/monprojet/public/index.php/user/5
   alors $user sera une instance de la classe User en rapport avec
   l'user ayant l'id 5 dans la table users. Tout ceci est fait
   automatiquement ;)
13.             return view('user/show',$data);  // affichera le
   template user.blade.php dans le dossier ressources/views/user/
14.
15.        }

    }
```

2.1) Méthodes de la classe Route que nous utiliserons le plus:

Route ::get(url,callback) >>> se déclenchera dans le cas où une page est chargée normalement (sans envoi de formulaire).

Route ::post(url,callback) >>> se déclenchera dans le cas où une page est chargée via un envoi de formulaire.

NB : Dans ce cas précis il faudra systématiquement renseigner un champs de formulaire caché contenant le jeton de session csrf_token :

```
<input type="hidden" name="_token" value="{{ csrf_token() }}">
```

Route ::match(url,callback) >>> se déclenchera qu'importe si la page est chargée avec ou sans envoi de formulaire.

2.1) Les Middlewares (https://laravel.com/docs/5.2/routing#route-group-middleware)

Il est possible de grouper les routes définies dans un Middleware qui se chargera d'appeler des classes définies dans le dossiers app/Http/Middleware/ avant l'exécution de la méthode du contrôleur définit dans la route.

Le middleware `web` définit dans notre exemple s'occupera par exemple de charger une série de classe qui vérifieront à notre place que le jeton de session csrf_token d'un formulaire est bien valide.

Dans le cadre de ce TP nous n'utiliserons d'ailleurs uniquement le groupe de middleware « web » et « auth ». Il faudra donc veiller à ce que les routes dans le fichier routes.php soient bien spécifiées à l'intérieur de la fonction anonyme définit en second paramètre de la méthode statique groupe :

```
8. Route::group(['middleware' => ['web']], function () {

9.     // Définir les routes ici
   }
```

```
10.      Route::group(['middleware' => ['auth']], function () {

11.          // Définir ici les routes dont l'accès ne sera autorisé que
   si l'utilisateur est connecté.
12.      }
```

3) Le dossier Models

https://laravel.com/docs/5.2/eloquent

C'est le dossier qui contiendra toutes nos classes représentant chacune des tables de notre base de données. Par exemple la classe Advert représentera la table adverts (annonces), la classe Category la table categories etc. Seule la classe User (table users) sera enregistrée à la racine du dossier app et fera l'objet d'un chapitre à part.

Ce dossier n'existe pas par défaut et il faudra donc le créer.

Toutes les classes définies dans le dossier Models devront avoir le namespace App\Models et hériter de la classe Model définie dans le namespace Illuminate\Database\Eloquent\ :

```php
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Advert extends Model
{
    //
}
```

3.1) Enregistrer de nouvelles informations en base de données :

La méthode save de la classe Model de laravel permet d'enregister une nouvelle entrée en base de données où de l'éditer si elle existe déjà :

```php
<?php namespace App\Http\Controller;

use App\Models\Category;
use Illuminate\Routing\Controller as BaseController;

class CategoryController extends BaseController {


    public function store(){
       // Méthode 1 avec la méthode save :
       $category = new Category;
       $category->name = "Webdesign";
       $category->save();  // Enregistrera une nouvel entrée dans la table categories. Si il y a un champs created_at alors sa valeur sera automatiquement définit comme étant la date d'auourd'hui.

             // Méthode 2 avec la méthode static create :
        $category = Category::create(['name' => 'Websdeign']) ;

     }
         // On édite une catégorie via l'url localhost/monsite/web/index.php/category/7/edit
     public function update(Category $category){

        $category->title = "Developpeur Web";
        $category->save();  // Mettra à jour l'entrée de la table categories correspondant à l'id 7. Si il y a un champs updated_at alors sa valeur sera automatiquement définit comme étant la date d'auourd'hui.

     }
     }
```

3.2) Récupérer des infos en base de données restituées sous forme d'objets :

La classe Model nous permettra de nous faciliter grandement la vie en ce qui concerne la réalisation des requête SQL en utilisant simplement des méthodes telles que :

```
1. $advert = Advert::find(4) ;   // Renverra une instance de la classe
   Advert avec une propriété attributes contenant un tableau associatif
   avec les infos de l'annonce de la table adverts ayant l'id 4.
2. echo $advert->title ; // Affichera la valeur de l'attribut title.
3. $adverts = Advert::all() ; // Renverra toutes les annonces de la
   tables adverts
4. $adverts = Advert::where('hourlyWage' ,'>' , 40)->get() ; // Renverra
   les annonce dont la valeur du champs hourlyWage (taux horaire) est
   inférieur à 40 (40€ / h).
```

La méthode get finale permet d'obtenir les résultats de notre requête sous forme d'une collection d'objet Laravel (une sorte de tableau amélioré contenant plusieurs instances de la classe Advert) : https://laravel.com/docs/5.2/eloquent-collections

Astuce : Il est possible de chainer les méthodes de nos requêtes :

```
1. $adverts = Advert::where('hourlyWage','>', 40)
2.                         ->where('title', 'LIKE' , '%webdesign%')
3.                         ->where('user_id', 3)
4.                         ->get() ;
```

Dans l'exemple ci-dessus on récupère toutes les annonces dont le taux horaire et inférieur à 40€ / hr, dont le titre contient le terme « webdesign » et dont l'auteur est l'utilisateur ayant l'id 3.

Autre syntaxe possible (utile lorsque l'on développe un moteur de recherche):

```
5. $query = Advert::where('hourlyWage','>', 40) ;
6. $query->where('title', 'LIKE' , '%webdesign%');
7. $query->where('user_id', 3) ;
8. $adverts =  $query->get() ;
```

L'essentiel des requêtes sont listées sur la documentation suivante :
https://laravel.com/docs/5.2/eloquent

Astuce : Si vous souhaitez générer un système de pagination sans faire d'effort procéder comme suit :

```
1. $adverts = Advert::where('hourlyWage','>',40)->orderBy('created_at','DESC')-
   >paginate(20);
2. /* Les résultats renvoyés dépendront du paramètre d'url "page"
3. Exemple: si on tape localhost/monsite/public/index.php/adverts?page=3
4.
5. alors $adverts contiendra les 20 annonces ordonnées par date de
   création décroissante à partir de l'index 40  (LIMIT 40,20).
```

En savoir plus sur https://laravel.com/docs/5.2/pagination

3.2.1) Au sujet des dates (objet Carbon) http://carbon.nesbot.com/docs/

Par défaut laravel renvoit les valeurs des champs created_at et updated_at sous forme d'instance de la classe Carbon.

Cette classe n'est ni plus ni moins qu'une amélioration de la classe native PHP DateTime (dont elle hérite). Elle offre des méthodes intéressantes telles que diffForhumans qui renverra la date sous forme de temps écoulé en fonction de la date enregistrée :

```
1. $advert = Advert::find(5);
2.
3. echo 'Annonce crée '. $advert->created_at->diffForHumans();
4. /*
5. Si cette annonce a été crée il y a 5minutes alors cette méthode
   renverra "il y a 5 minutes"
6.
7. Si cette annonce a été crée il y a 3 heures alors cette méthode
   renverra "il y a 3 heures" */
```

3.2.2) Modifier en amont la valeur d'un champ d'une table avant son affichage.

https://laravel.com/docs/5.2/eloquent-mutators

En TP nous avions vu les méthodes magiques __get et __set qui étaient exécutées lorsque l'on tentait d'accéder à une propriété inexistante ou définit comme étant protected ou private. Laravel implémente déjà ces méthodes-là dans sa classe Model (dont héritent toutes les classes du dossier app/Models). Cependant leur fonctionnement est quelque peu différent.

Ce qu'il faut savoir c'est que lorsque l'on fait par exemple :

```
1. <?php $advert = $advert ::find(5) ;
2.       echo $advert->title ;
3. ?>
```

la propriété title n'existe pas dans la classe Advert. La valeur du champ title renvoyée est en fait récupérée à l'intérieur d'une propriété « attributes » contenu dans la classe Model (dont hérite la classe Advert). Cette propriété contient en fait un tableau associatif contenant les valeurs brutes d'une entrée récupérées en base de données (identique à ce que l'on récupère habituellement via la méthode fetch(PDO ::FETCH_ASSOC)) :

```
1. ['id' => 5,
2.  'title => 'Cherche développeur Web Laravel',
3.  'content' => ' Lorem ipsum dolor … ',
4.  'category_id' => 7 ,
    'user_id' => 3 ,
5.  'created_at' => '2016-12-25 14 :25 :33',
6.  'updated_at' => '2016-12-27 17 :38 :45']
```

Comme expliquée dans la section précédente **3.2.1** , les valeurs de propriétés « created_at » et « updated_at » seront renvoyées sous forme d'instance de la classe Carbon. Cependant les autres valeurs (id,title, content,category_id, user_id) seront renvoyées telles quelles, sauf si vous définissez une méthode « spéciale » au sein de la classe Advert dont la synthaxe est la suivante :

```
public function get{ Nom de la propriété }Attribute($value){

   //

}
```

Exemple concret ci-dessous :

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

use Illuminate\Database\Eloquent\Model;

class Advert extends Model
{
    public function getTitleAttribute($value)
    {
        return ucfirst($value);
    }
}

$advert = Advert::find(5);

echo $advert->title;  // Renverra le titre avec la première lettre en majuscule
```

Dans l'exemple ci-dessus lorsque l'on cherchera à afficher $advert->title, laravel vérifiera si la méthode getTitleAttribute existe dans la class Advert. Si c'est le cas elle sera appelée et son paramètre $value correspondra à la valeur brute du champ title.

En savoir plus https://laravel.com/docs/5.2/eloquent-mutators

3.2.3) Etablir des relations entre nos différents models (Advert , Category, User etc.)

https://laravel.com/docs/5.2/eloquent-relationships

Admettons que l'on souhaite récupérer une annonce et sa catégorie dans la foulée.

Avant il nous fallait faire :

```php
$query = $pdo->query('SELECT adverts.id, adverts.title,
adverts.content, categories.name as name
FROM adverts
INNER JOIN categories ON categories.id = adverts.category_id
WHERE id = 5');

$query->execute();
```

```php
$advert = $query->fetch(PDO::FETCH_ASSOC);

echo $advert['name'];  // Affiche le nom de la catégorie
```

Avec Laravel la tâche s'en trouve simplifiée :

Structure de la class Advert :

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

use Illuminate\Database\Eloquent\Model;

class Advert extends Model
{
    /**
     * Get the category record associated with the advert.
     */
     public function category()
     {
         return $this->belongsTo('App\Models\Category');
     }
}
```

Code permettant de récupérer le nom d'une catégorie associée à une annonce :

```php
$advert = Advert::where('id',5)->with('category')->first();

// La méthode with permet de faire une jointure de table

// La méthode first permet de ne récupérer qu'une seule entrée. Si on avait plusieurs entrées à récupérer alors on aurait alors utilisé la méthode get.

echo $advert->category->name; // affiche le nom de la catégorie associée à l'annonce récupérée (basée sur la clé étrangère category_id de la table adverts).
```

Tous les autres types de relation sont détaillés sur la documentation officielle :
https://laravel.com/docs/5.2/eloquent-relationships

4) La classe User et Auth
https://laravel.com/docs/5.2/authentication

La classe User enregistrée à la racine du dossier app est fournie par laravel et est en lien avec la table users. Celle-ci est étroitement liée à une autre classe : Auth.

A chaque fois que l'on s'authentifiera sur le site avec notre email et mot de passe, une instance de la classe User correspondant à ces identifiants sera automatiquement créé et intégrer à la classe Auth.

Exemple dans le contrôleur AuthController aussi fourni par laravel (dossier app/http/Controller/Auth) :

```
1.  <?php
2.
3.  namespace App\Http\Controllers;
4.
5.  use Auth;
6.
7.  class AuthController extends Controller
8.  {
9.      /**
10.      * Handle an authentication attempt.
11.      *
12.      * @return Response
13.      */
14.     public function authenticate()
15.     {
16.         if (Auth::attempt(['email' => $_POST['email'], 'password' => $_POST[
    'passwprd']))
                {
17.                 // Autorisation acceptée
18.                 $userConnected = Auth::User();
19.             }
20.     }


    }
```

La méthode attempt prend en paramètre un tableau associatif comprenant 2 propriétés email et password dont les valeurs correspondent aux champs du formulaire de connexion.

Si le mot de passe et email concordent, alors à tout moment dans notre code nous pourrons accéder aux informations de l'utilisateur connecté via la méthode statique : Auth ::User(). Il faudra juste bien veiller à charger la classe Auth comme en ligne 5 de l'exemple ci-dessus.

En savoir plus sur https://laravel.com/docs/5.2/authentication

5) Le fichier helpers.php

Ce fichier n'est pas fourni de base par Laravel. Il nous faudra le créer et dans celui-ci nous définirons nos propres fonctions « faites maison » et qui seront disponibles n'importe où dans notre code. Toutefois il faudra que ce fichier renseigné dans le fichier composer.json. Pour se faire il faudra ajouter une propriété « files » contenant un tableau de fichiers à charger :

```
"autoload": {
        "files": [
            "app/helpers.php"
        ],
...
}
```

Et ensuite il faudra lancer la commande « composer dump » dans le terminal pour que nos fonctions soient disponibles.

Il faut aussi savoir que Laravel met à notre disposition énormément de « helpers » prêt à l'emploi : https://laravel.com/docs/5.2/helpers

6) Le dossier public

Ce dossier a vocation à contenir tous les fichiers js css et images. Pour une organisation optimale il est recommandé de créer 3 dossiers js, css et images dans celui-ci. Il contient également le fichier index.php (communément appelé dispatcheur) vers lequel toutes les requêtes http pointeront.

7) Le dossier views https://laravel.com/docs/5.2/views

C'est le dossier qui contiendra les fichiers contenant notre code HTML, soit l'équivalent du phtml (en mieux). L'extension des fichiers contenus dans ce dossier est .blade.php indiquant que l'on utilise le moteur de template blade : https://laravel.com/docs/5.2/blade

7.1) Afficher un template depuis un contrôleur

Pour afficher un template depuis un contrôleur, il suffit juste d'utiliser la fonction (helper) view à l'intérieur d'une méthode:

```
1. < ?php namespace App\Http\Controllers ;

2. use Illuminate\Routing\Controller as BaseController;
3. use App\User ;

4. class UserController extends BaseController {

        public function profil(User $user){

            $data['userData'] = $user, ;
            $data['title'] = 'Profil de '.$user->fullName() ;

            return view('user.profile', $data);

        }
5.

   }
```

Le premier paramètre de la fonction view est le template à afficher. Ici on cible le fichier profile.blade.php stocké dans le dossier user du dossier views.

Le second paramètre est un tableau associatif contenant les infos à transmettre à notre template. Au sein du template ces données seront accessibles sous forme de variables dont le nom correspond aux index de ce tableau associatif.

Dans notre exemple nous auront donc une variable $title = « Mon profil » et $userData = $user.

7.2) Le moteur de template blade https://laravel.com/docs/5.2/blade

7.2.1 : l'héritage de template

Contrairement à ce qu'on avait l'habitude de faire avec le phtml, le template qui est affiché en premier est le template « enfant » (équivalent du register.phtml qui était inclus dans le template parent layout.phtml).

Généralement le template « parent » se nomme layout.blade.php et est enregistré à la racine du dossier views. Ce template contient l'ensemble du code commun à toutes les pages du site (balise body, fichiers css, js etc.) comme nous l'avions fait pour le fichier layout.phtml.

Quant aux templates « enfants » ils sont enregistrés dans des sous-dossiers dans le dossier views (views/user/profile.blade.php) et héritent du template parent :

Exemple avec le template enfant « profile.blade.php » :

```
@extends('layout')  < !—layout correspond au fichier layout.blade.php -->

@section('content') < !—Tout le code compris entre ces 2 instructions en rouge sera inclus au niveau de l'instruction @yield('content') du fichier layout.blade.php -->

<h1>Mon profil</h1>
- Pseudo: {{ $userData->name }}
- Email : {{ $userData->email }}
@endsection
```

Template parent layout.blade.php :

```
<!DOCTYPE html>

<html lang="fr">

        <head>

                <meta charset="utf-8">

                <title>{{ $title }}</title>

        </head>

        <body>

        <header>Super Site</header>

        <div id= ''main''>
                @yield('content')

        </div>

        </body>

</html>
```

Lors de l'appel du template profile.blade.php via un contrôleur :

```
return view('user.profile', $data);
```

celui-ci sera inclus dans le template parent layout.blade.php exactement là où sera défini l'instruction **@yield('content').**

7.2.2 Affichage des variables dans blade

Dans les templates blade, les variables sont affichées via la notation {{ $maVariable }} et sont automatiquement encodées en caractère html :

{{ $title }} équivaut à faire < ?php echo htmlentities($title) ; ?>

{ !! $title !!} équivaut à faire < ?php echo $title ; ?> (pas de protection).

7.2.3 Chargement des fichiers js et css

Pour charger dans un template un fichier js ou css exeterne utiliser la fonction asset :

<link href="{{ asset('css/app.css') }}" rel="stylesheet">

NB : le dossier css est enregistré dans le dossier public/

7.2.4 Inclusion de fichiers dans blade

Vous pouvez inclure des fichiers .blade.php dans vos templates via l'instruction @include :

@include('includes.menu') aura pour effet d'inclure le fichier menu.blade.php stocké dans le dossier ressources/views/includes/

En savoir plus sur blade et toutes ses fonctionnalités : https://laravel.com/docs/5.2/blade

8) Le dossier vendor

Ce dossier contient toutes les librairies php (dépendances) dont notre application laravel a besoin pour fonctionner.

Pour mettre à jour ces dépendances il suffit de lancer la commande composer update.

Pour rajouter une nouvelle dépendance (chose que nous avons fait avec l'ajout de la debugbar), il faut se rendre sur https://packagist.org/ , trouver une librairie qui nous intéresse et suivre les instruction.

Par exemple si nous souhaitons installer une librairie permettant de chercher des utilisateurs sur un rayon de X km en fonction de notre position il suffit de se rendre sur https://packagist.org/packages/jackpopp/geodistance , puis dans composer.json ajouter la propriété :

"jackpopp/geodistance": "dev-master" dans l'objet littéral de la propriété « require » :

```
{
    "require": {
                "jackpopp/geodistance": "dev-master",
                 ….
                }
  …
}
```

Et enfin lancer ensuite la commande composer update dans le terminal.

9) Le fichier composer.json

C'est en quelque sorte le chef d'orchestre de notre application dont voici la structure :

```
{
  "name": "laravel/laravel",
  "description": "The Laravel Framework.",
  "keywords": ["framework", "laravel"],
  "license": "MIT",
  "type": "project",
  "require": {
    "php": ">=5.5.9",
    "laravel/framework": "5.2.*",
                "laravelcollective/html": "5.2.*",
  },
  "require-dev": {
    "fzaninotto/faker": "~1.4",
    "mockery/mockery": "0.9.*",
    "phpunit/phpunit": "~4.0",
    "symfony/css-selector": "2.8.*|3.0.*",
    "symfony/dom-crawler": "2.8.*|3.0.*",
    "barryvdh/laravel-debugbar": "^2.3"
  },
  "autoload": {

                "files": [
                        "app/helpers.php"
                ],
    "classmap": [
      "database"
    ],
    "psr-4": {
      "App\\": "app/"
    }
  },
  …
}
```

Composer update :

Dépendances qui seront active en local et en production (c'est-à-dire une fois le site mis en ligne).

Dépendances qui seront active uniquement en local.

Composer dump :

Fichier à charger au tout début. Le fichier helpers.php contient pour rappel nos fonctions « faites maison ».

Comme vue sur le TP Silex, nous avons ici tous les préfixes de namespace ciblant des dossiers bien définit.

Par défaut le préfixe de namespace App\ définit par Laravel cible le dossier app/