GALIMATECH
L'EXPERIENCE DIGITALE

Le framework Laravel

***

Laravel est en ce moment le framework le plus populaire sur github. Il est facile à appréhender et est très puissant. Il offre la possibilité de concevoir tout type de projet Web aussi bien en front-end qu'en back-end.

Installation:Composer

Pour installer et configurer Laravel nous allons avoir besoin de deux gestionnaires de paquets: composer pour Php et npm pour Node js.
Pour installer composer sur Ubuntu il est conseillé de passer par curl. Nous allons donc installer curl et les modules Php nécessaire.

```
sudo apt install curl php-cli php-mbstring git unzip
```

Puis nous allons télécharger composer en utilisant curl.

```
curl -sS https://getcomposer.org/installer -o composer-setup.php
```

Puis nous devons récupérer la dernière clé SHA-384 liée au fichier composé téléchargé à cette adresse:`https://composer.github.io/pubkeys.html.`
Nous pouvons ensuite vérifier l'état de notre installation avec le code suivant:

```
php -r "if (hash_file('SHA384', 'composer-setup.php') === '$HASH') { echo 'Installation reussie'; }
else { echo '???Fichier corrompu???'; unlink('composer-setup.php'); } echo PHP_EOL;
```

Si le message "installation réussie" est affiché, nous pouvons alors lancer la commande pour installer composer.

```
sudo php composer-setup.php --install-dir=/usr/local/bin --filename=composer
```

Installation:npm

Nous devons aussi disposer du gestionnaire de paquets npm. Il n'est pas nécessaire pour l'installation de Laravel, mais nous allons en avoir besoin sur certaines parties de la configuration.
Pour installer npm nous allons d'abord activer les référentiels NodeSource sur notre système avec la commande suivante:

```
curl -sL https://deb.nodesource.com/setup_10.x | sudo -E bash -
```

Nous pouvons maintenant installer nodejs qui installera du coup NPM.

```
sudo apt install nodejs
```

Nous pouvons mai

Installation:Laravel

Maintenant que nous avons les outils nécessaires nous pouvons installer laravel. Nous avons deux méthodes pour installer Laravel.
Nous pouvons télécharger l'installeur de laravel et utiliser cet installeur à chaque fois que nous devons créer un projet avec la commande laravel new nom_de_mon_projet.
Ou nous pouvons utiliser la commande create-project de composer:

```
composer create-project --prefer-dist laravel/laravel
nom_de_mon_projet
```

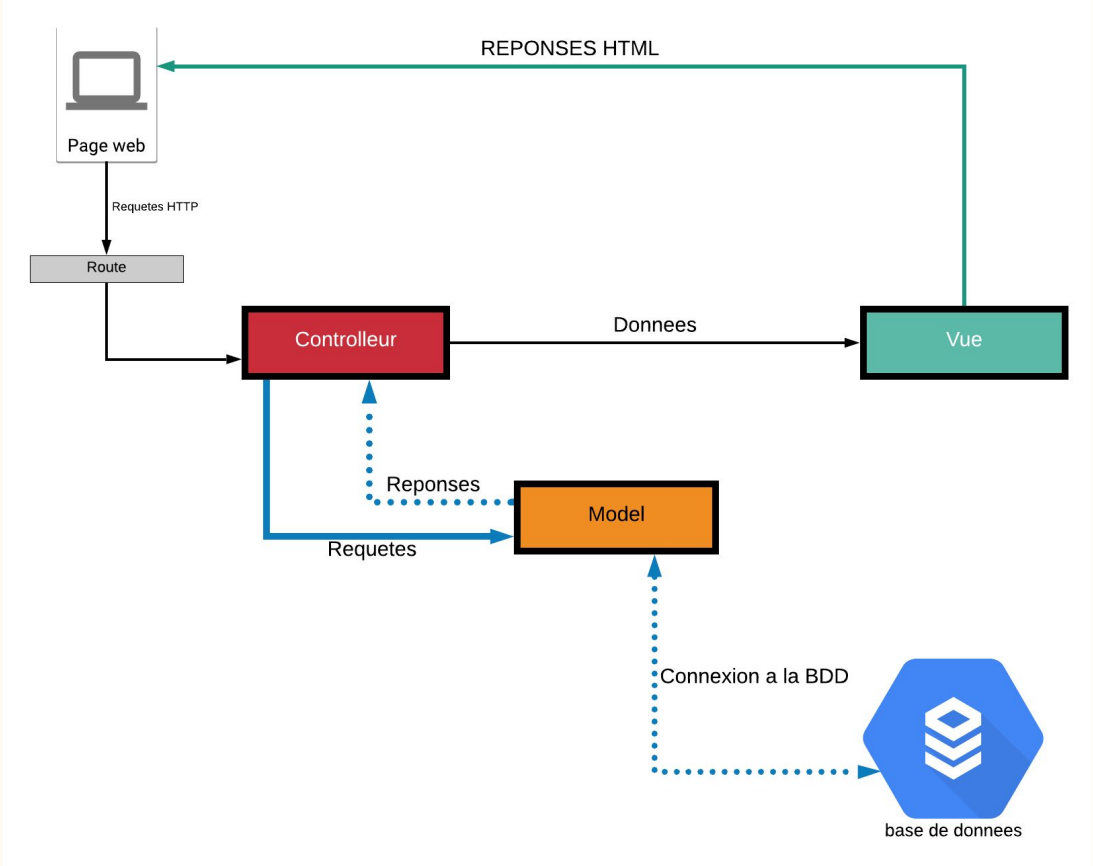
Le pattern MVC et Laravel

Le Patern MVC est une manière d'organiser les dossiers de notre projet en trois grands blocs que sont:

- Les contrôleurs: ils sont chargés de recevoir les requêtes et de les traiter. Une fois le traitement terminé ils vont envoyer les données aux vues
- Les vues: les vues sont chargées de communiquer avec le navigateur ils gèrent l'affichage de la page demandé. Il n'y a généralement pas de traitement dans les vues on ne fait qu'afficher.
- Les modèles: les modèles sont la phase intermédiaire entre les vues et les contrôleurs. Ils font souvent une liaison avec la base de données.

Le pattern MVC et Laravel

Le contrôleur va contacter le modèle que si la requête nécessite d'aller chercher des données sur la base de donnée. Le modèle gère donc la requête en base de données et retourne les données au contrôleur. Le contrôleur va alors traiter ces données avant de les envoyér à une vue pour l'affichage.



Les contrôleurs

Les fichiers des contrôleurs sont localisés dans app/http/controllers. Ce sont des fichiers dont le nom se termine toujours par "Controller'". Ils contiennent des classes qui héritent d'une classe Controller() et qui vont avoir des méthodes appelées actions. Ces méthodes vont le plus souvent correspondre aux pages de notre site web.

Les contrôleurs jouent un rôle intermédiaire entre les classes modèles et les vues. ils vont recevoir les données de l'URL, faire un traitement et envoyer les valeurs aux vues. Ils sont le point central de notre application.

Pour ajouter un controller nous pouvons créer un fichier dans app/Http/Controllers. Mais nous pouvons aussi utiliser php artisan. La commande pour créer un controller est alors php artisan make:controller NomController. Cette commande va créer un fichier avec le Nom défini et la classe du controller.

Les Vues

Les vues sont chargées de faire les retours HTML des pages. Les fichiers des vues sont localisés dans le dossier resources/views/. Ils ont toujours l'extension nomfichier.blade.php. La raison à cela est que Laravel utilise un moteur de template nommé Blade. Blade permet de mixer le HTML avec des scripts qui vont permettre d'afficher les données du contrôleur ou faire des opérations tels que des boucles.
Les variables sont affichées entre des doubles accolades `{{ }}`
ceci est l'équivalent de `<?php echo $var ?>`

Avec blade nous pouvons définir des layouts qui seront utilisés sur toutes les pages. Pour cela nous allons créer un fichier dans app.blade.php dans un dossier layouts dans les views. Ce fichier fait appel aux fichiers css et js necessaires et définit aussi un espace de rendu. Cet espace est définit avec le mot clé @yield suivit du nom de l'espaces
`ex: @yield('content')`

Les autres vues pourront alors utiliser ce layout en l'important avec. @extends('layouts.app'). Puis il vont mettre du contenu dans la zone yield définit dans le layout comme suite:

```
@section('content')
    ici je met le contenu de ma vue
@endsection
```

Les Vues:Compiler les composants front-end

Laravel s'adapte très bien avec les outils front-end telque Bootstrap, React et vue js. Il utilise la librairie laravel/ui pour mettre en place Bootstrap ou Vues.
Nous pouvons installer cette librairie avec composer:

```
composer require laravel/ui --dev
```

Une fois la librairie installée, nous pouvons importer Bootstrap avec la commande suivante:

```
php artisan ui bootstrap
```

Pour charger les fichiers Css et js, Laravel utilise un système de compilation. Cela consiste à regrouper tous les fichiers Css/js en un seul fichier et de n'importer que ce fichier au niveau du layout. Cette compilation se fait dans le fichier webpack.mix.js.
Dans ce fichier nous utiliserons différentes fonctions pour charger du Css ou du js.
Pour compiler du js on utilise la fonction js() cette fonction prend en paramètre un ou plusieurs fichiers js à compiler et l'emplacement du fichier compilé. Par défaut le Javascript compilé est enregistré dans le dossier public/js dans un fichier App .Js.

```
mix.js(["fichier1.js,fichier2.js"], "public/js").
```

De la même manière, nous pouvons compiler du css avec la fonction style(). Elle fonctionne comme la fonction js(). Le css compilé est enregistré dans le dossier public/css dans un fichier app.css:

```
mix.style(["fichier1.css,fichier2.css"], "public/css");
```

On peut donc se retrouver avec un fichier webpack.mix.js qui ressemble à cela:

```
mix.js(["resources/library/scripts.js", "resources/library/jscripts.js"], "public/js")
                .styles( [ "resources/styles/design.css", "resources/styles/style.css" ],
"public/css/all.css" );
```

Après avoir conçu notre fichier mix, nous pouvons utiliser npm pour compiler les fichiers.

```
npm run dev
```

Les fichiers routes

Les diffèrentes pages de l'application sont affichées à travers les urls. Le systèmee qui permet de lier une page à un controlleur s'appel le routing. Les fichiers chargés de gerer ce sytème sont dans le dossier 'routes' à la base du projet. Nous agirons principalement sur deux type de routes : le routing en mode api, qui est fait pour ceux qui veulent créer une API avec Laravel; il est géré par le fichier api.php. Et le routing pour les pages web qui est géré par le fichier web.php

Pour écrire nos routes nous allons faire des appels statiques d'une classe oute dans le fichier web Php.

```
Route::get('/', function(){
        echo "je suis sur le fichier routes/web.php";
    });
```

Dans l'exemple nous utilisons la méthode get qui correspond à un appel HTTP classique. Cette méthode va prendre deux paramètres:

- l'url de la page
- l'action à exécuté sur cette url

L'URL peut contenir des variables. Pour ajouter des variables on met le nom de la variable entre accolades.

```
Route::get('/show/{product_slug}',ProduitsController@show);
```

Cet exemple va générer l'URL suivant server/show/slug_product. Avec slug_product qui va être égal au slug d'un produit. Au niveau du contrôleur l'action show doit avoir un paramètre $product_slug:

```
public function show($product_slug){ ... }
```

Les migrations et les modèles

Laravel facilite énormément l'interaction avec la base de données. Actuellement il est adapté à quatre types de base de données que sont: MySql, Postgresql, SQLite, SQL Server.

Le fichier .env à la racine du projet contient les configurations par defaut. Dans ce fichier nous pouvons définir la connexion à notre base de données en remplissant les valeurs

```
DB_CONNECTION=mysql qui va representé le type de basse de données
DB_HOST=127.0.0.1 // l'adresse du serveur
DB_PORT=3306 // le port
DB_DATABASE=iotvine //le nom de la base de données
DB_USERNAME=root //L'utilisateur de la base
DB_PASSWORD=$mart_B4 // le mot de passe de l'utilisateur.
```

Les migrations et les modèles

Pour gérer les tables de notre base de données nous allons utiliser les modèles et les migrations Éloquent Éloquent est l'Orm intégré dans Laravel qui offre une manière simple de gérer les tables de base de données. Pour utiliser la puissance de cet ORM nous allons utiliser ses modèles et migrations et nous efforcer de respecter les conventions.
Pour créer un modèle Éloquent nous allons utiliser la commande Php artisan Make: modèle Nom Modèle. Les modèles sont écrits en anglais avec la première lettre en majuscules. Le modèle créé sera enregistré à la racine du dossier App. Les modèles sont des classes qui héritent de la classe Éloquent\Modèle. Les modèles sont souvent accompagnés d'un fichier migration qui va définir la table de la base de données qui doit être liée au modèle. Pour créer une migration en même temps que le modèle nous allons utiliser l'option -m avec la commande de création du modèle.

```
php artisan make:modèle Product -m.
```

Les fichiers de migration sont enregistrés dans le dossier database/migrations. Ils ont un nom qui commence par la date et l'heure de création. Ils contiennent une classe qui hérite de la classe migration. Une migration contient deux méthodes nommées up() et down(). La méthode up est exécutée pour créer la table ou apporter des modifications sur la table et la méthode down permet de défaire les actions de la méthode up().
Dans ces deux méthodes nous utilisons les méthodes static d'une classe Schema pour construire ou détruire une table.

```
public function up()
{
        Schema::create('products', function (Blueprint $table) {
                $table->bigIncrements('id');
                $table->string('name',"90");
                $table->text('description')->nullable(true);
                $table->integer('price')->default(0);
                $table->timestamps();
        });
}
```

Eloquent first steps

Éloquent est l'ORM utilisé par Laravel pour communiquer avec la base de données. Pour une utilisation optimale d'Éloquent, il faut veiller à respecter certaines règles.
La première concerne les modèles. Le nom d'un modèle doit être au singulier et en anglais. Et sa première lettre en majuscules. Éloquent utilise les noms des modèles qu'il va mettre au pluriel pour faire correspondre un nom de table dans notre BD. Cependant si on souhaite utiliser un nom de table différent du nom du modèle, il est possible de définir ce nom dans la classe du modèle concerné:
Dans l'exemple nous avons un modèle qui se nomme Utilisateurs. Dans la classe de ce modèle nous avons créé une variable de classe $table qui va contenir le nom de la table du modèle.
Les objets Eloquent font des appels statiques sur des méthodes qui vont faciliter grandement les interactions avec la base de données.

Eloquent: récupération des données

Nous pouvons récupérer tous les enregistrements sur un objet de la classe Products avec la méthode All() comme suite:

```
$products = App\Products::all();
```

La variable $products va contenir tous les enregistrements de la table Products.

```
App\Products::find(1);// va récuperer le produit avec l'id 1
App\Products::where('active', 1)->first();//Récupére tous les produits avec le champ active = 1 et retourne le premier.
```

À noter que la méthode where() peut être utilisée pour récupérer tous les enregistrements. Mais elle devra être accompagnée de la méthode get() dans ce cas.

```
App\Products::where('active', 1)->get();
```

Éloquente offre trois fonctions qui vont retourner des informations en agrégation.

- la méthode max() retourne la valeur max d'une collection
- la méthode sum() retourne la somme des valeurs d'une collection
- la méthode count() retourne le nombre total des valeurs d'une collection

Ces méthodes sont utilisées comme suites;

```
$count   = App\Products::where('active', 1)->count();//nbr total des produits actifs
```

Eloquent: insérer des données

Nous pouvons ajouter un enregistrement en utilisant la méthode save() avec un objet d'un modèle. Mais avant d'appeler cette méthode, il faut d'abord donner des valeurs aux différents champs que nous voulons enregistrer.

```
$produit = new App\Produit();
$produit->name = "Téléphone Nokia";
$produit->price = 90000;
$produit->save();
```

Nous pouvons aussi utiliser la méthode create(). Avec cette méthode nous faisons ce qu'on appel le mass assignement; c'est a dire le fait d'attribuer des valeurs à plusieurs champs en même temps. C'est une manipulation qui peut être dangereuse. Raison pour laquelle, avant de pouvoir utiliser la méthode create(). Nous devons d'abord définir les champs qui peuvent être inclus dans une "mass assignement" au niveau du modèle.

```
namespace App;
use Illuminate\Database\Eloquent\Modèle;
class Products extends Modèle
{
        protected $fillable = ['name'.'description','price'];
}
```

Dans la variable $fillable on liste les champs qui peuvent être inclus dans un enregistrement de masse. On peut ensuite faire:

```
$create_produit = App\Products::create([ 'name' => 'Chaussure en cuire, 'description'=>'la description',
'price'=>10000 ]);
```

Eloquent: mettre à jour des données

Nous pouvons mettre à jour des données après les avoir récupérées en utilisant la méthode Save.

```
$prod = App\Products::find(1);// on recupére le produit avec l'id 1
$prod->name = 'Change le nom du produit';// On attribue un nouveau nom au produit
$prod->save();
```

Nous pouvons aussi mettre plusieurs champs à jour avec la méthode update(). Cette méthode prend en paramètre un tableau associatif avec les noms des champs liés à leur nouvelle valeur. Cette méthode fait du 'mass assignment'. Il faudra donc définir dans le modèle les champs qui peuvent être inclus dans le 'mass assignment'.

```
App\Product::find(1)->update(['name'=>"Nouveau nom","price"=>12000]);
```

Eloquent: supprimer un élément

Nous pouvons supprimer un élément en utilisant la méthode delete() sur un objet.

```
$p = App\Product::find(1);
$p->delete();
```

Nous pouvons aussi utiliser la méthode static destroy(). Cette méthode peut prendre plusieurs Ids correspondant aux différents enregistrements à supprimer.

```
Product::destroy([1,4,5]);
```

Eloquent: firstOrCreate & updateOrCreate

La méthode firstOrCreate() va chercher un enregistrement qui a la même valeur dans la table. Si l'enregistrement est retrouvé, la méthode firstOrCreate() va le retourner. Sinon, elle crée un enregistrement avec les valeurs passées en paramètre.

```
$prod = App\Product::firstOrCreate(['name'=>"Chaussure en cuire","description"=>"De jolies chaussures en cuire"]);
```

updateOrCreate va permettre de mettre à jour un enregistrement si celui-ci existe déjà. Sinon elle crée l'enregistrement.

```
$prod = App\Product::updateOrCreate(['name'=>"Chaussures en cuire","price"=>"30000"]);
```

Si la méthode retrouve un produit avec comme nom "Chaussures en cuir", elle va mettre à jour son prix. Sinon, elle crée l'enregistrement

Formulaires & Validation des données

Dans notre application nous allons avoir besoin de récupérer des données de nos utilisateurs. Pour cela nous devons utiliser les formulaires.

Nous allons créer dans notre contrôleur une méthode qui va retourner une vue avec un formulaire.

```
public function create(){
        return view('products.create');
}
```

Dans la vue product/create.blade.php nous allons créer un formulaire qui va permettre d'enregistrer un produit. Nous utiliserons un formulaire HTML classique. Le traitement de ce formulaire se fera à l'adresse /product/store avec une méthode post.

```
<form action="{{route('store product')}}" method="post">
            <div class="row"><input type="text" name="name" id="name" class="form-control"
placeholder="Nom du prodtuit">
            </div>
            <div class="row"><input type="text" name="price" id="price" class="form-control"
placeholder="Prix du produit">
            </div>
            <div class="row">
                <button class="btn btn-primary">Valider</button>
            </div>
        </form>
```

Formulaires & Validation des données

Nous devons alors sur notre fichier web.php, créer une nouvelle url pour le traitement du formulaire. Cette url va pointer sur la méthode store() de notre contrôleur.

```
Route::post('/products/store','ProductsController@store')->name('store_product');
```

la méthode store:

```
public function store(Request $request)
{
        Products::create(['name'=>$request->input('name')]);
        return redirect()->back()->with('success',"Produit parfaitement enregistré");
}
```

Cette méthode est responsable de l'enregistrement d'un nouveau produit dans la base de données. Mais à ce niveau si nous essayons d'enregistrer nous allons avoir une erreur. En effet Laravel nous protège des attaques csrf. Il va s'assurer que les données à enregistrer proviennent de formulaires qui appartiennent à notre application. Pour cela il vérifie l'existence d'une clé sur chaque formulaire. Pour ajouter cette clé nous pouvons utiliser l'écriture blade @csrf à l'intérieur du formulaire.

```
<form action="{{route('store_product')}}" method="post">
        @csrf
        ...
```

Formulaires & Validation des données

Notre formulaire ne fait aucune vérification. Les données envoyées sont enregistrées directement dans la base de données. Laravel permet de valider les données d'un formulaire grâce à un système de validation. Nous pouvons valider les champs de notre formulaire avec ce système comme suite dans la méthode store.

```
$data = $request->validate([
        'name' => 'required | max:300 | min: 3',
        'price' => 'numeric | max:7 | required | min:2
        ]);
```

La méthode validate est une méthode de la classe Illuminate\Http\Request. Elle prend en paramètre un tableau dont les clés sont les champs de notre formulaire. Ces champs vont avoir comme valeur des indications qui vont permettre de les valider. Les validations sont séparées par '|':

- require: indique que le champ est obligatoire
- max: permet de fixer la longueur maximale du champ
- min: permet de fixer la longeur minimale du champ
- numeric: permet de s'assurer que le champ contient que des valeurs numériques.

Nous pouvons retrouver la liste complète des validations sur ce lien:
https://laravel.com/docs/6.x/validation#available-validation-rules

Formulaires & Validation des données

Notre méthode store va alors ressembler à cela

```
public function store(Request $request)
{
        $data = $request->validate([
        'name' => 'required | max:300 | min: 3',
        'price' => 'numeric | max:7 | required | min:2'
        ]);
        Products::updateOrCreate($data);
        return redirect('/');
}
```

Et au niveau de la vue nous mettrons les instructions pour afficher les erreurs au cas ou un champ est incorrect

```
@if($errors->any())
        @foreach($errors->all() as $error)
        <div class="alert alert-danger">{{$error}}</div>
        @endforeach
@endif
```

Eloquent les relations

Éloquent nous permet de très facilement gérer les différentes relations qui peuvent exister entre les tables de notre base de données. Nous allons voir dans ce document deux type de relation:

Le un à plusieurs

Le plusieurs à plusieurs

Eloquent les relations un à plusieurs

Imaginons dans notre application nous avons un modèle pour les produits nomé Product et un modèle pour les catégories nommé Category. Un produit appartient à une catégorie et une catégorie est liée à plusieurs produits.

Pour gérer cette relation nous devons avoir un champ dans la table des produits qui va être la clé étrangère de la catégorie à laquelle le produit est lié.

La manière de nommer ce champ suit une convention qui consiste à mettre le singulier de la table suivit de '_id'. Suivant cette convention le champ pour l'identifiant de la catégorie dans la table 'products' va être category_id.

Eloquent les relations un à plusieurs

Pour gérer la relation entre les deux tables nous allons utiliser la méthode belongsTo() et hasMany().

Un produit appartient à une catégorie. Nous devons donc créer une méthode category() qui va retourner la relation belongsTo() dans la classe Product. Cette relation nous permettra de récupérer la catégorie liée à un produit.

```
class Product extends Modèle
{
        public function category() {
                return $this->belongsTo('App\Post');
        }
}
```

Nous allons utiliser la relation hasMany() pour la relation inverse dans la classe Category().

```
class Category extends Modèle
{
        //Cette méthode va nous permettre de récupérer les produits liés à une catégorie.
        public function product() {
                return $this->hasMany('App\Post');
        }
}
```

Eloquent les relations un à plusieurs

Une relation plusieurs à plusieurs va toujours générer une autre table. Dans cette autre table nous allons avoir les clés étrangères des tables en associations.

Prenons le cas d'une relation entre une table 'commandes' qui va contenir les commandes et une table pour les produits. Un produit peut figurer dans plusieurs commandes et une commande peut contenir plusieurs produits. Pour représenter cela nous allons d'abord créer la table d'association.

```
php artisan make:migration JoinCommandeProduct
```

Cette commande va générer un fichier de migration qui va contenir une méthode up() et une méthode down().

Dans la méthode up() on peut ajouter les champs pour les clés étrangères.

```
public function up()
        {
                Schema::create('profile_user', function(Blueprint $table){
                        $table->bigIncrements('id');
                        $table->unsignedBigInteger('profile_id');
                        $table->unsignedBigInteger('user_id');
                        $table->timestamps();
                });
        }
```

Eloquent les relations

Nous devons maintenant créer les relations dans les modèles concernés avec la relation belongsToMany().

```
//dans le modèle Product
        public function commandes(){
                return $this->belongsToMany('App\Commande');
        }

//dans le modèle Commande
        public function products(){
                return $this->belongsToMany('App\Product');
        }
```

La création de ces relations, nous permet maintenant récupérer les commandes liées à un produit.

```
$prod = App\Product::find(1);//On récupére le produit avec l'id 1

foreach ($prod->commandes as $commande) {
        echo $commande->prix_total;
}
```

Authentification

Laravel intègre un système de gestion des utilisateurs très avancé. Pour utiliser ce système il faut d'abord installer la library ui de laravel.

```
composer require laravel/ui --dev
```
Nous pouvons alors utiliser cette librairie pour ajouter les vues nécessaires à l'authentification.

```
php artisan ui vue --auth
```
Laravel crée aussi par défaut une migration pour la table user. Nous pouvons donc faire php artisan migrate pour créer la table.

Le système vient avec tout ce dont peut avoir besoin sur un système d'authentification.

Si une page est réservée aux utilisateurs connectés nous pouvons utiliser le middleware('auth') pour ajouter cette restriction.

```
Route::get('/', "ProductsController@index")->middleware('auth');
```
Nous pouvons aussi ajouter ce middleware sur le constructeur d'un contrôleur. Cela fera que toutes les actions d'un contrôleur ne seront accessibles que pour les utilisateurs connectés.

Authentification

Pour travailler avec les utilisateurs nous avons une classe Illuminate\Support\Facades\Auth qui va nous être très utile.

Nous pouvons vérifier si un utilisateur est connecté avec la méthode static check()

```
use Illuminate\Support\Facades\Auth;
    ...
   Auth::check() ;//cette méthode va retourner true si le user est connecté.
```

Nous pouvons aussi récupérer l'utilisateur qui est actuellement connecté.

```
use Illuminate\Support\Facades\Auth;
...
$user = Auth::user()
```

Liens utiles

GALIMATECH
L'EXPERIENCE DIGITALE

- https://laravel.com/docs/6.x