Fascicule de l'atelier Framework côté serveur : Le Framework Laravel

Houcem Hedhly

Fascicule de l'atelier Framework Web côté serveur

Le Framework Laravel

Houcem Hedhly

ISET Bizerte 2018 - 2019

DSI 21 & 22

Love beautiful code? We do too.

The PHP Framework For Web Artisans

Table des matières

Présentation

Objectif général

Orienter les étudiants vers la maîtrise des concepts relatifs au développement des sites Web dynamiques à travers un Framework Web côté serveur.

Objectifs spécifiques

- Comprendre l'intérêt des Frameworks
- Dévoiler l'architecture logicielle d'un Framework
- Comprendre le mécanisme de routage dans un Framework
- Connaître le rôle du contrôleur
- Connaître le principe d'ORM
- Créer un RESTful API avec Laravel

Prérequis

Les notions de base en :

- HTML et CSS
- langage PHP
- programmation orientée objet
- base de données

Atelier 1 : Préparation de l'environnement et installation

A la fin de cet atelier vous serez capables de :

- Préparer l'environnement de développement Laravel
- Créer un nouveau projet Laravel
- Comprendre la structure et le rôle des fichiers dans le projet

1. Prérequis

- PHP >= 7.1.3
- Les extensions PHP :
  - OpenSSL
  - PDO
  - Mbstring
  - Tokenizer
  - XML
  - Ctype
  - JSON
  - BCMath

  } WAMP / XAMPP
- Serveur Web (Apache)
- MySQL
- Editeur de texte (Visual Studio Code, Atom, Brackets, Sublime Text, …)
- Invite de commande (Power Shell, Git Bash, Cmder, …)

2. Installation

2.1. Composer

Composer est un outil de gestion des dépendances pour PHP (**Dependancy Manager**). Il permet de déclarer les bibliothèques, desquelles dépend un projet en PHP, et il va les gérer (installer / mettre à jour).

Pour télécharger et installer Composer :

https://getcomposer.org/download/

2.2. Laravel

L'installation de Laravel sera faite via Composer en exécutant la commande suivante dans le dossier choisi pour héberger le projet Laravel :

```
$ composer create-project --prefer-dist laravel/laravel nomprojet
```

Activité :

> Créer un projet nommé **dsiBank** avec la dernière version de Laravel.

2.3. Choix de version

Il faut vérifier la configuration de l'environnement utilisé avant de créer le projet. Si la configuration existante ne supporte pas la dernière version de Laravel il faut créer le projet dans une version plus ancienne du Framework (par exemple pour la version Laravel 5.4, on doit avoir PHP>=5.6.4). Dans ce cas on est obligé de spécifier la version après le nom de projet dans la commande composer:

```
$ composer create-project --prefer-dist laravel/laravel nomprojet version
```

3. La structure des fichiers

- `App\User.php` : ce fichier est une classe **Model**. Tous les modèles de Laravel seront typiquement stockés dans le dossier **App**.
- `App\Http\Controllers` : ce dossier contiendra les différents **Controllers** pour notre application. Par défaut il y a un **Core Controller**, et un dossier **Auth** contenant les Controllers nécessaires à la fonction d'authentification (sera traité plus tard dans l'atelier 8).
- `Resources\views` : c'est le dossier qui stockera les différentes **Views** d'une application. Laravel utilise **Blade** comme Templating Engine (moteur de génération de Templates) : HTML + propriétés dynamiques.
- `Routes\` : c'est le dossier qui prendra en charge l'hébergement de tous les fichiers des **Routes** de notre application. Souvent, et dans la majorité des cas, toutes les **Routes** seront définies dans le fichier **web.php**.
- `.env` : c'est le fichier de configuration de l'environnement Laravel.
- `package.json` : c'est le fichier dédié à définir les dépendances de notre application.
- `Public\` : ce dossier contient tous les fichiers publics (HTML, CSS, JS, …)

Atelier 2 : Le Routing, les Controllers et la CLI

A la fin de cet atelier vous serez capables de :

- Comprendre le mécanisme de routage avec Laravel
- Utiliser la CLI
- Créer et manipuler un Controller
- Lier un Controller à une Route

1. Routing

Toutes les Routes de Laravel sont définies dans le dossier **Routes**. Ces fichiers sont chargés automatiquement par le Framework depuis ce dossier.

Pour la majorité des applications Laravel, les Routes sont définies dans le fichier **routes\web.php**. Ce fichier contient les Routes de l'interface Web de l'application.

Voici la Route défini par défaut par Laravel :

```
Route::get('/', function(){
   return view('welcome') ;
}) ;
```

Cette Route :

- prend en charge une requête GET (**GET request**),
- est utilisée pour la page d'accueil (**Home : '/'**),
- renvoie la **View** (page) nommée **welcome**.

Activité :

a) Ecrire, dans le fichier **web.php**, une Route permettant de diriger vers la page **about**.
b) Créer la page **about**.

Les méthodes du Routing disponibles sont :

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::delete($uri, $callback);
Route::patch($uri, $callback);
Route::options($uri, $callback);
```

On a souvent besoin de faire passer des paramètres dans les URL. Pour faire passer ce genre de paramètres on peut utiliser la syntaxe suivante :

```
Route::get('/user/{id}', function($id){
   return 'User id= '.$id ;
}) ;
```

2. Controllers

Au lieu de définir la logique de réponse aux différents requêtes dans les fichiers du Routing comme on a vu dans les exemples précédents, il est recommandé d'organiser ces comportements dans des **Controllers**.

Les Controllers permettent de regrouper les logiques de réponse aux requêtes http reliées dans la même classe.

On trouve les Controllers dans le dossier **App\Http\Controllers**.

La création des Controller peut être faite manuellement, cependant, l'utilisation de la console intégrée de Laravel : **Artisan** s'avère une meilleure solution.

3. Artisan

Artisan est une interface de ligne de commande (**CLI : Commande Line Interface**) intégrée avec Laravel. Il permet de fournir un grand nombre de commandes très utiles lors du développement d'une application.

Pour lister les commandes Artisan :

```
$ php artisan list
```

Pour créer un Controller avec Artisan on utilise la commande :

```
$ php artisan make:controller NameController
```

Voici un exemple permettant de créer un Controller nommé **PostsController** :

```
$ php artisan make:controller PostsController
```

L'idée est de faire le Routing des différentes requêtes vers le Controller puis de définir dans ce dernier des méthodes permettant de répondre aux différents Routes.

Pour diriger une requête vers un Controller, il faut spécifier son nom dans le Route ainsi que le nom de la méthode permettant de répondre à cette requête comme le montre cette syntaxe :

```
Route::get('uri', 'NameController@method') ;
```

Voici un exemple permettant de gérer la requête de la page **posts** et de faire la redirection vers la méthode **index()** du Controller **PostsController**:

```
Route::get('/posts', 'PostsController@index') ;
```

Application :

a) Créer un Controller nommé **PagesController** en utilisant une commande Artisan.
b) Créer, dans **PagesController**, une méthode nommée **index()** permettant de prendre en charge la requête demandant la page **index**.
c) Créer une deuxième méthode nommée **about(),** permettant de répondre à la requête demandant la page nommée **about**.
d) Modifier le fichier **routes\web.php** pour rediriger les requêtes vers les méthodes définies dans les questions précédentes.

Atelier 3 : Le Templating avec Blade et les Assets de compilation

A la fin de cet atelier vous serez capables de :

- Comprendre les principes du Blade Templating Engine
- Créer des Views avec Blade
- Comprendre et configurer Laravel Mix

1. Blade

1.1. Introduction

Le Framework Laravel intègre un **Templating engine** (moteur de génération de Templates) simple et très puissant : **Blade**.
Contrairement aux autres Templating engines PHP populaires (Twig, Mustache, …) Blade n'empêche pas l'utilisation du code PHP dans les Views. En fait, toutes les Views Blade sont compilées dans du code PHP pur, et mises en cache jusqu'à ce qu'elles soient modifiées, ce qui signifie que Blade ajoute zéro charge à votre application.

Les Views créées avec Blade utilisent l'extension **.blade.php** et sont, typiquement, enregistrées dans le dossier **resources/views**.

1.2. Définir un Layout

Activité 1 :

a) Créer, dans le dossier dédié aux Views par défaut, un dossier nommé **layouts**.
b) Créer dans ce dossier un fichier Blade nommé **myapp**.
c) Définir dans ce fichier un layout qui doit être utilisé par les autres Views.

L'idée est de mettre dans ce fichier **myapp.blade.php** le contenu qui sera utilisé dans toutes pages (le layout général). Il suffit ensuite d'intégrer les sections dynamiques avec ce layout dans les Views.

L'instruction **@yield** est utilisée dans les Views pour afficher le contenu d'une section définie ailleurs.

```
…
<body>
    @yield('content')
</body>
…
```

1.3. Utiliser un Layout

Activité 2 :

> En utilisant le layout créé dans l'activité précédente, modifier la View **welcome.blade.php** afin qu'elle intègre ce layout.

Pour permettre à une View d'utiliser une autre, il faut utiliser l'instruction Blade **@extends**. Cette instruction permettra de spécifier quel layout la View doit **hériter**.

Les Views qui héritent un layout Blade peuvent en injecter du contenu en utilisant l'instruction **@section**. Rappelez-vous que, dans la section précédente, on a utilisé l'instruction **@yield** pour afficher un contenu dynamique à partir d'une autre View.

```
@extends('layouts.app')
@section('content')
     {{-- contenu à injecter --}}
@endsection
```

1.4. Passage des variables

Activité 3 :

a) Ajouter, dans chacune des méthodes du **PagesController**, une variable pour contenir les titres des pages.
b) Passer ces variables en paramètre vers la View.
c) Afficher le contenu des variables passées en paramètre dans les Views.

Voici à quoi doit ressembler la classe **PagesController** après les modifications :

```
class PagesController extends Controller
{
    public function index(){
      $title = 'Welcome';
      return view('welcome',compact('title'));
    }

    public function about(){
      $title = 'About';
      return view('about')->with('title',$title);
    }
}
```

Pour pouvoir afficher les valeurs des variables passées vers la View dans le Controller, on doit utiliser les accolades pour envelopper ces variables dans les Views : **{{ $title }}**.

Cette syntaxe des accolades ne permet pas seulement d'afficher le contenu des variables passées en paramètre aux Views, mais aussi d'afficher les résultats des fonctions PHP prédéfinies. Exemple :

```
Le timestamp UNIX à l'instant est : {{ time() }}
```

Application 1 :

a) Ajouter, dans le **PagesController**, une méthode nommée **services** permettant de définir un tableau de données contenant :
   - Le titre de la page
   - La liste des services dans une banque : création compte, versement, retrait et transfert d'argent.

b) Créer, en utilisant le layout créé précédemment, la View nommée **services**.

c) Utiliser le tableau de données fourni par le Controller pour afficher le titre et la liste des services

d) Ajouter dans le fichier **routes/web.php** le Route permettant d'accéder à la page des services.

2. Assets

2.1. Introduction

Laravel intègre par défaut **Laravel Mix** qui fournit une API fluide pour définir les étapes de construction du Webpack de votre application en utilisant plusieurs préprocesseurs CSS et JS.

2.2. Installation et configuration

La View livrée par défaut avec Laravel **welcome.blade.php** utilise un style interne défini dans le **<head>** du fichier.

Laravel est livré aussi avec un autre fichier de style qui contient tous les styles prédéfinis (y compris **Twitter Bootstrap**). Pour que nos pages puissent utiliser ces styles précompilés, on doit ajouter le lien nécessaire dans le layout :

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

Ce lien permettra de référencer le fichier **public/css/app.css** qui est une compilation des fichiers **SASS** (le préprocesseur CSS intégré avec Laravel), et qui se trouve dans le dossier **resources/assets/sass**.

Pour pouvoir utiliser les modules importés par le fichier **app.scss** (y compris Bootstrap), et qui sont déjà définis dans le fichier **package.json**, on a besoin d'exécuter la commande suivante (**npm** doit être installé auparavant) :

```
$ npm install
```

Cette commande permettra d'installer tous les modules complémentaires définis comme des dépendances à l'application Laravel.

Le **npm** (**Node Package Manager**) est un gestionnaire de packages pour JavaScript, et peut être installé via **Node.js** en visitant cette adresse :

https://nodejs.org/en/download/

2.3. Modification et exécution

Pour modifier la mise en forme du style prédéfini, il faut modifier dans le fichier **ressources/assets/sass/_variables.scss** et non pas dans le fichier **public/css/app.css**, car ce dernier est la version compilée du premier.

Toute modification doit être suivi de recompilation des fichiers source en exécutant la commande :

```
$ npm run dev
```

Pendant la phase de développement (production), où vous avez besoin de recompiler plusieurs fois pour tester, vous pouvez utiliser la commande suivante qui permettra de surveiller vos fichiers et recompiler automatiquement après chaque enregistrement :

```
$ npm run watch
```

Pour ajouter un style personnalisé, il est recommandé de créer un nouveau fichier **resources/assets/sass/_custom.css** et d'y définir les styles personnalisés souhaités. Si c'est le cas, il ne faut pas oublier d'importer ce fichier dans le fichier **resources/assets/sass/app.scss** en ajoutant à la fin la ligne suivante :

```
@import "custom" ;
```

Application 2 :

> Créer un **navbar**, en utilisant Bootstrap, contenant des liens vers les trois pages créées précédemment : **home**, **about** et **services**.

Atelier 4 : BD : Models, Migrations & Seeders

A la fin de cet atelier vous serez capables de :

- Comprendre et utiliser un Model
- Créer et exécuter une Migration
- Créer et exécuter un Seeder

1. Introduction

Le Framework Laravel permet une interaction très facile avec les BD à travers une variété de backends de BD qui permettent d'utiliser soit le langage **SQL** brut, un **Query Builder** très fluide et l'**ORM Eloquent**.

Laravel supporte initialement quatre SGBD :

- MySQL
- PostgreSQL
- SQLite
- SQL Server

2. Configuration

La configuration de la BD se trouve dans **config/database.php**. Dans ce fichier on peut définir toutes les connexions de BD et choisir laquelle utiliser par défaut.

La configuration dans ce fichier utilise des variables d'environnement le l'application Laravel qu'on trouve dans le fichier **.env**.

Activité :

a) Ouvrir l'interface **phpmyadmin** et créer une nouvelle base de données nommée **dsibank**.
b) Configurer l'application Laravel avec la nouvelle BD dsibank.

3. Models

4.1. Définition

L'**ORM Eloquent** inclus avec Laravel fournit un outil simple pour travailler avec la BD.

A chaque table dans la BD doit correspondre un **Model** qui sera utilisé pour interagir avec la table.

4.2. Création d'un Model

Tous les **Models** de l'application Laravel sont stockés dans le dossier **App**. Pour créer un model il suffit d'exécuter la commande :

```
$ php artisan make:model NomModel
```

4.3. Les conventions d'Eloquent pour les Models

4.3.1. Nom de la table

Par convention le nom du Model est le singulier du nom de la table qi lui correspond dans la BD. Si le nom de la table est différent, il faut le spécifier dans le model :

```
protected $table = 'my_clients';
```

4.3.2. Clé primaire

Eloquent suppose également que chaque table possède une colonne de clé primaire appelée **id**. Vous pouvez définir une propriété **$primaryKey** pour remplacer cette convention :

```
public $primaryKey = 'ncin' ;
```

De plus, Eloquent suppose que la clé primaire est une valeur entière incrémentée, ce qui signifie que, par défaut, la clé primaire sera automatiquement convertie en **int**. Si vous souhaitez utiliser une clé primaire non incrémentée ou non numérique, vous devez définir la propriété publique **$incrementing** sur votre modèle sur **false**.

```
public $incrementing = false ;
```

4.3.3. Timestamps

Par défaut, Eloquent s'attend à ce que les colonnes **created_at** et **updated_at** existent sur vos tables. Si vous ne souhaitez pas que ces colonnes soient gérées automatiquement par Eloquent, définissez la propriété **$timestamps** sur votre modèle sur **false**:

```
public $timestamps = false;
```

4. Migrations

Les **Migrations** sont une sorte de contrôleur de version pour la BD, permettant, à une équipe, de modifier et partager le schéma d'une BD facilement.

Les Migrations sont stockées dans le dossier **database/migrations**. Pour créer une migration, il suffit d'exécuter la commande suivante :

```
$ php artisan make:migration create_table_name
```

L'exécution d'une Migration permet de créer la table dans la BD avec le schéma défini dans la Migration. La commande permettant d'exécuter les Migrations est :

```
$ php artisan migrate
```

Il existe une commande qui permet de créer une Migration avec le Model :

```
$ php artisan make:model --migration NomModel
```

Ou bien :

```
$ php artisan make:model -m NomModel
```

Application 1 :

a) Créer, en utilisant une seule commande, un Model nommé **Client** ainsi qu'une Migration permettant de créer la table correspondant au Model
b) Ajouter, dans la Migration, le code nécessaire permettant de créer la table dans la BD sachant que : **clients (id, nom, prenom, dateNaissance, adresse, tel)**
c) Lancer la commande permettant d'exécuter la Migration.

5. Seeders :

Laravel comprend une méthode simple pour remplir votre BD avec des données pour le test. Toutes les classes **Seed** sont stockées dans le répertoire **database/seeds**.

Par convention, les noms des classes de **Seeding** sont écrits dans ce format : **UsersTableSeeder**.

Pour générer un **Seeder**, il suffit d'exécuter la commande suivante :

```
$ php artisan make:seeder UsersTableSeeder
```

Une classe **Seeder** ne contient qu'une seule méthode par défaut: **run()**, dans laquelle vous pouvez insérer des données dans votre BD comme vous le souhaitez. Vous pouvez utiliser le **Query Builder** pour insérer manuellement des données ou vous pouvez utiliser les **Eloquent Model Factories**. Laravel intègre une bibliothèque très utile dans ce contexte nommé **Faker**, qui permet de générer des fausses données à insérer dans la BD.

Pour exécuter un Seeder, il suffit d'utiliser la commande :

```
$php artisan db:seed --class=UsersTableSeeder
```

Application 2 :

a) Créer, en utilisant la ligne de commande, un Seeder nommé **ClientsTableSeeder**.
b) Ajouter, dans le Seeder, le code nécessaire permettant de remplir la table **Clients** par des données de test.
c) Exécuter le Seeder puis vérifier la table dans l'interface **phpmyadmin**.

Atelier 5 : Récupération des données avec Eloquent

A la fin de cet atelier vous serez capables de :

- Générer un Controller avec méthodes prédéfinies
- Générer des Routes pour les méthodes prédéfinies d'un Controller
- Récupérer les données depuis la BD

1. Générer un Controller avec ressources

On peut assigner les Routes du typique **CRUD** (Create, Read, Update & Delete) à un Controller avec une seule ligne commande grâce au Routing de ressources.

Pour créer un Controller avec ressources on utilise la commande suivante :

```
$ php artisan make:controller ControllerNom --resource
```

Ou bien :

```
$ php artisan make:controller ControllerNom -r
```

Cette commande permettra de générer un Controller qui contiendra une méthode pour chacune des opérations :

`index()` `create()` `store()` `show()` `edit()` `update()` `destroy()`

Pour référencer ces ressources dans le fichier des Routes, on n'a pas besoin d'écrire un Route pour chaque méthode, on peut écrire une seule ligne permettant de générer les Routes pour toutes les ressources à la fois :

```
Route::resource('photos, 'PhotosController') ;
```

On peut lister les Routes de l'application en exécutant cette commande :

```
$ php artisan route:list
```

Application 1 :

a) Créer un Controller avec ressources nommé **ClientsController**.
b) Lister les Routes définis dans l'application.
c) Modifier le fichier **routes/web.php** en ajoutant la ligne permettant de générer les Routes de ressources pour les clients.
d) Re-lister les Routes de l'application.

2. Récupération des données

2.1. La méthode index()

Il faut considérer que chaque Model Eloquent comme un Query Builder puissant permettant d'interroger couramment la table de la BD qui lui est associée.

La première étape est d'indiquer au Controller quel Model il va utiliser. Ensuite, dans la méthode **index()**, ajouter la méthode prédéfinie **all()**, et Eloquent va se charger de récupérer toutes les lignes de la table.

Voici un exemple :

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Flight;

class FlightsController extends Controller
{
    public function index()
    {
        $flights = Flight::all();
        return $flights;
    }
```

La méthode **all()** permet ainsi de récupérer tous les enregistrements de la table en question. On peut ajouter des contraintes à nos requêtes et utiliser ainsi la méthode **get()** pour les récupérer :

```
$flights = App\Flight::where('active', 1)
              ->orderBy('name', 'desc')
              ->take(10)
              ->get();
```

Pour afficher les résultats de la requête, il suffit de parcourir la variable qui héberge le résultat :

```
foreach ($flights as $flight) {
    echo $flight->name;
}
```

Application 2 :

a) Ajouter, dans la méthode **ClientsController@index**, une requête permettant de récupérer la liste des clients depuis la BD.
b) Envoyer le résultat obtenu vers une View nommée **index**.
c) Créer dans le répertoire **resources\views** un nouveau répertoire nommé clients.
d) Créer dans le répertoire **resources\views\clients** une View nommée **index.blade.php**.
e) Afficher la liste des clients envoyée par le **ClientsController** dans la nouvelle View.

2.2. La méthode show()

Eloquent nous permet également de récupérer des enregistrements individuels en utilisant **find()** ou **first()**. Au lieu de renvoyer une collection de modèles, ces méthodes renvoient une seule instance de modèle comme le montre les exemples suivants :

```
// Récupérer un Model par sa clé primaire...
$flight = App\Flight::find($id);


// Récupérer le premier Model correspondant à la contrainte de la requête...
$flight = App\Flight::where('active', 1)->first();
```

Application 3 :

a) Ajouter, dans la méthode **ClientsController@show**, une requête permettant de récupérer le client avec l'**id** passé en paramètre de la méthode.
b) Envoyer le résultat obtenu vers une View nommée **show**.
c) Créer, dans le répertoire **resources\views\clients**, une View nommée **show.blade.php**.
d) Afficher, dans la nouvelle View, le client envoyé par le **ClientsController**.

Application 4 :

a) Ajouter, dans le **navbar**, un lien permettant de naviguer vers la page contenant la liste des clients.
b) Ajouter, dans la méthode **ClientsController@index**, une contrainte à la requête permettant d'ordonner les clients par prénom dans l'ordre ascendant.
c) Ajouter la méthode permettant de paginer le résultat de la requête
d) Modifier la View **resources\views\clients\index.blade.php** pour afficher les liens de pagination.
e) Ajouter dans cette View un bouton permettant de revenir vers la page contenant la liste des clients.

Atelier 6 : Les formulaires et l'insertion de données

A la fin de cet atelier vous serez capables de :

- Créer un formulaire en utilisant Laravel Collective
- Insérer des données dans la BD
- Implémenter les messages d'alerte

Activité :

a) Ajouter, dans la méthode **ClientsController@create**, le code nécessaire permettant de retourner la View **clients/create.blade.php**.
b) Créer la View **clients/create.blade.php**.

1. Laravel Collective

1.1. Définition

Laravel Collective est une communauté qui a pour mission de maintenir certains composants qui ont été supprimé du noyau Laravel. Ces composants qui ont été abandonnés par les développeurs du noyau Laravel sont demandés par certains développeurs qui les utilisaient et qui souhaitent continuer à les utiliser. Pour plus d'information, visitez le site https://laravelcollective.com/

1.2. Installation

Pour commencer, il faut installer le package en utilisant **Composer**. La commande est :

```
$ composer require "laravelcollective/html":"^5.4.0"
```

Une fois Composer termine l'installation des packages Collective, il faut ajouter dans le tableau nommé **providers** dans **config/app.php** la ligne suivante :

```
'providers' => [
    // ...
    Collective\Html\HtmlServiceProvider::class,
    // ...
],
```

Dans le même fichier **config/app.php**, ajouter dans le tableau **aliases** la ligne suivante :

```
'aliases' => [
  // ...
    'Form' => Collective\Html\FormFacade::class,
    'Html' => Collective\Html\HtmlFacade::class,
  // ...
],
```

1.3. Ouvrir un formulaire

La syntaxe pour ouvrir un formulaire est la suivante :

```
{!! Form::open(['action' => 'Controller@method']) !!}
    //
{!! Form::close() !!}
```

Par défaut la méthode **POST** est assigné au formulaire. Si on souhaite utiliser une autre méthode il faut ajouter un deuxième élément au tableau de paramètres :

```
open(['action' => 'Controller@method' , 'method' => 'put'])
```

1.4. Les éléments d'un formulaire

- **Label**

Pour générer un label on utilise le code suivant :

```
{{ Form::label('prenom', 'Prénom') }}
```

Dans cet exemple la méthode **label()** va générer l'élément HTML suivant :

```
<label for="prenom">Prénom</label>
```

- **Input**

Pour générer un input de type texte on utilise le code suivant :

```
{{ Form::text('prenom', '', ['class'=>'form-control',
 'placeholder'=>'Saisir le prénom ici']) }}
```

Dans cet exemple la méthode **text()** permettra de générer l'élément HTML suivant :

```
<input type="text" id="prenom" class="form-control"
placeholder="Saisir le prénom ici" />
```

- **Submit**

Pour générer un bouton submit on utilise le code suivant :

```
{{ Form::submit('Enregistrer',['class'=>'btn btn-success']) }}
```

Dans cet exemple la méthode **submit()** permettra de générer l'élément HTML suivant :

```
<input type="submit" value="Enregistrer" class="btn btn-success">
```

Application 1 :

> Ajouter, dans la View **clients/create.blade.php**, le code nécessaire permettant de créer le formulaire de saisi d'un nouveau client.

2. La méthode store()

2.1. Validation de la saisie

Laravel fournit plusieurs approches pour valider les données entrantes de votre application.

Chaque Controller dans le Framework Laravel utilise par défaut le Controller de base de Laravel **App\Http\Controllers\Controller**. Si on examine cette classe on peut voir qu'elle utilise à son tour une classe nommée **ValidatesRequests** permettant de fournir une méthode nommée **validate()**, qui peut être pratique pour tous les Controllers.

Voici un exemple d'utilisation de cette méthode **validate()** :

```
$this->validate($request, [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ]);
```

Application 2 :

> Ajouter, dans la méthode **ClientsController@store**, le code nécessaire permettant la validation des données saisies dans le formulaire d'ajout d'un nouveau client.

2.2. Enregistrer les données

Un fois les données saisies par l'utilisateur sont vérifiées et validées, on passe à la récupération et l'enregistrement. Cette action consiste en fait à la création d'un nouvel enregistrement au niveau de la table de la BD.

Pour ce faire, on doit tout simplement créer une nouvelle instance de notre **Model**, affecter les valeurs récupérées depuis le formulaire aux attributs de l'objet, puis appeler la méthode **save()**.

Voici un exemple illustrant les étapes décrites ci-dessus :

```
public function store(Request $request)
    {
        $flight = new Flight;
        $flight->name = $request->name;
        $flight->save();
    }
```

Application 3 :

Ajouter, dans la méthode **ClientsController@store**, le code nécessaire permettant l'insertion, dans la BD, des données saisies dans le formulaire d'ajout d'un nouveau client.

3. Les messages d'alerte

L'ajout des messages d'alertes dans Laravel est aussi simple. Laravel contient une classe nommé **MessageBag** qui contient une variété de méthodes pratiques pour utiliser les messages d'erreur.

Une variable prédéfinie **$errors**, qui est une instance de cette classe, est automatiquement instancié et mise à la disposition dans toutes les **Views**.

Activité :

a) Créer un fichier **resources/views/layouts/messages.blade.php**.
b) Ajouter le code suivant dans la View :

```
@if(count($errors)>0)
    @foreach($errors->all() as $error)
        <div class="alert alert-danger">
            {{$error}}
        </div>
    @endforeach
@endif
@if(session('success'))
    <div class="alert alert-success">
        {{session('success')}}
    </div>
@endif
@if(session('error'))
```

```
    <div class="alert alert-danger">
        {{session('error')}}
    </div>
@endif
```

c) Inclure ce fichier dans le fichier **resources/views/layouts/myapp.blade.php**.

d) Ajouter un message dans la redirection de l'application précédente :

```
return redirect('/clients')->with('success','Client ajouté');
```

Atelier 7 : Modification et suppression de données

A la fin de cet atelier vous serez capables de :

- Modifier des données stockées dans la BD
- Supprimer des données stockés dans la BD

Activité :

a) Avec la **CLI**, exécuter la commande permettant de lister tous les Routes.
b) Ajouter, dans le fichier **resources/views/clients/show.blade.php**, un bouton **Editer** permettant de naviguer vers le Route nommé **clients.edit**.
c) Ajouter, dans le même fichier, un bouton **Supprimer** permettant de naviguer vers le Route nommé **clients.destroy**.

1. L'interface d'édition

Pour répondre au **Request** qui sera généré par le bouton **Editer** créé dans l'activité précédente, on a besoin de le faire au niveau de la méthode **ClientsController@edit**.

Le retour de cette méthode doit être une View permettant d'éditer les données de l'enregistrement déjà enregistrées dans la BD.

Application 1 :

1. Ajouter dans la méthode **ClientsController@edit** le code nécessaire à l'extraction de l'enregistrement demandé de la BD, puis envoyer ces données vers une View nommée **edit.blade.php**.
2. Créer le fichier **resources/views/clients/edit.blade.php**
3. Ajouter le code nécessaire pour afficher un formulaire contenant les données renvoyées par le la méthode **ClientsController@edit**, sachant que les données modifiées sur ce formulaire doivent être envoyées vers la méthode **ClientsController@update**.

2. La méthode update()

Le rôle de la méthode **ClientsController@update** est semblable à celui de la méthode **ClientsController@save**. La seule différence à noter ici est qu'on doit récupérer les données de l'enregistrement à partir de la BD, puis les remplacer par les données envoyées depuis la View d'édition avant de les, enfin, stocker dans la BD.

Il ne faut pas oublié ici que, comme dans la méthode **store()**, les données doivent être validées avant d'être enregistrées.

Application 2 :

1. Ajouter, dans la méthode **ClientsController@update**, le code nécessaire permettant la validation des données saisies dans le formulaire d'édition d'un client.
2. Ajouter, par la suite, le code nécessaire à l'extraction de l'enregistrement demandé de la BD (en utilisant l'**$id**)
3. Affecter les données récupérées depuis le formulaire aux attributs de l'enregistrement.
4. Enregistrer les modifications dans la BD puis faire une redirection vers le Route nommé **clients.show** pour afficher le client en cours (ne pas oublier un message d'alerte).

3. La méthode destroy()

Cette méthode est utilisée pour supprimer un enregistrement de la BD. La démarche est simple : trouver l'enregistrement puis le supprimer.

Application 3 :

1. Ajouter dans la méthode **ClientsController@destoy** le code nécessaire permettant d'extraire l'enregistrement en question de la BD (en utilisant l'**$id**).
2. Supprimer l'enregistrement en question.
3. Faire une redirection vers le Route nommé **clients.index** avec un message d'alerte.

Atelier 8 : Authentification

A la fin de cet atelier vous serez capables de :

- Implémenter la fonction d'authentification
- Intégrer les fichiers générés dans le travail déjà réalisé

1. Introduction

L'implémentation de l'authentification avec Laravel est très simple. En fait, presque tout est déjà configuré et prêt à l'emploi. Le fichier de configuration d'authentification se trouve dans **config/auth.php**, qui contient plusieurs options bien documentées pour modifier le comportement des services d'authentification.

Par défaut, Laravel contient déjà un Model prédéfini **App\User**, qui peut être utilisé pour assurer l'implémentation de l'authentification.

Laravel est livré aussi avec des Controllers prédéfinis dédiés à l'authentification. Ces Controllers sont dans le répertoire **App\Http\Controllers\Auth**. Le **RegisterController** gère les nouvelles inscriptions, **LoginController** prend charge des connexions, **ForgotPasswordController** gère les emails envoyés pour réinitialiser les mots de passe, et **ResetPasswordController** gère l'opération de réinitialisation elle-même.

Deux Migrations sont également prédéfinies dans le répertoire database\migrations : create_users_table.php et create_password_resets_table.php, et qui serviront à créer les tables nécessaires pour mettre en marche le système d'authentification prédéfini.

2. make:auth

2.1. Routing

Laravel fournit une manière très simple pour mettre en place la dernière pièce du puzzle de l'authentification. Une simple commande est utilisée pour construire les Routes nécessaires à ce processus :

```
$ php artisan make:auth
```

Cette commande devrait être utilisée sur de applications nouvellement créées, et installera une View de Layout, des Views pour l'inscription et la connexion, ainsi que des Routes pour tous les Requests d'authentification. Un **HomeController** sera également généré pour gérer les demandes post-connexion au tableau de bord de votre application.

2.2. Views

Comme mentionné dans la section précédente, la commande **make:auth** créera toutes les Views dont vous avez besoin pour l'authentification et les placera dans le répertoire **resources/views/auth**.

La commande **make:auth** créera également un répertoire **resources/views/layouts** contenant une mise en page de base pour votre application. Toutes ces Views utilisent le Framework CSS **Bootstrap**.

Application :

1. Exécuter la commande permettant de générer le système d'authentification prédéfini de Laravel.
2. Modifier le fichier **resources/views/layouts/app.blade.php** pour ajouter les liens vers les pages créées précédemment.

Atelier 9 : Les relations dans la BD

A la fin de cet atelier vous serez capables de :

- Paramétrer une relation un-à-un
- Paramétrer une relation un-à-plusieurs

1. Introduction

Les tables de la BD sont souvent liées les unes aux autres. Par exemple, un article de blog peut contenir de nombreux commentaires, ou une commande peut être liée à l'utilisateur qui l'a placé. **Eloquent** facilite la gestion et le travail avec ces relations, et prend en charge plusieurs types de relations (un par un, un à plusieurs, plusieurs à plusieurs, …).

Application :

On souhaite ajouter au travail déjà réalisé des fonctionnalités pour mettre en place un système informatique de gestion des comptes courants bancaires.

Ce système offrira à ses utilisateurs les tâches suivantes :

- Gestion des clients (déjà développée)
- Gestion des comptes
- Gestion des transactions (versement, retrait, virement, …)

Travail à faire :

a) Utiliser une commande Artisan pour créer un Controller avec ressources nommé **ComptesController**.
b) Ajouter dans le fichier **routes/web.php** les Routes correspondants aux différentes méthodes du **ComptesController**.
c) Utiliser une commande Artisan pour créer un Model nommé **Compte**, avec sa Migration nommée **créer_compte_table**.
d) Créer dans un nouveau répertoire nommé **resources/views/comptes**, les Views correspondant à chacun des Routes.

2. Définir des relations

Les relations avec Eloquent sont définies comme des méthodes sur les Models.

Comme les Models d'Eloquent eux-mêmes, les relations servent aussi de puissants constructeurs de requêtes, la définition de relations en tant que méthodes fournit de puissantes capacités de chaînage et d'interrogation de méthodes.

2.1. Relation un-à-un

Par exemple, un **User** peut-être associé à un **Phone**. Pour définir cette relation, nous plaçons une méthode **phone()** dans le Model **User**.

```
class User extends Model
{
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

L'argument passé à la méthode **hasOne()** est le nom du Model associé. Une fois la relation définie, nous pouvons récupérer l'enregistrement associé en utilisant les propriétés dynamiques d'Eloquent. Les propriétés dynamiques vous permettent d'accéder aux méthodes relationnelles comme s'il s'agissait de propriétés définies dans le modèle lui-même:

```
$phone = User::find(1)->phone;
```

Eloquent déduit la clé étrangère de la relation en fonction du nom du Model. Dans ce cas, le Model **Phone** est automatiquement supposé avoir une clé étrangère **user_id**. Si vous souhaitez remplacer cette convention, vous pouvez passer un deuxième argument à la méthode **hasOne()**:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

En outre, Eloquent suppose que la clé étrangère doit avoir une valeur correspondant à la colonne **id** (ou le **$primaryKey** personnalisé) du parent. En d'autres termes, Eloquent recherchera la valeur de la colonne **id** du **User** dans la colonne **user_id** de l'enregistrement dans **Phone**. Si vous souhaitez que la relation utilise une valeur autre que **id**, vous pouvez passer un troisième argument à la méthode **hasOne()** en spécifiant votre clé personnalisée:

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

Maintenant, nous allons définir l'inverse de la relation en définissant une relation dans le Model du **Phone**, qui nous permettra d'accéder à **User** qui possède le téléphone.

```
class Phone extends Model
{
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

2.2. Relation un-à-plusieurs

Le type de relation un-à-plusieurs est utilisé pour définir des relations où un même Model possède un nombre quelconque d'autres Models. Par exemple, un article de blog peut avoir un nombre infini de commentaires.

```
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Maintenant, définissons la relation permettant à un commentaire d'accéder à son message parent. Pour définir l'inverse d'une relation **hasMany**, nous avons à définir une fonction dans le Model enfant qui appelle la méthode **belongsTo()**:

```
class Comment extends Model
{
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

3. Exercice d'application

a) Ajouter, dans chacun des Models **Client** et **Compte**, le code nécessaire à la définition de la relation entre les deux Models.

b) Ajouter, dans la Migration **create_comptes_table**, le code nécessaire à créer, dans la BD, la table **comptes** sachant que :
*comptes (**id**, codeBanq, codeGuichet, rib, cleRib, #titulaire, solde, devise)*
Avec des numéros aléatoires pour : *rib* (11 digits), *codeBanq* et *codeGuichet* (5 digits), *cleRib* (2 digits). Le champ **titulaire** est une clé étrangère qui fait référence à l'**id** du client titulaire du compte.

c) Ajouter, dans chacune des méthodes du **ComptesController**, le code nécessaire à assurer la fonctionnalité demandée par la méthode, en assignant les différents résultats aux différentes Views du compte.

d) Ajouter, dans chacune des Views du compte, le code nécessaire permettant de prendre en charge les données envoyées depuis le **ComptesController** et les afficher dans les pages.

e) Ajouter dans la barre de navigation les liens nécessaires à la navigation dans les nouvelles fonctionnalités de gestion des comptes.

f) Ajouter le code jugé nécessaire pour mettre en marche la gestion des transactions bancaires relatives à un compte courant : retrait, versement et virement.

NB : l'utilisation d'Ajax est un bonus noté.

Atelier 10 : Création d'une RESTful API

A la fin de cet atelier vous serez capables de :

- Comprendre l'utilité d'une API
- Mettre en place et tester une RESTful API

1. Introduction

Avec l'essor du développement mobile et des Frameworks JavaScript, l'utilisation des RESTful APIs est la meilleure option pour créer une interface unique entre vos données et votre client.

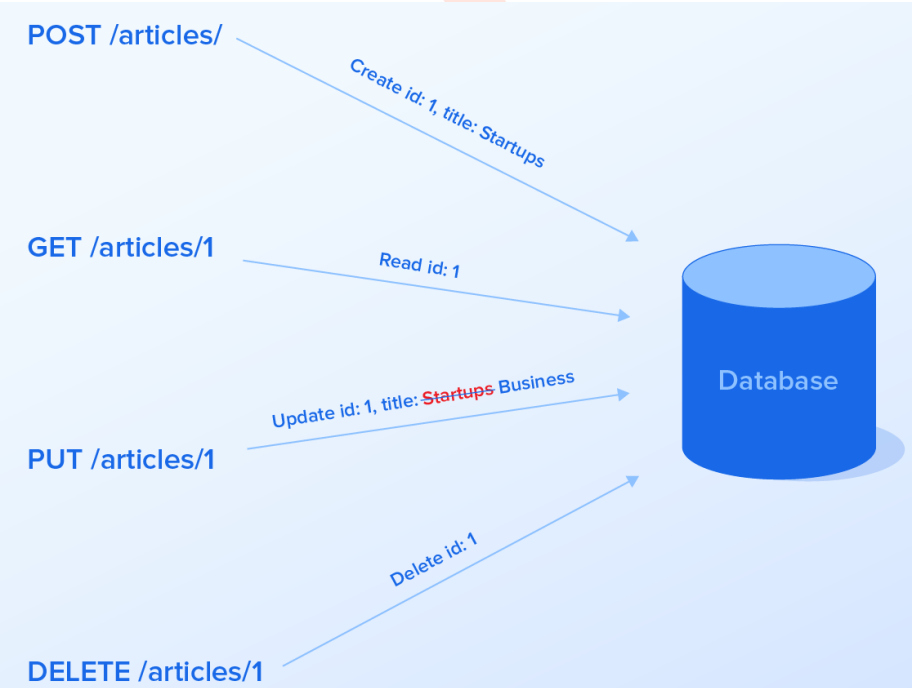
1.1. Définition

REST est synonyme de **REpresentational State Transfer** et est un style architectural pour la communication entre les applications (via Internet), qui repose sur un protocole sans état (généralement HTTP) pour l'interaction.

1.2. Les Verbes HTTP

Dans les API RESTful, nous utilisons les verbes HTTP comme actions, et les **endpoints** sont les ressources utilisées. Nous utiliserons les verbes HTTP pour leur signification sémantique:

- **GET**: récupérer des ressources
- **POST**: créer des ressources
- **PUT**: mettre à jour des ressources
- **DELETE**: supprimer des ressources



2. Mise en place du service Web

Activité :

a) Créer, dans le répertoire **App/http/Controllers**, un nouveau dossier nommé **Api**.
b) Utiliser une commande Artisan pour créer, dans le dossier **App/http/Controllers/Api**, un **ClientsController** avec ressources.

Pour commencer on va créer les différents Routes pour prendre en charge les différentes demandes http entrant à l'API. Pour ceci on modifiera le fichier **routes/api.php** et qui sera comme suit :

```
//lister tous les clients
Route::get('clients','Api\ClientsController@index');
//afficher un seul client
Route::get('client/{id}','Api\ClientsController@show');
//créer un nouveau client
Route::post('client','Api\ClientsController@store');
//modifier un client
Route::put('client/{id}','Api\ClientsController@update');
//supprimer un client
Route::delete('client/{id}','Api\ClientsController@destroy'
```

On remarque que les réponses des Routes qu'on vient de créer sont les méthodes de notre **ClientsController**. On peut remarquer aussi que les méthodes **create()** et **edit()** ont été supprimé du Controller puisqu'on en n'aura pas besoin.

L'étape suivante est de préparer les données en réponse dans les méthodes du **ClientsController** :

```
class ClientsController extends Controller
{
    public function index()
    {
        return Client::all();
    }
    public function store(Request $request)
    {
        return Client::create($request->all());
    }
```

```
public function show($id)
    {
        return Client::find($id);
    }
    public function update(Request $request, $id)
    {
        $client = Client::findOrFail($id);
        $client->update($request->all());
        return $client;
    }
    public function destroy($id)
    {
        $client = Client::findOrFail($id);
        $client->delete();
        return 204;
    }
}
```

Références

[1] B. Traversy, «Laravel from scratch,» Traversy Media, [En ligne]. Available: https://www.youtube.com/playlist?list=PLillGF-RfqbYhQsN5WMXy6VsDMKGadrJ-.

[2] Laracasts, «Laravel 5.4 From Scratch,» [En ligne]. Available: https://laracasts.com/series/laravel-from-scratch-2017.

[3] C. LaravelCollective, «Laravel Collective,» [En ligne]. Available: https://laravelcollective.com/.

[4] F. Zaninotto, «Faker,» GitHub, [En ligne]. Available: https://github.com/fzaninotto/Faker.

[5] T. Otwell, «Documentation officielle,» [En ligne]. Available: https://laravel.com/docs/5.4.

[6] A. Castelo, «Laravel API Tutorial: How to Build and Test a RESTful API,» [En ligne]. Available: https://www.toptal.com.