Laravel

introduction

Florian Rodriguez

Introduction

Qu'est-ce que Laravel ?

- Un **framework PHP** pour le développement web moderne.
- Un framework est comme une boite à outil à disposition
- Facilite l'implémentation de l'authentification, la sécurité, le routage, etc.

Avantages de Laravel

- Simplicité
- Documentation complète
- Communauté active.
- Architecture MVC
- Nombreux modules disponibles

Ressources

- Le plus important https://laravel.com/docs/master
- Tutos par Jeffrey Way le créateur de laracast: (en anglais) https://laracasts.com/series/laravel-from-scratch-2026
- Un cours laravel en français: https://laravel.sillo.org

Plan

Composer

Qu'est-ce que Composer?

- Composer est un **gestionnaire de dépendance** (Dependency Manager) pour PHP.
- Composer permet de **déclarer les librairies** (aussi appelées paquets) dont vous avez besoin dans votre projet puis de les installer et les mettre à jour facilement.
- Composer va aussi gérer toutes les dépendances des librairies que l'on veut utiliser.
- Composer utilise https://packagist.org/ comme dépôt principal de librairies.
- Composer utilise un fichier **"composer.json"** pour gérer les librairies d'un projet.

Un exemple

composer.json

```
{
    "name": "laravel/framework",
    "description": "The Laravel Framework.",
    "keywords": ["framework", "laravel"],
    "license": "MIT",
    "homepage": "https://laravel.com",
    "support": {
        "issues": "https://github.com/laravel/framework/issues",
        "source": "https://github.com/laravel/framework"
    },
    "authors": [
        {
            "name": "Taylor Otwell",
            "email": "taylor@laravel.com"
        }
    ],
    "require": {
        "php": "^8.3",
        "ext-ctype": "*",
        "ext-filter": "*",
        "ext-hash": "*",
        "ext-mbstring": "*",
        "ext-openssl": "*",
        "ext-session": "*",
        "ext-tokenizer": "*",
        "composer-runtime-api": "^2.2"
```

https://github.com/laravel/framework/blob/master/composer.json

Installation

- Télécharger composer https://getcomposer.org/download/
- Ajouter composer à votre PATH selon votre système.
  Ex de solution pour linux:

```
sudo mv composer.phar /usr/local/bin/composer
```

- Vous pouvez vous inspirer de cette aide.

Installation de Laravel

Liens vers la documentation:

https://laravel.com/docs/master#creating-a-laravel-project

Deux solutions d'installation

Lancement de l'application

Nouvelle méthode

- Se placer dans le répertoire du projet (`cd exemple-app`), puis:

```
composer run dev
```

Ancienne méthode

- Laravel inclut un outil pour la ligne de commande: `artisan`

  `artisan` offre, entre autre, un serveur web local.

  Pour l'utiliser, on se place dans le projet (`cd exemple-app`), puis:

```
composer run dev
```

On peut accéder ensuite à l'application avec l'url: http://localhost:8000

Configuration et environnement

- Le fichier `.env` permet de configurer l'application (database, etc)
- Les répertoires `bootstrap/cache` et `storage` doivent être accessible en écriture pour que Laravel puisse y créer des fichiers.

Repo git exemple du cours

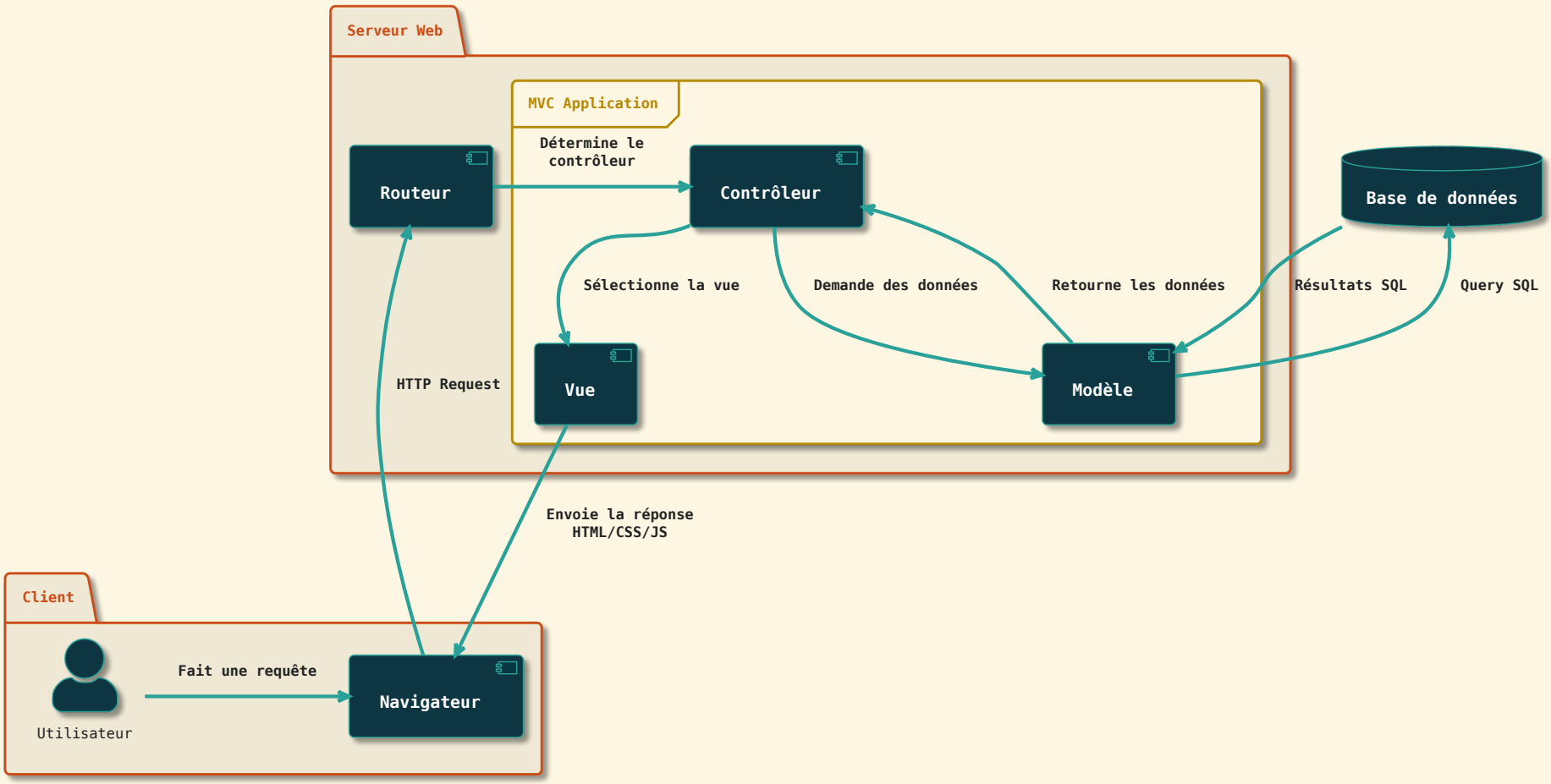
- Le repository de l'exemple du cours: https://github.com/MIASHS-UGA-PWS/example-cours-2026
- Si vous souhaitez utiliser l'exemple du cours *(ou récupérer un projet existant déjà sur git, c'est à dire sans l'installer avec composer)*, voici la marche à suivre:
  - On clone le repo (`git clone https://github.com/MIASHS-UGA-PWS/example-cours-2026`)
  - On initialise le projet et installe les dépendances (`composer run setup`)

```
git clone https://github.com/MIASHS-UGA-PWS/example-cours-2026
composer run setup
```

  - Cette commande remplace les commandes suivantes:
    - ~~on «créer» un `.env` (`cp .env.example .env`)~~
    - ~~on installe les dépendances du projet (`composer install`)~~
    - ~~on installe les dépendances front-end (`npm install && npm run build`)~~
    - ~~on genere une clé d'application (`php artisan key:generate`)~~
    - ~~on crée la base de donnée sqlite (`touch database/database.sqlite`)~~
    - ~~on lance les migrations (`php artisan migrate`)~~
- Cette opération est à réaliser une seule fois. Ensuite il suffira de faire un `git pull` pour récupérer les mises à jour du projet.
- On peut ensuite lancer le projet:

```
composer run dev
```

Rappel schéma MVC



Structure et routes

Structure des fichiers

- **Models** → `app/Models`
- **Controllers** → `app/Http/Controllers`
- **Views** → `resources/views`
- **Routes** → `routes/web.php`

  Les routes permettent de définir quelles actions seront exécutées en fonction des URLs demandées.
  En général, c'est un controller qui sera appelé.

Routage Basique (Routing)

Voyons comment fonctionne l'application de base.

Tout d'abord lançons l'application:

```
composer run dev
```

puis ouvrons http://localhost:8000

Que ce passe t'il?

1. La requête HTTP est envoyé et «réceptionner» par le framework.
2. Le framework va déterminer quoi faire avec la requête dans le fichier `routes/web`.

3. En fonction de l'url demandé (ici `/`),y l'action ne sera pas la même.

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

4. → Le fichier `resources/views/welcome.blade.php` sera retourné

- Test avec une route non écrite http://localhost:8000/welcome
- Ajout d'une nouvelle route

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

- Retour chaine

```
Route::get('/', function () {
    return "Hello World";
});
```

- Retour json

```
Route::get('/', function () {
    return ['foo' => 'bar'];
});
```

- Retour nouvelle vue

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

→ erreur

→ on créer la vue (`resources/views/test.blade.php`)

Données dans la requête

```php
Route::get('/', function () {
    $name = request('name');
    return $name;
});
```

http://localhost:8000/?name=Florian

Les Vues avec Blade

Blade: Le système de Templating

Qu'est-ce qu'un système de templating (En anglais: template processor, template engine ou template parser) ?

Son rôle est d'aider dans la lisibilité et la logique du projet en général. Au lieu d'écrire directement du PHP dans le code HTML, on va utiliser le langage du système de Templating (Pour Laravel, c'est Blade, mais il en existe d'autres: Smarty, Twig, etc.)

Par exemple, pour un menu qui doit être le même sur plusieurs page, on peut créer un fichier template menu.blade.php puis l'utiliser ensuite dans d'autre fichiers template avec : `@extends(menu)`

Similaire à l'utilisation de *include* avec PHP, mais dans ce cas, on sépare bien le langage utilisé pour les templates et les vues et le code PHP.

- Dans le template (`layout.blade.php`)

```
@yield('title','Titre par défaut')  // valeur par défaut
@yield('content')
```

- Dans la vue (`home.blade.php`)

```
@extends('layout')

{{-- On peut aussi déclarer la section en inline --}}
{{-- @section('title','Title - Home') --}}

@section('content')
<h1>Home</h1>
@endsection
```

- Route (`Routes/web.php`)

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

Passer des données à la vue

la base

Routes/web.php

```php
Route::get('/', function () {
    $tasks = [
        'Aller faire les courses',
        'Aller à la gym',
        'Dormir'
    ];
    return view('home',[
        'tasks' => $tasks
    ]);
});
```

home.blade.php

```
@extends('layout')

@section('title','<h1>Home</h1>')

@section('content')

<h1>Home</h1>

    <ul>
        <?php foreach ($tasks as $task ) : ?>
            <li><?= $task; ?></li>
        <?php endforeach; ?>
    </ul>

@endsection

```

Passer des données à la vue

en blade

routes/web.php

```php
Route::get('/', function () {
    $tasks = [
        'Aller faire les courses',
        'Aller à la gym',
        'Dormir'
    ];
    return view('home',[
        'tasks' => $tasks
    ]);
});
```

home.blade.php

```
@extends('layout')

@section('content')

<h1>Home</h1>

    <ul>
        @foreach ($tasks as $task )
            <li>{{ $task }}</li>
        @endforeach
    </ul>

@endsection
```

Passer des données à la vue

plus de variables

routes/web.php

```php
Route::get('/', function () {
    $tasks = [
        'Aller faire les courses',
        'Aller à la gym',
        'Dormir'
    ];
    return view('home',[
        'tasks' => $tasks,
        'test' => 'Mon test'
    ]);
});
```

home.blade.php

```
@extends('layout')

@section('content')

<h1>{{ $test }} Home</h1>

    <ul>
        @foreach ($tasks as $task )
            <li>{{ $task }}</li>
        @endforeach
    </ul>

@endsection

```

Passer des données à la vue

input en GET

routes/web.php

```php
Route::get('/', function () {
    $tasks = [
        'Aller faire les courses',
        'Aller à la gym',
        'Dormir'
    ];
    return view('home',[
        'tasks' => $tasks,
        'test' => request('title')
    ]);
});
```

home.blade.php

```php
@extends('layout')

@section('content')
  <h1>{{ $test }} Home</h1>
  <ul>
  @foreach ($tasks as $task )
     <li>{{ $task }}</li>
  @endforeach
  </ul>
@endsection

```

- http://localhost:8000/?title=Bonjour

Passer des données à la vue

échappement

Routes/web.php

```php
Route::get('/', function () {
    $tasks = [
        'Aller faire les courses',
        'Aller à la gym',
        'Dormir'
    ];
    return view('home',[
        'tasks' => $tasks,
        'test' => '<script>alert("coucou")</script>'
    ]);
});
```

home.blade.php

```php
@extends('layout')

@section('content')
    <h1>{{ $test }} Home</h1>
    <ul>
        @foreach ($tasks as $task )
            <li>{{ $task }}</li>
        @endforeach
    </ul>
@endsection

```

- `{{ $test }}` est un peu plus que `<?= $test; ?>`
- La variable est échappé avec *htmlspecialchars*
  → prévient les attaques xss
- On peut «forcer» avec `{!! $test !!}` **attention!!!!**

Passer des données à la vue

autres écritures

routes/web.php

```php
Route::get('/', function () {
    $tasks = [
        'Aller faire les courses',
        'Aller à la gym'
    ];
    return view('home',[
        'tasks' => $tasks,
        'test' => 'Mon test'
    ]);
    // ou
    return view('home')->withTasks($tasks)->withTest('Mon test with 1');
    // ou
    return view('home')->with([
            'tasks' => $tasks,
            'test' => 'Mon test with 2'
    ]);
});
```

Passer des données à la vue

Clean et recap layout

routes/web.php

```php
Route::get('/', function () {
    return view('home',[
        'title' => 'Home'
    ]);
});
```

home.blade.php

```php
@extends('layout')

@section('title')
  <h1>{{ $title }}</h1>
@endsection

@section('content')
  <p>Contenu de la page d'accueil</p>
@endsection
```

Les Controllers

Requêtes → Controllers

On va maintenant rediriger nos requêtes vers des controllers

routes/web.php

```php
use App\Http\Controllers\PagesController;
Route::get('/', [PagesController::class, 'home']);

/*
  Route::get('/', function () {
      return view('home',[
          'title' => 'Home'
      ]);
  });
*/
```

→ erreur car le controller n'existe pas.

→ On pourrait aller dans `app/Http/Controllers` et créer un controller mais…

Controller

L'outil artisan permet aussi la création de fichiers dans l'application!

```
php artisan make:controller PagesController
```

Le fichier est automatiquement créé dans `app/Http/Controllers` Ensuite, on peut ajouter la méthode home et inclure le code que l'on utilisait pour appeler la view:

app/Http/Controller/PagesController.php

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PagesController extends Controller
{
    public function home()
    {
        return view('home',[
            'title' => 'Home'
        ]);
    }
}
```

Ajouter des pages

- On va «router» de nouvelles url vers de nouvelles méthodes de notre controller `Routes/web.php`

```php
Route::get('/', [PagesController::class, 'home']);
Route::get('/about', [PagesController::class, 'about']);
Route::get('/contact', [PagesController::class, 'contact']);
```

- Puis on va ajouter de nouvelles méthode à notre controller `app/Http/Controllers/PagesController.php`

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class PagesController extends Controller
{
    public function home()
    {
        return view('home',[
            'title' => 'Home'
        ]);
    }
    public function about()
    {
        return view('about');
    }
    public function contact()
    {
        return view('contact');
    }
}

```

- On créer les vues `about.blade.php` et `contact.blade.php`

```blade
@extends('layout')

@section('title')
  <h1>{{ $title }}</h1>
@endsection

@section('content')
  <p>Contenu de la page about</p>
@endsection
```

- On ajoute le *title* dans les données qu'on passe à notre vue dans le controller

```php
public function about()
{
    return view('about',[
        'title' => 'About'
    ]);
}
public function contact()
{
    return view('contact',[
        'title' => 'Contact'
    ]);
}
```

  - On peut tester
    http://localhost:8000/about
    http://localhost:8000/contact

Recap

- Création d’un projet avec composer:
  (ou l'outil laravel `laravel new project`)

```
composer create-project laravel/laravel example-app
```

- Pour définir les **routes** de l’application (web), il faut éditer le fichier `routes/web.php`
- Les **contrôleurs** sont situés dans le répertoire `app/Http/Controllers`
- Ils peuvent être **générés** automatiquement avec la commande:

```
php artisan make:controller MonController
```

- Les **vues** sont situées dans le répertoire `resources/views`
- On peut **passer de la données** au vue en second paramètre de la fonction view

```php
$monTableauData = ['attr' => 'value', 'attr2' => 'value2'];
return view('mavue', $monTableauData);
```

- On utilise **blade** dans les vues et on peut afficher les données avec par exemple `{{$attr}}`
- Les **assets** de l'application dans les répertoire resources/js, resources/sass, etc

Bases de données et Migration

Installation / Configuration

- On commence par la création d'une base de données.
- On configure ensuite les accès dans le fichier `.env`

Pour sqlite

```
DB_CONNECTION=sqlite
DB_DATABASE=/ABSOLUTE_PATH/database/database.sqlite
```

Pour mysql

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

- On lance ensuite en ligne de commande la commande suivante:

```
php artisan migrate
```

Avantages

- Avantages de `php artisan migrate`:
  - On définit nos schémas de base et nos modifications en classe, **pas besoin d'écrire du SQL**
  - Si on est en équipe, on clone, on migrate et on a instantanément le projet qui tourne
  - On a un pseudo «versionning» de la DB
  - On peut rollback (revenir en arrière) avec `php artisan migrate:rollback`

Exemple

- Par exemple, si on veut changer le champ name dans users…?
  - On édite `database/migrations/create_users_table.php`
  - On change par exemple `name` en `username`
  - On lance un rollback:

```
php artisan migrate:rollback
```

  - On relance ensuite:

```
php artisan migrate
```

  - On peut voir dans la base que ça a changé.

Migrate:fresh

On peut facilement réinitialiser une base de données si on veut “recommencer”

On lance la commande:

```
php artisan migrate:fresh
```

Cette commande **efface toutes les tables** puis relance la migration

!!! Attention… à utiliser avec parcimonie et surtout pas en production !!!

Création de migrations

Pour créer des nouvelles migrations, on peut utiliser `artisan`

- Description de la commande:

```
php artisan help make:migration
```

- Création d'une migration:

```
php artisan make:migration create_projects_table
```

- La commande précédente crée le fichier: `database/migrations/create_projects_table.php` :

database/migrations/create_projects_table.php

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('projects', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('projects');
    }
```

- create → Schema::create
- up → migrate
- down → rollback
- garder le timestamps c’est une bonne idée

database/migrations/create_projects_table.php

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('projects', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->text('description');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('projects');
    }
};
```

- Pour exécuter notre nouvelle migration:

```
php artisan migrate
```

  php artisan migrate

- On peut vérifier dans notre base de données que la table a bien été créée.
- On peut revenir en arrière:

```
php artisan migrate:rollback
```

- La table est supprimée.
- La méthode down() est appelé lors du rollback. On peut donc adapter les migration en éditant les méthodes **up** et **down**
- Si erreur on peut `migrate:fresh`

Récap

- Après création d'une base de données, on configure les accès dans le fichier `.env`
- On peut lancer des migrations de base de données avec l'outil artisan: `php artisan migrate`
- On peut aussi créer de nouvelles migrations et revenir en arrière facilement
- On peut réinitialiser sa base de données avec `php artisan migrate:fresh`
- Ce sont les méthodes *up* et *down* d'une migration qui sont appelées lors des *migrate* et des *rollback*
- On peut ensuite facilement modifier la méthode up pour configurer la table de la base de données sans avoir à écrire de SQL

Les Modèles et Eloquent ORM

BDD + Modèles = Eloquent ORM

- Comment accède-t-on à la table?
- Comment insère-t-on des données?
- Comment interroge-t-on la base?

Eloquent

Eloquent est l' **ORM** (Object Relational Mapping) de Laravel.

Un ORM permet à un modèle d'interagir facilement avec la base de données (sa table, mais aussi ses relations)

- Pour créer un nouveau modèle:

```
php artisan make:model Project
```

- Le modèle est créé dans le répertoire: `app/Models/Project.php`
- On peut maintenant utiliser toutes les fonctionnalités de Eloquent: https://laravel.com/docs/master/eloquent

Tinker: Piloter laravel en ligne de commande

Tinker est un outil artisan qui permet d'interagir avec la base de données depuis la ligne de commande. En réalité, on peut accéder aux fonctionnalités de Laravel, depuis la ligne de commande, pour effectuer des tests, essais, vérifications, création, etc.
→ On le lance avec la commande:

```
php artisan tinker
```

```
php artisan tinker
Psy Shell v0.12.0 (PHP 8.3.2 — cli) by Justin Hileman
> 2+2
= 4

> App\Models\Project::all();
= Illuminate\Database\Eloquent\Collection {#5761
    all: [],
  }

> use App\Models\Project;
> Project::all();
= Illuminate\Database\Eloquent\Collection {#5710
    all: [],
  }

> Project::first();
= null

> Project::latest()->first();
= null

> $project = new Project;
= App\Models\Project {#4994}

> $project->title = 'Mon premier projet';
```

→ On peut vérifier que l’objet a bien été créé dans la base de données.

```
php artisan tinker
Psy Shell v0.12.0 (PHP 8.3.2 — cli) by Justin Hileman
> Project::first();
= App\Models\Project {#4995
    id: 1,
    title: "Mon premier projet",
    description: "Lorem ipsum",
    created_at: "2024-02-05 22:58:45",
    updated_at: "2024-02-05 22:58:45",
  }

> Project::first()->title;
= "Mon premier projet"

> Project::first()->description;
= "Lorem ipsum"

> Project::all();
= Illuminate\Database\Eloquent\Collection {#5985
    all: [
      App\Models\Project {#5975
        id: 1,
        title: "Mon premier projet",
        description: "Lorem ipsum",
        created_at: "2024-02-05 22:58:45",
        updated_at: "2024-02-05 22:58:45"
```

→ On créer un nouveau projet

```
> $project = new Project;
= App\Models\Project {#5829}

> $project->title = 'Mon projet 2';
= "Mon projet 2"

> $project->description = 'Lorem ipsum 2';
= "Lorem ipsum 2"

>  $project->save();
= true

> Project::all();
= Illuminate\Database\Eloquent\Collection {#5984
    all: [
      App\Models\Project {#5975
        id: 1,
        title: "Mon premier projet",
        description: "Lorem ipsum",
        created_at: "2024-02-05 22:58:45",
        updated_at: "2024-02-05 22:58:45",
      },
      App\Models\Project {#5009
        id: 2,
        title: "Mon projet 2",
        description: "Lorem ipsum 2"
```

```
Project::all();
Project::all()[0];
Project::all()[1];
Project::all()[1]->title;
Project::all()->map->title;
```

Recap

- **Eloquent** est une classe permettant aux modèles d'interagir facilement avec la base de données. Nous reviendrons plus en détail sur Eloquent par la suite.
- **Tinker** est un outil permettant d'utiliser son application Laravel depuis la ligne de commande. On peut facilement appeler les classes et méthodes de l'application.

Un exemple complet MVC

Nous allons faire un exemple complet MVC d'une page affichant des "projets" stocker en base de données.

Il nous faut:

- Créer une nouvelle **route** pour traiter l'url `/project`
- Un **controller** qui va gérer les actions concernant les projets
- Une méthode de se controller qui va:
  - récupérer les projets grâce au **model**
  - retourner une vue et lui passer de la donnée (les projets)
- Une **vue** qui affiche les projets

Création de la route et du controller

- On définit les routes de l'application dans le fichier `routes/web.php`

```php
use App\Http\Controllers\ProjectController;
Route::get('/project', [ProjectController::class, 'index']);
```

- On créer un nouveau controller avec la commande:

```
php artisan make:controller ProjectController;
```

→ Le fichier est créé dans: `app/Http/Controller/ProjectController.php`

Appel de la vue

- On ajoute une méthode index

  `app/Http/Controller/ProjectController.php`

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class ProjectController extends Controller
{
    //
    public function index()
    {
        return view('project.index');
    }
}

```

- On créer un fichier vue dans le répertoire `resources/views` : `project/index.blade.php`
  → On voit qu'on peut ranger dans des sous dossier et appeler le fichier vue avec `view('folder.file')`
- On lance l'application avec: `composer run dev` et on peut voir la page dans le navigateur

Appel du model

- On va maintenant coder la méthode `index` du contrôleur `ProjectsController`
- On va récupérer les projets:

```php
$projects = \App\Models\Project::all();
```

ou

```php
use App\Models\Project;
$projects = Project::all();
```

Appel du model

ProjectsController

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Project;

class ProjectController extends Controller
{
    //
    public function index()
    {
        $projects = Project::all();
        return $projects;
    }
}
```

Résultat

```json
[
    {
        "id": 1,
        "title": "Mon premier projet",
        "description": "Lorem ipsum",
        "created_at": "2024-02-05T22:58:45.000000Z",
        "updated_at": "2024-02-05T22:58:45.000000Z"
    },
    {
        "id": 2,
        "title": "Mon projet 2",
        "description": "Lorem ipsum 2",
        "created_at": "2024-02-05T23:04:15.000000Z",
        "updated_at": "2024-02-05T23:04:15.000000Z"
    }
]
```

→ return json

On passe les données à la vue

ProjectsController

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Project;

class ProjectController extends Controller
{
    //
    public function index()
    {
        $projects = Project::all();
        return view('project.index', ['projects' => $projects]);
    }
}
```

ou

```php
return view('project.index', compact('projects'));
```

voir https://www.php.net/manual/en/function.compact.php

Affichage des projets dans la vue

- Dans la vue `resources/views/projects/index.blade.php` on itère sur les projets:

```
<h1>Projects</h1>
<ul>
    @foreach ($projects as $project)
    <li>{{ $project->title }}</li>
    @endforeach
</ul>
```