**1)Laravel Installation**

Composer create-project laravel/laravel nomApplication –prefer-dist

**2)Actionner le server**

Cd nomApplication

Php artisan serve

**3)Routing**

Attention ceci change selon la version. La structure des fichiers change selon les versions utilisées

nomApplication/app/http/routes.php

->Créer des routes

Route ::*get*('/about',**function**()**{  
 Return** 'Le contenu est ici';  
**}**);

->Passer une variable dans la route

Route::*get*('about/{theSubject}', **function**($theSubject)   
**{  
 return** $theSubject.' content goes here.';  
**}**);  
  
Route::*get*('/about/classes/{theSubject}', **function**($theSubject)  
**{  
 return** "Content about **{**$theSubject**}** classes goes here.";  
**}**);

Valeur par défaut:

Route::*get*('about/{theSubject}', **function**($theSubject=”default”)   
**{  
 return** $theSubject.' content goes here.';  
**}**);

Controller le type ou la morphologie du paramètre avec un regex :

// Introduire un paramètre  
Route::*get*('/random/{id}',**function**($id) **{  
 return** "quelque chose $id";  
**}**)->where('id','\d\*');

Rendre un paramètre optionnel par le biais de ?:

// Introduire un paramètre  
Route::*get*('/random/{id?}',**function**($id="") **{  
 return** "quelque chose $id";  
**}**);

Créer un alias:

Route::*get*('/newRoute',**array**(  
 'as'=>'broom',  
 **function**()**{  
 return** 'Broom broom';  
 **}**));

**Rediriger**:

Route::*get*('/testRedirection',**function**()**{** //à une route  
 **return** Redirect::*to*('/route');  
 //à un alias  
 **return** Redirect::*route*('alias');  
**}**);

Redirect avec un parameter:

//Comment on fait un redirect?  
Route::*get*('/where',**function**()**{  
 return** Redirect::*route*('rand',**array**('id'=>1));  
**}**);

Je n’ai réussi qu’à le faire marcher avec un alias et la method Redirect::route() ;

Depuis le contrôlleur:

**public function** random()  
**{  
 return** redirect()->route('rand',['id' => 1]);  
**}**

Route::*get*('about/{theSubject}', **function**($theSubject)   
**{  
 return** $theSubject.' content goes here.';  
**}**);

**4) Reenvoyer un html view**

Les templates sont stockés dans le dossier

Resources/views/

Route::*get*('/', **function**()  
**{  
 return** View::*make*('hello');  
**}**);

Marche aussi avec view() simplement

Route::*get*('/form',**function**()**{** //form.blade.php  
 **return** view('form');  
**}**);

5) Les resources css peuvent être stockés dans le dossier

Public si le dossier css ou javacript n’existe pas créez les dans public

Essayer avec <link rel=’stylesheet’ href=’< ?php echo asset(‘css/main.css’) ; ?>’ />

Toutes les commandes php marchent ainsi que leurs fonctions

templating

<p>  
 **{{** $theLocation **}}** on **{{** date('M d, Y') **}}**</p>  
**@if(**$weather == 'sunny'**)** <p>It's a **{{**$weather**}}** day.</p>  
**@else** <p>No forecast available</p>  
**@endif**<p>Don't miss:</p>  
<ul style="text-align:left">  
**@foreach(**$theLandmarks **as** $landmark**)  
 @unless(**$landmark == 'Central Park'**)** <li>**{{**$landmark**}}**</li>  
 **@endunless  
@endforeach**</ul>

Difféncier include de extends dans blade

Le layout.blade.php

<html>  
 <head>  
 <title>Mon Forum - **@yield(**'title'**)**</title>  
 </head>  
 <body>  
 **@section(**'sidebar'**)** Menu gauche  
 **@show** <div class="container">  
 **@yield(**'content'**)** </div>  
 **@include(**'footer'**)** </body>  
</html>

Utilisation du layout.blade.php

**@extends(**'layout'**)  
  
@section(**'title'**)** Ceci est mon titre  
**@endsection  
  
@section(**'sidebar'**)  
 @parent** Ceci est ma sideBar  
**@endsection  
  
@section(**'content'**)** Ceci est mon contenu  
**@endsection**

Footer.blade.php

**<div>** Je suis un footer  
**</div>**

**6)Création de base de données**

Créer une base de données manuellement avec phpMyAdmin. On peut la créer aussi via les migrations mais je ne pense pas que ce soit nécessaire… mais je vous donne un lien si quelqu’un peut être intéressé : <http://laravel.io/forum/09-13-2014-create-new-database-and-tables-on-the-fly>

Changer les identifiants dans le fichier qui sont les var par défaut s’il n y a pas de fichier .env

nomApplication/app/config/databse.php

.env qui définie les variables locales, en théorie il ne faut que modifier le fichier .env si celui est présent.

Schema::*create*('art',**function**($newTable)**{** $newTable->increments('id');  
 $newTable->string('artist');  
 $newTable->string('title',500);  
 $newTable->text('description');  
 $newTable->date('created');  
 $newTable->date('exhibition\_date');  
 $newTable->timestamps();  
**}**);

//update  
Schema::*table*('art',**function**($newTable)**{** $newTable->boolean('alumni');  
 $newTable->dropColumn('exhibition\_date');  
**}**);

**7) Migrations**

Les migrations permettent de garder un suivi des changements de la base de données.

Générer des fichiers de migration

Dans le dossier database/migrations

>php artisan make :migration create\_art

Préparer l’objet de migration

*/\*\*  
 \* Run the migrations.  
 \*  
 \** ***@return*** *void  
 \*/***public function** up()  
**{** Schema::*create*('art2',**function**($newTable)**{** $newTable->increments('id');  
 $newTable->string('artist');  
 $newTable->string('title',500);  
 $newTable->text('description');  
 $newTable->date('created');  
 $newTable->date('exhibition\_date');  
 $newTable->timestamps();

**$newTable->engine = "InnoDB";**

**}**);  
**}***/\*\*  
 \* Reverse the migrations.  
 \*  
 \** ***@return*** *void  
 \*/***public function** down()  
**{** Schema::*drop*('art2');  
**}**

Ensuite lancez la ligne de commande

>php artisan migrate :install

>php artisan migrate

Pour défaire la migration migrate :rollback

>php artisan migrate :rollback

**8) Générer un model avec Laravel : Ce qui nous permettra de faire des requêtes en base de données. Notez que nous sommes aidés par l’ORM**

Les modèles sont des classes qui permettent à l’ORM de reconnaître les entités sur quoi elle est sensée faire des opérations. Elle représente un tableau en base de données (du moins pour des structures simples).

> php artisan make :model Painting

Une fois que le fichier est généré dans firstapp/app/

On peut appeler l’objet pour insérer des informations

A partir du routeur

$painting = **new** Painting();  
$painting->title = 'Joconde';  
$painting->artist = 'Mona Lisa';  
$painting->year=2014;  
$painting->save();

Pour appeler des données en provenance de la base de données :

Note : TOUTE CECI SE PASSE VIA L’ORM ELOQUENT

Notez que implicitement on cherche par id

$painting = Painting::*find*(1);  
**return** $painting->title;

Pour updater l’instance:

$painting = Painting::*find*(1);  
$painting->title='Do No Wrong - Just Do Right';  
$painting->save();  
**return** $painting->title;

Pour effacer utiliser “delete”

$painting = Painting::*find*(1);   
$painting->delete();  
**return** “deleted”;

**10) Seed the database**

Remplir une base de données pour faire du testing est commun il faut voir ce que nous pouvons faire avec une librarie qui s’appel faker. Une librarie gui génère des fausses données.

Aller dans le dosser database>seeds

Rajouter le nom du fichier faisant l’opération de seeding dans DatabaseSeeders

$this->call('PaintingsTableSeeder');

* Console

Composer require fzaninotto/faker

In the seeder file, for exemple PaintingsTableSeeder (fichier créé à la main)

**use** App\Painting;

**use** Illuminate\Database\Seeder;  
  
**Class** PaintingsTableSeeder **extends** Seeder  
**{  
 public function** run()  
 **{** $faker = Faker\Factory::*create*();  
  
 Painting::*truncate*();  
  
 **for**($i=0; $i<10;$i++)  
 **{** $painting = Painting::*create*(**array**(  
 'title'=>$faker->realText(rand(20,40)),  
 'author'=>$faker->name,  
 'description'=>$faker->realText(100)  
 ));  
 **}  
 }  
}**

ligne de commande pour lancer les seeds:

>>php artisan db:seed

Nous avons rencontré un problème lors de la génération de seeds dès lors que nous avons des clés étrangères comme contraintes dans un tableau. Par exemple nous avons créer la table comments avec une clé étrangère.

**public function** up()  
**{** Schema::*create*('comments',**function**($newTable)**{** $newTable->increments('id');  
 $newTable->integer('painting\_id')->length(10)->unsigned();  
 $newTable->integer('user\_id');  
 $newTable->text('comment');  
 $newTable->date('created')->default(Carbon::*now*());  
 $newTable->timestamps();  
 $newTable->engine = "InnoDB";  
 **}**);  
  
 Schema::*table*('comments',**function**($table)**{** $table->foreign('painting\_id')->references('id')->on('paintings')->onDelete('cascade');  
**}**);  
**}**

Notez bien que j’aurai pu faire ça pour l’id user. Observez aussi que j’ai rajouté $newTable->engine= »InnoDB » ; dans la table painting ET la table comment pour pouvoir faire ce lien. Enfin en cascade lie les deux entités de manière à si un painting est effacé les commentaires qui y sont associés le seront aussi.

Nous avons déjà vu en SQL ce qu’est une contrainte clé. Pour faire les seeds nous allons devoir désactiver le checking de ces clés. Dans le fichier database/seeds/DatabaseSeeder.php

DB::*statement*('SET FOREIGN\_KEY\_CHECKS=0');  
  
$this->call(PaintingsTableSeeder::*class*);  
$this->call(CommentsTableSeeder::*class*);  
  
DB::*statement*('SET FOREIGN\_KEY\_CHECKS=1');

**11) Créer un controlleur**

Peut être créé par le biais de

>> php artisan make :controller [Name]Controller

**12) Forms**

A)Création d’un formulaire sign up et redirection a une autre page “merci beaucoup”

Il faut un setup au préalable :

Rajouter à composer json

"require": {

"laravelcollective/html": "5.\*"

}

Alors composer update

Ensuite,

Rajouter dans config/app.php

'providers' => [

Collective\Html\HtmlServiceProvider::*class*,

],

'aliases' => [

'Form' => Collective\Html\FormFacade::*class*,  
'Html' => Collective\Html\HtmlFacade::*class*,

],

Ça depend de la version…

Rajouter encore dans config/app.php

'Input' => Illuminate\Support\Facades\Input::*class*,

Tapper dans le template

**{{** Form::*open*(**array**('url'=>'thanks')) **}}  
 {{**Form::*label*('email','Email Address')**}}  
 {{**Form::*text*('email')**}}  
 {{**Form::*label*('os','Operating System')**}}  
 {{**Form::*select*('os',**array**(  
 'linux'=>'Linux',  
 'mac'=>'MAC OS X',  
 'windows' => 'Windows'  
 )) **}}  
  
 {{**Form::*label*('comment','Comments')**}}  
 {{**Form::*textarea*('comment','',**array**('placeholder'=>'What are you interests?'))**}}  
  
 {{**Form::*checkbox*('agree','yes',**false**)**}}  
 {{**Form::*label*('agree','I agree to your terms of service')**}}  
 {{**Form::*submit*('Sign up')**}}  
{{** Form::*close*() **}}**

Aller au controlleur

Route::*get*('/signup',**function**()**{  
 return** View::*make*('signup');  
**}**);  
  
Route::*post*('/thanks',**function**()**{** $theEmail = Input::*get*('email');  
 **return** View::*make*('thanks')->with('theEmail',$theEmail);  
**}**);

B) Exemple avec gestion des erreurs : On peut également ajouter le handling d’erreur dans le formulaire et dans le service qui servira à gérer la soumission de celui-ci. Exemple :

**@if (**count($errors) > 0**)** <div class="alert alert-danger">  
 <ul>  
 **@foreach (**$errors->all() **as** $error**)** <li>**{{** $error **}}**</li>  
 **@endforeach** </ul>  
 </div>  
**@endif  
  
{{** Form::*open*(**array**('route'=>'newPaintingService')) **}}** <div class="form-groups">  
 **{{** Form::*label*('email','Email Address') **}}  
 {{** Form::*text*('email','',['class'=>'form-control']) **}}  
 {!!** $errors->first('email',"<div class='alert alert-danger'><strong>:message</strong></div>") **!!}** </div>  
  
 **{{** Form::*label*('os','Operating System') **}}  
 {{** Form::*select*('os',**array**(  
 'linux'=>'Linux',  
 'mac'=>'MAC OS X',  
 'windows'=>'Windows 10'  
 )) **}}  
  
 {{** Form::*label*('comment','Comment') **}}  
 {{** Form::*textarea*('comment','',  
 **array**(  
 'placeholder'=>'Quels sont tes hobbies?',  
 'class'=>'form-control'  
 )) **}}  
 {!!** $errors->first('comment',"<div class='alert alert-danger'><strong>:message</strong>") **!!}  
  
 {{** Form::*button*('submit',  
 **array**(  
 'type'=>'submit',  
 'class'=>'btn btn-primary'  
 )  
 ) **}}  
{{** Form::*close*() **}}**

Notez bien la presence des variable error qui sont en fait les erreurs stockés via le service que voici :

**public function** serviceNewPainting(Request $request)  
**{** $this->validate($request,[  
 'email'=>'required|max:255',  
 'comment'=>'required',  
 ]);  
  
 //save  
**}**

Remarques: il faut rajouter l’objet request qui gère les soumission de formulaire. N’oubliez pas le use correspondant :

**use** Illuminate\Http\Request;

Vous pouvez voir les contraintes qu’on applique à chaque champ. À vous de voir ce que vous voulez vérifier.

B) Si vous voulez désemcombrer votre contrôlleur des conditions de checking d’un formulaire. allez à la ligne de commande et tappez :

>> php artisan make :request PaintingRequest

Un fichier sera généré dans le dossier App\Http\Requests au quel vous pourrez rajouter les conditions que vous souhaitez :

**namespace** App\Http\Requests;  
  
**use** Illuminate\Foundation\Http\FormRequest;  
  
**class** Paintings **extends** FormRequest  
**{** */\*\*  
 \* Determine if the user is authorized to make this request.  
 \*  
 \** ***@return*** *bool  
 \*/* **public function** authorize()  
 **{  
 return true**;  
 **}** */\*\*  
 \* Get the validation rules that apply to the request.  
 \*  
 \** ***@return*** *array  
 \*/* **public function** rules()  
 **{  
 return** [  
 'title'=>'required|min:20|max:255',  
 ];  
 **}  
}**

change le authorize à true et rajouter vos règles dans rules.

Maintenant depuis le contrôleur

**public function** store(PaintingRequest $request)  
**{** $p = $request->all();  
 $p['user\_id'] = Auth::*id*();  
 $post = Post::*create*($p);  
 **return** redirect(route('post.show',$post->id));  
**}**

et n’oubliez pas l’include correspondant:

**use** App\Http\Requests\PaintingRequest;

**13) Créer un repositoire**

"require": {

"bosnadev/repositories": "0.\*"

}

Ou >>composer require "bosnadev/repositories: 0.\*"

Etape 2

**Create repository**

**Example**

**<?php  
  
namespace** App\Repositories;  
  
**use** Bosnadev\Repositories\Contracts\RepositoryInterface;  
**use** Bosnadev\Repositories\Eloquent\Repository;  
  
**class** PaintingsRepository **extends** Repository **{  
  
 public function** model() **{  
 return** 'App\Painting';  
 **}  
}**

**Update the Model**

**<?php  
  
namespace** App;  
  
**use** Illuminate\Database\Eloquent\Model;  
  
**class** Painting **extends** Model  
**{  
 protected** $primaryKey='id';  
 **protected** $table='paintings';  
  
  
**}**

**And finally prepare the Controller**

**<?php  
  
namespace** App\Http\Controllers;  
  
**use** Illuminate\Http\Request;  
  
**use** App\Http\Requests;  
// ad Repo with alias  
**use** App\Repositories\PaintingsRepository **as** Painting;  
  
**class** HomeController **extends** Controller  
**{** //add var  
 **private** $painting;  
 //prepare the repo to be used in the contructor  
 **public function** \_\_construct(Painting $painting)  
 **{** $this->painting=$painting;  
 **}  
  
 public function** myMethod()  
 **{** $p = $this->painting->all();  
  
 dd($p);   
 **}  
}**

**14) Créer un système d’authentication**

**>>**php artisan make:auth

**14) Créer une page d’erreur 404. (Au cas ou l’URL demandée n’existe pas)**

Dans le fichier App/Exceptions/Handler.php updater la méthode

**public function** render($request, Exception $exception)  
**{  
 return parent**::*render*($request, $exception);  
**}**

à

**public function** render($request, Exception $exception)  
**{  
 if**($exception **instanceof** NotFoundHttpException)  
 **{  
 return** response()->view('missing', [], 404);  
 **}  
 return parent**::*render*($request, $exception);  
**}**

et n’oubliez pas le use correspondant

**use** Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

**Évidemment complétez la vue.**

**15) Les query en Base de données :**

**L’intéraction avec la base de données avec Laravel 5 se passe de deux façon. La première via L’ORM et la deuxième via le QueryBuilder**

1. VIA L’ORM : Eloquent nous permet d’effectuer des opérations sans devoir taper des requêtes SQL. L’ORM est en fait une panoplie d’objets qui parlent avec la base de données et par conséquent nous n’auront pas à le faire

L’ORM nous permet de faire des opérations CRUD (Create, read, update and delete) comme nous avons vu précédemment. Je les reposte :

Opération create :

$painting = **new** Painting();  
$painting->title = 'Joconde';  
$painting->artist = 'Mona Lisa';  
$painting->year=2014;  
$painting->save();

Opération read et update:

$painting = Painting::*find*(1); //implicitement cherche pas id  
$painting->title='Do No Wrong - Just Do Right';  
$painting->save();  
**return** $painting->title;

Operation delete:

$painting = Painting::*find*(1); //implicitement cherche pas id  
$painting->delete();

L’ORM comporte d’autre filter de recherché par exemple pour rechercher par d’autres champs :

**return** User::*where*('name',$username);

Attention : Il est important d’observer que précédemment avec

$painting = Painting::*find*(1); //implicitement cherche pas id

On pouvait accéder au attributs par

Echo $user->name;

Par contre avec User::where et les autres méthodes qui suivent ceci n’est plus vrai car cela retournera une collection de models (Un array mais formaté objet) :

Pour y accéder vous devez

dd($user->first()); ou dd($user->last());

ou encore

dd($user->get()); ou dd($user->get()[0]); //Si plusieurs

Ensuite

dd($user->get()[0]->name);

Vous pouvez changer le type des données retourner avec ->toArray() ;

Au lieu d’un objet vous aurez un tableau.

**return** User::*where*('name',$username)->toArray();

Eloquent permet de chercher une batterie d’id

**return** User::*find*([1,2,3]);

nous rend les instances pour les users avec id 1 2 et 3.

Il y a bien d’autres méthodes mais je ne vais pas toutes les faire elles sont disponible dans la doc officielle <https://laravel.com/docs/5.3/eloquent> (version 5.3) mais laplupart marche pour la 5.2 mais après votre formation il est important de continuer à se documenter car ce framework en particulier connait pas mal de changements.

Une autre méthode utile est

firstOrCreate qui permet de localiser une ligne en base de données et si cette ligne n’existe pas une nouvelle insertion sera créée

**return** User::*firstOrCreate*('name',$username);

Précédemment nous avons vu que pour faire un delete d’un model il nous faut au préalable avoir le model instantié via la methode find($id). Avec la méthode destroy ceci n’est plus nécessaire :

User::*destroy*(1);

1. Le QueryBuilder

L’ORM est intelligent mais parfois pas assez et en plus il pompe beaucoup de ressources selon l’opération que l’on fait (à vérifier cependant). Pour cela il y a une alternative à cela… le query Builder. C’est-à-dire le constructeur de requête.

**return** DB::*table*('posts')  
 ->join('users', 'users.id','=','posts.user\_id')  
 ->where('posts.id',$postid)  
 ->select('posts.\*','users.name','users.email')  
 ->orderBy('id','DESC')  
 ->get();

La syntaxe commence par un DB::table plus le nom de la table.

Le join est pour joindre les table, le where pour faire un where(lol c’est assez explicatif si vous avez bien étudié les requêtes SQL). Le select peut venir après, ce qui change des requêtes traditionnelles en SQL. Le get de la fin signifie au queryBuilder d’obtenir le resultat.

Le query builder a une multitude de fonctions qui permettent de simplifier la manipulation des données, par exemple si on a juste besoin d’un champ on peut tapper :

DB::*table*('users')->name('username','apocalypse')->value('email');

Si on ne veut que la première ligne d’une requête comportant plusieurs lignes :

DB::*table*('users')->first();

Si on veut un tableau avec toutes les valeurs d’une seule colonne :

DB::*table*('users')->pluck('username');

Opération de comptage ou encore de maximization

DB::*table*('users')->count();

DB::*table*('commandes')->max('prix');

Parfois ces méthodes ne sont pas suffisantes et il faut combiner le query builder avec des expressions dites crues (SQL raw). Observez la ligne select

**return** DB::*table*('posts')  
 ->join('users','users.id','=','posts.user\_id')  
 ->leftJoin('comments','comments.post\_id','=','posts.id')  
 ->groupBy('posts.id')  
 ->orderBy('posts.id','DESC')  
 ->select(DB::*raw*('count(comments.id) AS nb\_comment,posts.\*,users.name AS username'))  
 ->get();

Enfin le query builder offre beaucoup de possibilités. Pour plus de détails regardez la doc

<https://laravel.com/docs/5.3/queries>

**Astuce : Visualiser ses requêtes en SQL cru en entier.** Parfois on n’est pas vraiment sûr avec l’ORM et le Querybuilder de la requête SQL qui est réellement passée. Pour les visualiser avant la/les requêtes :

DB::*enableQueryLog*();

Après la requête :

dd(DB::*getQueryLog*());

Si vous les placer stratégiquement vous pouvez vois absolument toutes les requêtes ou encore seulement en voir une selon vos besoins.

**Astuce : Ajouter un barre de debuggage**

Tappez la commande composer suivante :

>> composer require "barryvdh/laravel-debugbar": "^2.2"

Rajouter dans config/app.php dans le tableau providers.

Barryvdh\Debugbar\ServiceProvider::*class*,

Maintenant rafraichissez n’importe qu’elle page comportant des balises <html> et regarder en bas du navigateur. Cette barre comporte des information comme le nombre de requêtes et leurs syntaxe,les layout la route, le conrôlleur, les exceptions…

**Astuce structure du Controlleur**

Jusqu’à maintenant on a vu que nous pouvions organiser les contrôleurs comme bon nous semble et malgré ce que je vais vous donner maintenant cela reste vrai. Néanmoins Laravel a organisé un standard bien précis de comme les opération CRUD doivent être gérée et leur structure ce qui maximize la production dans bien des cas.

Concrèment si on tappe la commande :

>> php artisan controller:make movieController –resource

Ceci créera un contrôleur avec les methodes typique dont on a besoin. Index(template), create(template), store(service), show(template), edit(template), update(service) et destroy(service).

À vous de voir si ça vous convient.

**Système de messagerie :**

Pack dispo sur : <https://github.com/cmgmyr/laravel-messenger>

1)Pour installer le pack

>> composer require cmgmyr/messenger

2)Updater le fichier config/app.php. En ajoutant dans le tableau providers :

'providers' => [

Cmgmyr\Messenger\MessengerServiceProvider::class,

]

3)Mettre à dispositions les assets (tout en un ligne):

php artisan vendor:publish –-provider=”Cmgmyr\Messenger\MessengerServiceProvider"

4)S’assurer que vous faites référence aux bons objets (User Model) dans config/messenger.php

5)Faire un >> php artisan migrate

6)Rajouter le trait dans votre user model

use Cmgmyr\Messenger\Traits\Messagable;

class User extends Model {

use Messagable;

}

Dans le lien plus au il y a une exemple de contrôleur.

<https://github.com/cmgmyr/laravel-messenger/blob/master/src/Cmgmyr/Messenger/examples/MessagesController.php>

Exemple de routes :

<https://github.com/cmgmyr/laravel-messenger/blob/master/src/Cmgmyr/Messenger/examples/routes.php>

et finalement exemple de vues :

<https://github.com/cmgmyr/laravel-messenger/tree/master/src/Cmgmyr/Messenger/examples/views>