# LARAVEL (8-2021) – FRAMEWORK PHP 6

# SOMMAIRE

# ETAPE 6

## 1 –Boutons d'action sur chaque todo  v14

**Suite de l'étape précédent**

**Objectifs**

- Ajouter 3 boutons d'action sur chaque todo : done/undone, supprimer, editer

  ⇒ done/undone : vert / jaune

  ⇒ supprimer : rouge

  ⇒ editer: bleu

**Déjà fait :**

- Ajout des boutons
- Action done / undone

**Reste à faire :**

- Action supprimer
- Action éditer

## 2 - Action supprimer (à mettre à jour en version 2025)

### Présentation : l'architecture est MVC

- L'objectif est simple : supprimer la todo quand on clique sur le bouton « supprimer ».
- Etapes quand on déclenche une action comme « supprimer » :
  1. On appelle une route qui est associé à un contrôleur.
     - ⇒ On va donc d'abord définir la route et le contrôleur associé.
     - ⇒ On va ensuite mettre à jour la vue qui appelle la route.
  2. On définit le contrôleur :
     - ⇒ On écrit le code du contrôleur : deleter une todo de la BD.
     - ⇒ Inclure la vue associée au contrôleur (la vue de résultat). Dans ce cas, ce sera un retour à la vue de départ.

### Route

- On fait un : php artisan route:list

⇒ Soit on a déjà une route todos/{todo}/destroy qui s'appelle todos.destroy

⇒ Soit on la crée :

- ⇒ todos/{todo}/destroy
- ⇒ c'est une route DELETE : on va supprimer un élément de la BD.
- ⇒ La route appelle la fonction destroy du contrôleur

```
use App\Http\Controllers\TodoController;

Route::delete(
  'todos/{todo}/destroy',
  [TodoController::class, 'destroy']
)->name('todos.destroy);
```

## Vue appelante

### HTML

- On met à jour la vue appelante pour pouvoir accéder à la route.
- C'est la vue du fichier resources/views/todos/index.php
- L'action (ou le href) est donnée par la méthode route() qui reçoit le nom de la route en paramètre, puis les paramètres dont elle a besoin.

```
<form action="{{route('todos.destroy', $data)}}" method="POST">
    @csrf
    @method("DELETE")
    <input type="submit" value="delete"></p>
</form>
```

- A noter :

  ⇒ $data c'est la todo. C'est un paramètre de la méthode route().

  ⇒ @crsf : pour gérer la sécurité

  ⇒ @method("DELETE") : nécessaire pour une bonne prise en compte des mises à jour de la BD.

## Controleur

- On vient de définir une fonction pour le contrôleur : destroy()

  ⇒ C'est une fonction pour App.Http/Controllers/todoControler.php

  ⇒ Elle a une todo en paramètre

- Cette fonction est peut-être déjà présente, sinon il faut la créer en faisant un copier-coller d'une fonction existant déjà pour avoir le format des commentaires.

- Cette fonction est simple : elle fait un delete de la todo passée en paramètre à jour l'attribut « done ».

```
    public function destroy(Todo $todo)
    {
        $todo->delete();
        return back(); // pour revenir à la page précédente
        // autrement dit pour rester sur la page
    }
```

- A noter :

  ⇒ La méthode delete() qui vient du Modèle.

  ⇒ return back(); // pour revenir à la page précédente

## 3 - Action éditer  v15 (à mettre à jour en version 2025)

### Présentation : l'architecture est MVC

- L'objectif est de pouvoir éditer une todo pour la modifier.

  ⇒ C'est une action équivalente à mettre à jour l'attribut « done », mais en passant par un formulaire de saisie intermédiaire.

- Etapes quand on déclenche une action comme « éditer » :
  1. On appelle une route qui est associé à un contrôleur.

     ⇒ On va donc d'abord définir la route et le contrôleur associé.
  2. On va mettre à jour la vue qui appelle la route.
  3. On définit le contrôleur :

     ⇒ On écrit le code du contrôleur : il ne fait tien d'autre que d'appeler sa vue

     ⇒ Inclure la vue associée au contrôleur (la vue de résultat). Dans ce cas, ce sera un formulaire de saisie.
  4. On définit la vue :

     ⇒ On écrit le code de la vue : le code du formulaire de saisie qui va déclencher une nouvelle action et de nouvelles étapes.

- De là, on repart pour une action déclenchée par le bouton de validation de la saisie :
  5. On appelle une route qui est associé à un contrôleur.

     ⇒ On va donc d'abord définir la route et le contrôleur associé : la route « update »
  6. On va mettre à jour la vue qui appelle la route.
  7. On définit le contrôleur :

     ⇒ On écrit le code du contrôleur : update une todo de la BD.

     ⇒ On inclut la vue associée au contrôleur (la vue de résultat). Dans ce cas, ce sera un retour à la vue de départ.
  8. On teste.

## 1 : Route

- On fait un : php artisan route:list

  ⇒ Soit on a déjà une route todos/edit destroy qui s'appelle todos.edit

  ⇒ Soit on la crée :

    ⇒ todos/{todo}/edit : une route spécifique pour chaque todo

    ⇒ c'est une route GET : la route consiste à appeler un formulaire. C'est le formulaire qui sera PUT

    ⇒ La route appelle la fonction edit() du contrôleur

```
use App\Http\Controllers\TodoController;

Route::get(
  'todos/{todo}/edit',
  [TodoController::class, 'edit']
)->name('todos.edit');
```

## 2 : Vue appelante

### HTML

- On met à jour la vue appelante pour pouvoir accéder à la route.
- C'est la vue du fichier resources/views/todos/index.php

  ⇒ L'action (ou le href) est donnée par la méthode route() qui reçoit le nom de la route en paramètre, puis les paramètres dont elle a besoin.

  ⇒ La method est « GET : à ce stade il n'y aura pas de modification côté serveur.

```
<form class="editer" action="{{route('todos.edit', $data)}}"
method="GET">
    <input type="submit" value="Editer"></p>
</form>
```

- A noter :

  ⇒ $data c'est la todo

  ⇒ @crsf : pour gérer la sécurité

  ⇒ @method("PUT") : nécessaire pour une bonne prise en compte des mises à jour de la BD.

## 3 : Contrôleur

- On vient de définir une fonction pour le contrôleur : edit()

  ⇒ C'est une fonction pour App.Http/Controllers/todoControler.php

  ⇒ Elle a une todo en paramètre

- Cette fonction est peut-être déjà présente, sinon il faut la créer en faisant un copier-coller d'une fonction existant déjà pour avoir le format des commentaires.

- Cette fonction est simple : elle ne fait rien d'autre qu'appeler sa vue : le vue todos.edit.

```
public function edit(Todo $todo)
{
    // dd($todo);
    return view('todos.edit', compact('todo'));
}
```

  ⇒ A noter qu'on passe la $todo en paramètre de la Vue pour afficher les valeurs dans le formulaire.

  ⇒ On peut afficher le contenu de $todo avec un dd()

## 4: La vue edit.blade.php

### Principes

- On va créer un formulaire de saisie qu'on initialise avec les valeurs de de la todo choisie.
- On partir de la vue create.blade.php et la dupliquer en edit.blade.php
- Ensuite, on met à jour les éléments.

### Mise à jour du tire

- On change le titre et on affiche le numéro de la todo concernée :

```
Modification de la todo {{ $todo->id }}
```

### L'action devient :

```
<form
    action="{{ route('todos.update', $todo->id) }}"
    method="POST"
>
    @csrf
    @method("PUT")
```

⇒ La route est todos.update avec l'id de la todo.

### Affichage des informations de la todo :

- Dans l'input du name, on ajoute :

```
value="{{ $todo->name }}"
```

- Dans l'input de la description, on ajoute :

```
value="{{ $todo->description }}"
```

### Ajout d'un bouton checkbox pour le done

- On ajoute avant le submit :

```
<div class="form-group">
    <input
        type="checkbox" name="done" id="done"
        {{ $todo->done ? 'checked' : ' ' }}
        value=1
    >
    <label for="done">Done ?</label>
</div>
```

⇒ Le value=1 permet de récupérer 1 quand la checkbox est checké (sinon, on récupère « on »).

⇒ Quand la checkbox n'est pas checkée, l'attribut checked n'existe pas.

## 5 : Route

- On va créer la route « update » :

```
use App\Http\Controllers\TodoController;

Route::put(
  'todos/update,
  [TodoController::class, 'update']
)->name('todos.update);
```

## 6 : Vue appelante

### HTML

- La vue appelante c'est edit.blade.php :
- Elle est déjà à jour.
- On a déjà mis la route :

```
<form
    action="{{ route('todos.update', $todo->id) }}"
    method="post"
>
    @csrf
    @method("PUT")
```

## 7 : La méthode update() du TodoController

- Le code :

```
public function update(Request $request, Todo $todo)
{
    // dd($todo);
    // dd($request);
    // dd($request->request);
    $todo->update($request->all()) ;
    return redirect()->route('todos');
}
```

**Bizarreries :**

- L'objet $todo contient les valeurs avant la mise à jour !
- C'est $request qui contient les données mises à jour !

  ⇒ Le nouveau todo est dans request->request

**Méthode update()**

- On avait déjà vu la méthode update() avec makedone() et makeundone()
- Sans paramètre, la méthode update() prend l'état de l'instance par défaut.
- Ensuite, il prend tout attribut passé en paramètre correspondant à l'instance

  ⇒ on pourrait faire : $test['name']='coucou';  $todo->update($test);
- Ici on passe le paramètre :  $request->all() :

  ⇒ https://laravel.com/docs/8.x/requests#retrieving-input

  ⇒ Les attributs de l'instance qu'on retrouve dans $request sont mis à jour dans la BD.

## 8 : Tests

- A ce stade, les updates marchent sauf si on passe à « undone »
- Pourquoi ?

  ⇒ On peut deboguer avec dd($request)

  ⇒ On voit que $request->done n'existe pas quand le bouton n'est pas coché.
- On va améliorer le code en ajoutant un attribut done à $request :

```
public function update(Request $request, Todo $todo)
{
    // dd($request);
    if(!isset($request->done)){
        // $request['done']=0;
    }
    $todo->update($request->all());
    return redirect()->route('todos.index');
}
```

⇒ Ca marche !!!

# 4 –Menus pratiques (à mettre à jour en version 2025)

## Objectifs

- Ajouter une barre de menu à gauche.
- Ajouter un accès direct à la page des todos quand on se connecte.

## Accès direct à la barre des todos quand on se connecte

### App/Providers/RouteServiceProvider.php

- Dans le fichier App/Providers/RouteServiceProvider.php :

  ⇒ Il y a une constante HOME = '/home' : c'est la home page de l'application

```
/**
 * The path to the "home" route for your application.
 *
 * This is used by Laravel authentication to redirect users after
login.
 *
 * @var string
 */
public const HOME = '/home';
```

  ⇒ La route home amène sur le HomeController.

- On crée une constante TODOS = /'todos' :

```
public const HOME = '/home';
public const TODOS = '/todos';
```

  ⇒ Il faut mettre à jour les commentaires.

**App/Http/Controllers/Auth/LoginController**

- Dans le fichier App/Http/Controllers/Auth/LoginController :

  ⇒ On cherche HOME :

```
    /**
     * Where to redirect users after login.
     *
     * @var string
     */
    protected $redirectTo = RouteServiceProvider::HOME;
```

  ⇒ On remplace par TODOS

**Test**

- On se connecte :

  ⇒ Ca marche !!! On arrive bien sur la page des todos

## Menu de gauche

### Objectif

- On va créer un petit menu à gauche avec 2 accès : l'apropos et les todos.

### Layout

- Dans le layout, on trouve :

```
<!-- Left Side Of Navbar -->
<ul class="navbar-nav mr-auto">

</ul>
```

- On va y mettre 2 <li> avec class nav-item et 2 <a> avec class nav-link

### Version moche :

```
<li> <a href="{{ route('todos')   }}"> Les todos </a> </li>
<li> <a href="{{ route('apropos') }}"> A propos  </a> </li>
```

- Et les routes : route('todos.index') et route('apropos')

### On peut reprendre le format des <li> de la Right Side Of the Navbar :

```
<li class="nav-item">
    <a class="nav-link" href="{{ route('todos') }}">
        Les todos
    </a>
</li>
<li class="nav-item">
  <a class="nav-link" href="{{ route('apropos') }}">A propos</a>
</li>
```

# 5 – Première gestion des utilisateurs (à mettre à jour en version 2025)

## Bilan de l'étape 3

- On peut trouver le code complet en version bootstrap ici :
- http://bliaudet.free.fr/IMG/zip/maTodolist-E3-complet-base.zip
- Pour l'étape 4, on part de ce code et on fait les mises à jour.
- Tout le document peut être vu comme un TP.

## Voir les utilisateurs facilement

### Objectifs

Affecter une todo à un utilisateur : un bouton, parmi les boutons d'action, qui liste les utilisateurs.

### Principes

- On va se doter d'une fonction d'accès aux utilisateurs qu'on place dans le modèle User.php
- On met à jour le TodoController pour récupérer les utilisateurs via un constructeur.
- On ajoute le nouveau bouton dans todos/index.blade.php

### On modifie le modèle User.php

- On crée une fonction statique : accessible sans objet.
- On pourrait se passer de la fonction et écrire User::all() quand on en a besoin : c'est mieux d'utiliser le MVC.

```
public static function getAllUsers(){
  return User::all();
}
```

### Initialisation de $users dans le TodoController

➢ ***attribut :***

```
    public $users; // on se donne cette propriété
```

➢ ***constructeur :***

```
public function __construct(){
      this->users = User::getAllUsers();
}
```

➢ ***vérification :***

- On peut par exemple, mettre au début de la méthode edit() du TodoController :

```
dd($this->users);
```

⇒ On teste en cliquant que le bouton d'action « éditer ».

### Passage de $users aux à la vue todos.index

➢ ***Méthodes qui retournent la vue todos/index***

- Dans les méthodes index, done et undone, on récupère les users et on les passe en paramètre à la vue.
- Ainsi, la vue pourra afficher les users.

```
$users = $this->users;
return view('todos.index', compact('datas, 'users'));
```

➢ ***vérification :***

- On peut vérifier en ajoutant cette ligne avant le foreach dans le fichier todos/index.blade.php :

```
Route :
    {{ Route::currentRouteName() }} <br>
Utilisateur connecté :
    {{ Auth::user()->name}} : {{ Auth::user()->id}} <br>
Les users :
    @foreach ($users as $user) {{ $user->name }}, @endforeach<br>
```

⇒ On teste en cliquant sur les boutons de menus : terminées, ouvertes, toutes.

⇒ Auth ::user() permet de récupérer l'utilisateur connecté.

⇒ Il est défini dans Illuminate\Support\Facades\Auth;

**Affichage dans la vue todos.index**

- On va modifier la vue todos.index pour afficher le bouton avec les utilisateurs.
- Dans todos.index.blade.php :

⇒ avant les 3 boutons d'action, un nouveau bouton : affected to

⇒ On insère un template bootstrap

```
b4-drop : dropdown-button
```

- on met à jour :

⇒ l'id du bouton : dropdownMenuButton

⇒ le texte affiché : Affecter à

⇒ on supprime les button de la div dropdown-menu

⇒ on y mettra à la place tous les users :

```
@foreach ($users as $user)
  <a class="dropdown-item" href="#">{{$user->name}}</a>
@endforeach
```

- On peut tester :

⇒ on a le bouton qui s'ouvre sur les users de la BD.

⇒ ça marche !!!

## Affecter une tâche à un utilisateur

### Principes

- Le bouton qu'on vient de créer va nous permettre d'affecter une tâche à un utilisateur.

  ⇒ Il va falloir définir la route pour le choix d'un utilisateur dans le bouton.

  ⇒ Il faudra mettre à jour la tables des users pour savoir à qui la todo est affectée.

  ⇒ Il faudra écrire la méthode associée à route choisi.

  ⇒ On va commencer par la mise à jour de la table des users.

## Mise à jour de la table des users

### Nouveaux attributs de la todo :

- Une todo a de nouvelles caractéristiques :

  ⇒ L'utilisateur créateur de la todo : createdBy_id

  ⇒ L'utilisateur qui a affecté la todo : affectedBy_id

  ⇒ L'utilisateur qui doit faire la todo : affectedTo_id

- A noter qu'un utilisateur pourra toujours affecter sa todo à un autre, même s'il n'est pas créateur de la todo (d'où la distinction entre l'utilisateur créateur et l'utilisateur affectant).

### Mettre à jour la BD : migration

- On va créer une migration pour ajouter ces attributs.

```
php artisan make:migration
add_createdby_affectedby_affectedto_to_todos_table
```

⇒ Le squelette est fait

- On modifie up et down qui on déjà un ::table de modification
- up() :

```
$table->bigInteger('createdBy_id')->default(0)
      ->after('done');
$table->bigInteger('affectedBy_id')->default(0)
      ->after('createdBy_id');
$table->bigInteger('affectedTo_id')->default(0)
      ->after('affectedBy_id');
```

⇒ default() permet d'avoir une valeur même si on ne saisit pas. Ce n'est pas vraiment utile.

⇒ after() positionne l'attribut dans la table après celui précisé

- down() :

```
$table->dropColumn('createdBy_id');
$table->dropColumn('affectedBy_id');
$table->dropColumn('affectedTo_id');
```

- Une fois ça fait, on vérifie la situation des migrations :

```
>php artisan migrate:status
```

- On effectue la migration :

```
>php artisan migrate
```

- On vérifie  en BD :

```
mysql>desc user
```

⇒ La table a bien été mise à jour.

**Vérification des todos**

- On consulte toutes les todos en BD :

```
mysql>select * from user\G
```

⇒ Les attributs valent tous 0 pour toutes les todos.

- Il faudra mettre à jour ces attributs.

⇒ L'application permettra que ces attributs soient mis à jour pendant le cycle de vie des todos.

⇒ Mais avec nos todos générées automatiquement, il faudra les mettre à jour à la main (par des requêtes SQL) ou revoir le code de génération automatique.

**Mise à jour des todos**

- On va supprimer toutes les todos :

⇒ On peut faire ça avec l'application.

⇒ On peut faire ça avec une requête SQL :

```
mysql>delete from todos;
```

- On va recréer des todo avec la TodoFactory.
- On commence par mettre à jour la TodoFactory :

```
'createdBy_id' =>$this->faker->numberBetween(1,2),
```

⇒ Ainsi, on aura 2 créateurs pour nos todos

- Ensuite on lance le seeder :

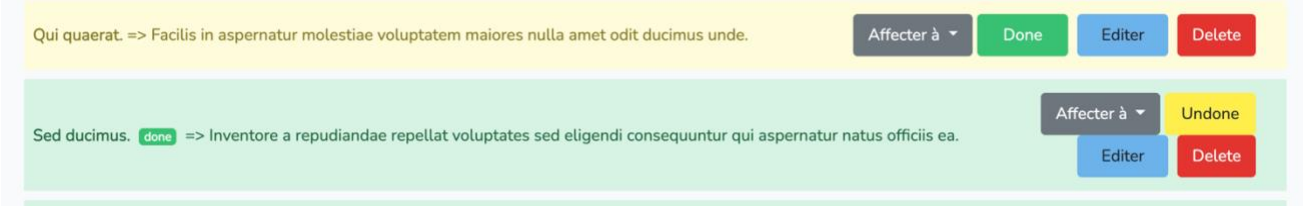
```
>php artisan db:seed
```

⇒ On peut visualiser nos 25 todos dans l'application.

## Mise à jour de l'affichage

### Situation et objectif

- A ce stade, on obtient ce résultat :




- On veut que la zone d'affichage de la todo soit de taille fixe et que les boutons s'affiche toujours (quand il y a la place) l'un à côté de l'autre.

### Solution

- On a un problème d'affichage des boutons : il faudra mettre le texte et les boutons sur 2 colonnes.
- Dans todos/index.blade.php : on ajoute col-sm sur la div de la todo :

```
<div class="col-sm">
```

## Tri des todos

### Objectif

- On va afficher les todos de la plus récente à la plus ancienne.

### Solution

```
$datas = Todo::orderBy('id', 'desc')->paginate(10);
```

## Définir la route pour le choix d'un utilisateur dans le bouton et méthode associé

### La route

- Dans l'affichage des utilisateurs sur le bouton, on a la route à définir :

```
@foreach ($users as $user)
  <a class="dropdown-item" href="#">{{$user->name}}</a>
@endforeach
```

- La route sera la suivante :

```
    href="/todos/{{$data->id}}/affectedTo/{{$user->id}}"
```

⇒ On passe une todo ($data->id) et un user ($user->id) en paramètre de la route

- Il faut créer la route dans web.php :

```
Route::get(
    'todos/{todo}'/affectedTo/{user}',
    [TodoController::class, 'affectedTo']
)->name('todos.affectedTo')
```

⇒ Cette route est associée à la méthode affectedTo() qu'il faudra écrire.

### La méthode affectedTo()

La fonction affectedTo() est dans le TodoController.

```
use Illuminate\Support\Facades\Auth;

/**
 * set a todo to a user
 * @param  Todo  $todo
 * @param  User  $user
 * @return \Illuminate\Http\Response
 */
public function affectedTo(Todo $todo, User $user)
{
    $todo->affectedTo_id = $user->id;
    $todo->affectedBy_id = Auth::user()->id;
    $todo->update();
    // dd($todo); // debug
    return back();
}
```

⇒ Ca marche !!!

⇒ On peut mettre un dd($todo) pour déboguer.

⇒ A noter le use départ pour accéder à la classe Auth. Auth ::user() fournit l'utilisateur connecté.

⇒ On peut vérifier en BD que les valeurs de affectedTo et affectedBy sont bien à jour.

### Vérification

- On ajoute dans le fichier todos/index.blade.php, après la description :

```
{
   createur: {{ $data->createdBy_id }},
   affecté à: {{ $data->affectedTo_id }},
   affecté par: {{ $data->affectedBy_id }}
}
```

⇒ On peut vérifier tout ce qu'on fait.

## Mise à jour du createdBy_id

### Objectif

- A ce stade, quand on crée une nouvelle todo, elle n'a pas de createdBy_id, pas plus que de affectedTo_id.
- On que chaque nouvelle todo ait un createdBy_id et un affectedTo_id.

### Solution

- On va gérer ça dans la fonction store() du TodoController.
- Dans store(), on ajoute :

```
$todo->createdBy_id = Auth::user()->id ;
$todo->affectedTo_id = Auth::user()->id ;
```

⇒ Le affectedTo_id n'est pas géré : il sera géré par le bouton d'action.

⇒ Ca marche !!!

## Affichage des todos en fonction de l'utilisateur connecté

### Objectif

- On ne va afficher que les todos créées par l'utilisateur connecté et celles qui lui sont affectées.

### Solution

- Pour ça, il faut filtrer les données dans la fonction index() du TodoController.
- On va aussi calculer le nombre de todos.

```
$userId = Auth::user()->id;

$datas = Todo::where('affectedTo_id', $userId)
    ->orWhere('createdBy_id', $userId)
    ->orderBy('id', 'desc')
    ->paginate(10);

$countTodos = Todo::where('affectedTo_id', $userId)
    ->orWhere('createdBy_id', $userId)->count();
```

⇒ Doc Laravel : https://laravel.com/docs/8.x/queries#or-where-clauses

⇒ Ca marche !!!

## Petit exercice :

### Objectif

- Affichez le numéro de la todo, au même format que le badge « done », au début de la todo, et fond sombre.

### Solution

- A trouver : regardez le code d'affichage des todos.

## Affichage de l'identifiant de la todo

### Objectif

- Affichez le numéro de la todo, au même format que le badge « done », au début de la todo, et fond sombre.

### Solution

- Dans todos/index.blade.php, dans le foreach, avant le nom de l'utilisateur {{ $data->name }} :

```
    <span class="badge badge-dark mx-1">{{ $data->id }}</span>
```

## Affichage de la description en déroulé

### Fonctionnel (objectifs)

- On veut que la description de la todo « s'ouvre » sous le titre de la todo.
- On présentera le n° de la todo sur une première ligne en gras.
- Puis le titre avec le done éventuel et les infos de création et d'affectation.
- Ensuite, on aura la description déroulante.

### Architecture

- Dans todos/index.blade.php, dans le foreach

### Technique

- On utilise les balises <detail> et <summary>

### Solution complète

```
<details>
   <summary><strong>
      {{ $data->name }}
      @if ($data->done)
         <span class="badge badge-success mx-1">done</span>
      @endif
      createur: {{ $data->createdBy_id }},
      affecté à: {{ $data->affectedTo_id }},
      affecté par: {{ $data->affectedBy_id }}
   </strong></summary>
   <p>{{ $data->description }}</p>
</details>
```

### Améliorations

- Il faut supprimer l'encadré du titre quand on déroule : HTML
- Il faut que les boutons ne bougent pas quand on déroule : HTML.