TP n°8 - un peu plus loin avec Laravel

Site: Université de Reims Champagne-Ardenne
Cours: INFO0303 - Technologies Web 2
Livre: TP n°8 - un peu plus loin avec Laravel

Imprimé par: CYRIL RABAT
Date: samedi 21 octobre 2023, 14:52

Table des matières

1. Pour commencer

Le premier objectif de ce TP est d'ajouter l'authentification en utilisant *Breeze* ou *Jetstream*. Le second objectif est de proposer un complément sur les bases de données en expliquant la création d'une relation *1-n* (en associant un article à un fournisseur) et d'une relation *n-n* (en associant un article à des catégories).

Pour installer *Breeze* ou *Jetstream*, nous avons besoin de *npm* qui est un gestionnaire de paquets pour *Node.js*. Il existe pour les distributions *Mac*/*Linux*. Pour *Windows*, il existe un outil équivalent : nvm pour Windows.

1. Installez *nvm* pour *Windows* (ou *npm* pour les autres) (cherchez *nvm-setup.exe*).

> Si vous avez déjà installé *nvm* et que vous n'avez pas la dernière version, relancez simplement l'installeur. Il détecte l'ancienne version et la désinstalle à la demande.

2. Une fois *nvm* installé, ouvrez l'invite de commandes **en mode administrateur** puis tapez les commandes suivantes :

```
nvm install latest
nvm use latest
```

3. Vérifiez que *Node.js* et *npm* sont correctement installés avec les commandes suivantes :

```
node -v
npm -v
```

2. Gestion des utilisateurs

Il existe plusieurs "*Starter Kits*" pour la gestion des utilisateurs dans *Laravel*., chacun ayant plusieurs piles possibles. Dans ce TP, nous allons utiliser *Breeze* (avec *Blade*) et *JetStream* (avec *Livewire*).

Le premier est plus léger mais possède moins de fonctionnalités que *JetStream*. Pour le projet, vous pouvez choisir l'un ou l'autre.

2.1. Breeze

Pour commencer, il faut installer *Breeze* à l'aide de composer.

1. Créez un nouveau projet *Laravel*.

2. Tapez la commande suivante (dans le répertoire de l'application) :

```
composer require laravel/breeze --dev
```

3. Installez via *artisan* et *npm* (choisissez "*blade with Alpine*" comme "*Breeze stack*")

```
php artisan breeze:install
npm install
npm run build
```

L'installation a ajouté de nombreux éléments :

- Des routes (dans `routes/web.php` et `routes/auth.php`)
- Des vues (dans `ressources/views/`, les répertoires `auth/`, `components/` et `layouts/`, la vue `dashboard.blade.php`)
- Des contrôleurs, des requests (répertoire `app/Http`)

Tout est bien entendu personnalisable.

Comme vous pouvez le constater dans le fichier `web.php`, une route a été ajoutée vers le *dashboard* (les autres routes sont disponibles dans le fichier `routes/auth.php`) :

```
Route::get('/dashboard', function () {
    return view('dashboard');
})->middleware(['auth', 'verified'])->name('dashboard');
```

L'installation supprime toutes les routes qui ont été créées précédemment. L'installation de *Breeze* doit être effectuée avant de développer le moindre élément.

4. Lister toutes les routes à l'aide de la commande suivante :

```
php artisan route:list
```

5. Saisissez l'URL `/register` (ou cliquez sur le lien "*Register*" sur la page principale) et créez un compte.

Pour le moment, cela vous amène sur le *dashboard* et vous êtes connecté à l'application. Il est possible de régler le comportement par défaut.

6. Depuis le *dashboard*, vous pouvez cliquez sur "Log out" dans le menu.

7. Tapez maintenant l'URL `/login` et saisissez à nouveau vos identifiants.

8. Accédez directement à l'URL `/logout`, .

Comme vous pouvez le constater, cela génère une erreur. En effet, c'est une route de type "POST" et non "GET". Il faut donc passer par un formulaire (vous pouvez d'ailleurs voir celui créé dans le *dashboard*) qui nécessite le jeton `@csrf`.

2.2. Jetstream / Livewire

Attention !!! Vous ne devez pas installer *Jetstream* / *Livewire* sur le même projet que *Breeze*. Créez d'abord une nouvelle application *Laravel*.

1. Pour installer *Jetstream* à l'aide de *composer* tapez la commande suivante :

```
composer require laravel/jetstream
```

Pour la partie *front*, *JetStream* propose deux "piles" : *Livewire* ou *Inertia*. Cette dernière utilisant *vue.js*, nous faisons le choix de *Livewire*.

2. Installer *Livewire* à l'aide d'*artisan* :

```
php artisan jetstream:install livewire
```

3. N'oubliez pas de configurer le fichier `.env` (pour la base de données). Finalisez l'installation à l'aide des commandes suivantes :

```
npm install
npm run build
php artisan migrate
```

La dernière commande permet de prendre en compte les tables et champs nécessaires pour *Jetstream*. Comme pour *Breeze*, les différentes routes `/login`, `/register` sont maintenant disponibles.

4. Testez la création du compte, la connexion, la déconnexion et les options du *dashboard*.

Il peut y avoir des problèmes d'accès au fichier `livewire.js`. La solution est de créer un hôte virtuel ou bien d'utiliser la commande `php artisan serve` et d'accéder à l'URL spécifié (**http://127.0.0.1:8000**).

2.3. Retour sur les articles

Vous avez installé 2 applications pour tester *Breeze* et *Livewire*. Supprimez l'une des deux.

Nous allons maintenant reprendre l'application du TP précédent. Vous trouverez les sources sur le dépôt *gitlab* suivant : https://gitlab-mi.univ-reims.fr/rabat01/tp7 (vous pouvez la cloner dans un autre répertoire et la supprimer ensuite).

1. Installez la `DebugBar`.

2. Récupérez les fichiers suivants depuis le TP précédent et placez-les dans la nouvelle application :

   - app\Http\Controllers\ItemController.php
   - app\Http\Requests\StoreItemRequest.php
   - app\Models\Item.php
   - resources\views\items* (toutes)
   - resources\views\template.blade.php
   - database\migrations\2023_XX_XX_XXXXX_create_items_table.php (la migration qui permet de créer la table items)
   - database\factories\ItemFactory.php
   - database\seeders\DatabaseSeeder.php
   - database\seeders\ItemSeeder.php

3. Remplacez la route par défaut (sur la vue `welcome`) par les deux routes suivantes :

```
use App\Http\Controllers\ItemController;
Route::resource('/', ItemController::class);
Route::resource('item', ItemController::class);
```

4. Lancez maintenant la migration et le peuplement de la base.

5. Vérifiez que la page principale affiche bien nos articles. php artisan migrate seed

6. Pour que l'utilisateur soit redirigé vers la page principale après s'être logué, vous pouvez modifier la valeur de la constante `HOME` dans la classe `app/Providers/RouteServiceProvider` (spécifiez `/` au lieu de `/dashboard`).

2.4. Restriction des accès

Pour illustrer le fonctionnement de *Breeze* ou de *Jetstream*, nous allons obliger l'utilisateur à être connecté pour pouvoir ajouter, modifier et supprimer les articles.

Pour cela, nous pouvons utiliser les directives *blade* suivantes qui permettent d'exécuter le code uniquement si l'utilisateur est connecté :

```
@auth
  Code exécuté si l'utilisateur est connecté
@endauth
```

1. Ajoutez le code suivant dans la vue `items/list.blade.php` au début de la section `content` (supprimez l'ancien `div` qui contenait le bouton de création) :

```
<form id="formLogout" action="{{url('/logout')}}" method="POST">
  @csrf
</form>
<div class="d-flex justify-content-center">
  <span>
@auth
    <a href="{{route('item.create')}}" class="btn btn-sm btn-primary mb-2 mr-2">
      Création d'un nouvel article
    </a>
    <button type="submit" form="formLogout" class="btn btn-sm btn-danger mb-2 mr-2">
      Déconnexion
    </button>
@else
    <a href="{{ url('/login') }}" class="btn btn-sm btn-primary mb-2 mr-2">
      Connexion
    </a>
    <a href="{{ url('/register') }}" class="btn btn-sm btn-primary mb-2 mr-2">
      Création d'un compte
    </a>
@endauth
 </span>
</div>
```

2. Protéger les boutons de modification et de suppression à l'aide des directives *blade* (entre `@auth` et `@endauth`).

3. Testez l'application et vérifiez que les boutons de création, modification et suppression sont bien absents.

4. Cliquez sur le bouton de connexion ou de création de compte (si vous ne l'avez pas déjà fait). Retournez sur la page principale et vérifiez que les boutons sont maintenant affichés.

5. Déconnectez-vous et vérifiez que vous pouvez toujours accéder à l'URL ~~news/create~~ item/create

Pour le moment, c'est tout à fait normal : nous n'avons joué que sur l'affichage et nous n'avons fait aucun contrôle dans le code. La classe `Auth` propose les fonctions suivantes :

- `Auth::user()` : récupère l'objet correspondant à l'utilisateur actuellement connecté (ou `null` sinon)
- `Auth::id()` : récupère l'identifiant de l'utilisateur connecté
- `Auth::check()` : retourne `true` si l'utilisateur est connecté

N'oubliez pas d'importer la classe lors de son utilisation :

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

6. Dans la méthode `create` du contrôleur des articles, ajoutez l'instruction suivante et vérifiez le comportement de votre application :

```
if(!Auth::check())
  return redirect('login');
```

app.controllers.Auth.ItemController.php

7. Vérifiez que vous êtes automatiquement redirigé vers la page de login si vous souhaitez accéder à la création d'un article. oui et j'ai rajouté

8. Protégez également toutes les autres méthodes qui le nécessitent.

```
elseif(Auth::check()&& Auth::id()
===1){
        return view('items.create');
    }
```

Il est possible de protéger toutes les opérations d'un contrôleur en ajoutant le constructeur suivant :

```php
public function __construct() {
  $this->middleware('auth');
}
```

middleware remplace le if(Auth:check()){}, quand on crée l'objet il va directement vérifier. Ici l'objet c'est le controller qui est appelé dans le web.php comme ceci Route::resource('/', ItemController::class);

Dans notre cas, comme certaines méthodes sont accessibles aux utilisateurs non connectés (comme la liste des articles), il est possible de créer 2 contrôleurs : l'un avec les méthodes publiques, l'autre avec les méthodes qui nécessitent une connexion. donc peut être 3; un troisième pour l'admin

Il est également possible de protéger les routes depuis le fichier `web.php` en appelant la méthode `middleware`. Par exemple :

```php
Route::get('page1', function () {
    return view("page1"); }
  )->middleware('auth');
```

Laravel propose des solutions complètes pour gérer l'authentification et les droits associés aux utilisateurs (les autorisations). Nous verrons cela dans un prochain TP.

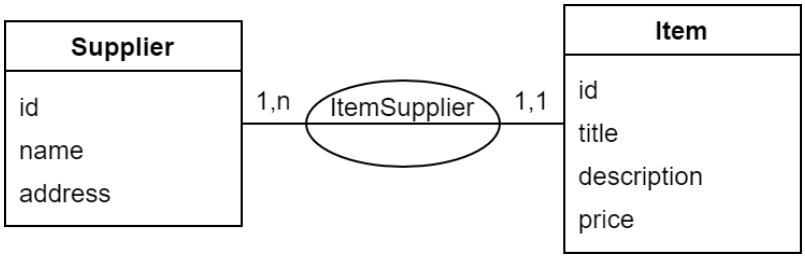
```php
if(!Auth::check())
 return redirect('login');
elseif(Auth::check()&& Auth::id()===1){
        return view('items.create');
    }
```

3. Un peu plus loin avec la base de données

Pour le moment, nous avons utilisé la base de données de manière plutôt simpliste (une seule table, pas de relation). Nous allons étudier les deux cas les plus courants : les relations *1-n* et *n-n*.

3.1. Relation 1-n

Pour illustrer le fonctionnement de la relation 1-n, nous allons associer les articles à des fournisseurs comme le montre le MCD suivant :



Nous devons modifier notre base de données, ainsi que les modèles et contrôleurs. Par défaut, *Laravel* utilise le moteur de stockage *MyISAM*, ce qui empêche la création de clés étrangères.

1. Éditez le fichier `config/database.php` et remplacez dans la ligne ci-dessous (dans la configuration de *MySQL*) le mot `null` par `'InnoDB'`.

```
'engine' => null,
```

2. Relancez la migration à la l'aide de la commande suivante et vérifiez avec *phpMyAdmin* que le moteur de stockage de la table `items` est bien `InnoDB`.

```
php artisan migrate:fresh --seed
```

3. Créez un nouveau modèle nommé `Supplier` (pour les fournisseurs) avec sa migration associée. Spécifiez le nom (une chaîne de 100 caractères) et l'adresse (du texte). Créez une *factory* associée (avec le *faker* vous pouvez utiliser les méthodes `company` pour le nom et `address` pour l'adresse) et un *seeder* (créez 10 fournisseurs). N'oubliez pas de modifier la classe `DatabaseSeeder` pour appeler le *seeder* des fournisseurs.

4. Lancez la migration et vérifiez que les fournisseurs ont été créés dans la base de données.

Pour illustrer le principe de l'amélioration continue à l'aide des migrations, nous allons créer une nouvelle migration pour modifier la table `items` (au lieu de modifier la migration existante).

5. Créez une migration nommée `add_supplier_to_item`.

```
php artisan make:migration add_supplier_to_item --table=items
```

6. Dans cette migration (méthode `up`), nous ajoutons la clé étrangère dans la table `items` vers l'identifiant de l'utilisateur dans la table `supplier` :

```
Schema::table('items', function (Blueprint $table) {
    $table->unsignedBigInteger('supplier_id');
    $table->foreign('supplier_id')->references('id')->on('suppliers');
  });
```

7. Dans la méthode `down`, nous pouvons effectuer l'opération inverse comme suit :

```
Schema::table('items', function (Blueprint $table) {
    $table->dropForeign('supplier_id');
  });
```

8. Exécutez la migration (sans l'option `fresh` !!!).

Comme il existe déjà des articles dans la base, il y a des problèmes de contraintes avec la clé étrangère : aucun identifiant n'a été spécifié comme fournisseur. Pour simplifier, nous allons réinitialiser la base de données.

9. Exécutez la migration (sans le peuplement) avec l'option `fresh`.

- Pour éviter de perdre des données, nous aurions pu spécifier des données aléatoires dans la migration ;
- Si vous avez des tables avec des clés étrangères, le vidage de la table dans les *seeders* à l'aide de la méthode `truncate` génère une erreur lors du peuplement.

10. Dans le modèle `Item`, ajoutez `supplier_id` dans l'attribut `fillable` et la méthode suivante dans la classe (pour récupérer le fournisseur d'un article) :

```
public function supplier() : BelongsTo {
  return $this->belongsTo(Supplier::class);
}
```

N'oubliez pas d'ajouter le `use` avant la classe :

```
use Illuminate\Database\Eloquent\Relations\BelongsTo;
```

11. Dans le modèle `Supplier`, ajoutez la méthode suivante pour pouvoir récupérer les articles d'un fournisseur donné (n'oubliez pas le `use`) :

```
public function items() : HasMany {
  return $this->hasMany(Item::class);
}
```

Nous allons maintenant modifier les *seeders* afin de tester la relation articles/fournisseurs.

12. Supprimez la création du "Canard en plastique" dans la méthode `run` de la classe `ItemSeeder`. (sinon, vous aurez une erreur lors de la création, le `supplier_id` n'étant pas spécifié).

Dans le *seeder* associé aux articles, nous devons maintenant ajouter des fournisseurs aléatoires.

13. Spécifiez le code suivant dans la méthode `definition` de la classe `ItemFactory` :

```
$suppliersID = DB::table('suppliers')->pluck('id');
return [
        'title' => $this->faker->realText(20),
        'description' => $this->faker->realText(200),
        'price' => $this->faker->randomFloat(2, 0, 99.99),
        'supplier_id' => $this->faker->randomElement($suppliersID)
    ];
```

La méthode `pluck` sur la table permet de récupérer tous les identifiants de la table `suppliers`. La méthode `randomElement` du `faker` permet ensuite d'en choisir un aléatoirement pour chaque article.

14. Lancez le peuplement et vérifiez le contenu des tables de la base de données.

15. Dans la vue `items.list`, ajoutez les lignes suivantes après l'affichage de la description de l'article (vous pouvez également ajouter ces lignes dans la vue `items.show`):

```
<br/>
vendu par <em>{{$item->supplier->name}}</em>
```

16. Actualisez la liste des articles dans votre navigateur : maintenant, les noms apparaissent pour chaque produit.

À noter que l'ajout d'un article n'est plus fonctionnel. Il faut ajouter le fournisseur.

17. Modifiez la méthode `edit` du contrôleur `ItemController` (n'oubliez pas le `use` pour la classe `supplier`) :

```
return view('items.edit', ['item' => Item::findOrFail($id), 'suppliers' => Supplier::orderBy('name', 'asc')->get()]);
```

18. Après le champ pour le prix, ajoutez une liste déroulante pour le choix du fournisseur dans la vue `items.edit` :

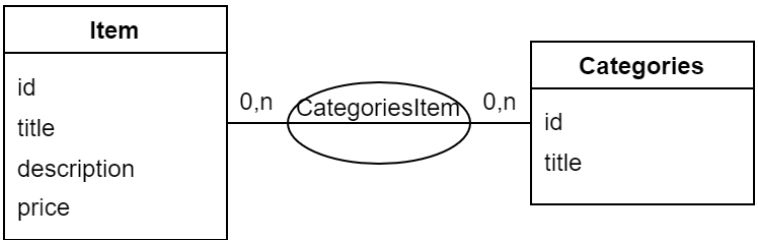
```
<div class="mb-3 row">
  <label for="supplier_id" class="col-sm-2 col-form-label">Fournisseur</label>
  <div class="col-sm-10">
    <select name="supplier_id" class="form-select" id="supplier_id" placeholder="Sélectionnez le fournisseur">
@foreach($suppliers as $supplier)
  @if($item->supplier->id == $supplier->id)
      <option selected value="{{$supplier->id}}">{{$supplier->name}}</option>
  @else
      <option value="{{$supplier->id}}">{{$supplier->name}}</option>
  @endif
@endforeach
    </select>
  </div>
</div>
```

19. Vérifiez que l'édition d'un article est fonctionnelle.

Vous pouvez également modifier la fonction `store` de la même manière, ainsi que la vue `items.create` (attention, il faut remplacer `$item->supplier->id` par `old('supplier_id')`).

3.2. Relation n-n

Pour illustrer le fonctionnement de la relation *n-n*, nous proposons d'associer les articles à des catégories comme le montre le MCD suivant :



En pratique, cela implique la création d'une table pivot permettant de faire le lien entre les articles et les catégories.

1. Créez un modèle `Category` pour les mots-clés avec la migration associée ; une catégorie correspond simplement à un intitulé court (`title`). Notez que la table est intitulée `categories`.

2. Créez un *seeder* et une *factory* pour générer des catégories aléatoires (créez 10 catégories).

3. Dans le modèle `Item`, ajoutez la méthode suivante :

```
public function categories() : BelongsToMany {
  return $this->belongsToMany(Category::class);
}
```

Vous pouvez dorénavant récupérer les catégories d'un article comme suit (où `X` est l'identifiant de l'article et `title` est le champ utilisé pour l'intitulé de la catégorie) :

```
$categories = Item::find(X)->categories()->orderBy('title')->get();
```

4. Dans le modèle `Category`, ajoutez la méthode suivante, ce qui permettra de récupérer les articles associés à la catégorie :

```
public function items() : BelongsToMany {
  return $this->belongsToMany(Item::class);
}
```

5. Enfin, créez la migration pour la table pivot :

```
php artisan make:migration create_category_item_table
```

6. Modifiez la migration en ajoutant les clés étrangères (supprimez l'identifiant qui est inutile) :

```
$table->timestamps();

$table->unsignedBigInteger('item_id');
$table->unsignedBigInteger('category_id');
$table->timestamps();

$table->primary(['item_id', 'category_id']);
$table->foreign('item_id')->references('id')->on('items');
$table->foreign('category_id')->references('id')->on('categories');
```

7. Pour associer aléatoirement des catégories aux articles, ajoutez le code suivant dans le *seeder* des articles (après la création des 10 articles) :

```
$faker = \Faker\Factory::create();
foreach(Item::all() as $item) {
    $categories = Category::inRandomOrder()->take(rand(1, 3))->pluck('id');
    $item->categories()->attach($categories, [
            'created_at' => $faker->dateTimeBetween('-10 day'),
            'updated_at' => $faker->dateTimeBetween('-10 day')
        ]);
}
```

8. Éditez `DatabaseSeeder` et ajoutez les appels aux *seeders* (dans quel ordre ?).

Il reste, entre autres, à afficher les catégories associées à un article. Pour cela, il faut modifier la vue `items.show`.

9. Ajoutez le code suivant (analysez les annotations *blade*) :

```
...
{{$item->description}}<br/>
Vendu par <em>{{$item->supplier->name}}</em><br/>
@foreach($item->categories as $category)
  @if($loop->first)
    <b>Catégorie(s)</b> :
  @endif
  {{$category->title}}
  @if(!$loop->last)
    ;
  @else
    <br/>
  @endif
@endforeach
```

10. Lancez la migration et le peuplement.

4. Le soft-deleting

La méthode de suppression proposée dans le TP précédent implique la suppression définitive des enregistrements. Or, il peut être nécessaire sur un site en production de les conserver dans la base de données pour avoir un historique. Cette méthode est appelée le *soft-deleting*.

1. Créez une nouvelle migration pour ajouter ce champ à la table `items` :

```
$table->softDeletes();
```

2. Dans la méthode `down`, l'instruction est la suivante :

```
$table->dropColumn('deleted_at');
```

3. Dans le modèle, spécifiez l'utilisation du trait `SoftDeletes` (il est situé dans le package Illuminate\Database\Eloquent) et l'attribut `dates` comme indiqué dans ce code :

```
use SoftDeletes;
...
protected $dates = [
    'created_at',
    'deleted_at',
    'started_at',
    'update_at'
  ];
```

Avec ces modifications (et après avoir effectué la migration), les articles ne sont plus supprimés définitivement et le champ `deleted_at` prend la date courante lors de la suppression.

4. Supprimez plusieurs articles.

5. Vérifiez qu'avec l'instruction suivante (dans la méthode `index` du contrôleur des articles) les articles supprimés sont présents :

```
$itemList = Item::withTrashed()->orderBy('title', 'desc')->take(10)->get();
```

5. Annexes : commandes Laravel

Installation d'une application *Laravel* de zéro :

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

Installation d'une application *Laravel* existante (récupérée depuis un dépôt, par exemple) :

```
composer install
php artisan key:generate
```

Création du lien pour les ressources publiques (images, CSS, JavaScript) :

```
php artisan storage:link
```

Installation de la barre de debug :

```
composer require barryvdh/laravel-debugbar --dev
```

Publication d'un package "vendor" :

```
php artisan vendor:publish
```

Liste des routes :

```
php artisan route:list
```

Lancement de la migration :

```
php artisan migrate
```

Annulation de la dernière migration :

```
php artisan migrate:rollback
```

Lancement de la migration ET du peuplement de la base :

```
php artisan migrate --seed
```

Lancement de la migration depuis zéro (suppression des tables et des données) :

```
php artisan migrate:fresh
```

Lancement de la migration depuis zéro ET du peuplement de la base :

```
php artisan migrate:fresh --seed
```

Création d'une migration (en spécifiant la table associée) :

```
php artisan make:migration nom_migration --table=nom_table
```

Création d'un modèle et de la migration associée :

```
php artisan make:model nom_model --migration
```

Création d'un seeder :

```
php artisan make:seed nom_seeder
```

Création d'une factory :

```
php artisan make:factory nom_factory
```

Lancement du peuplement de la base :

```
php artisan db:seed
```

Lancement du peuplement de la base d'un seeder spécifique :

```
php artisan db:seed --class=MonSeeder
```

Création d'un contrôleur :

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

Création d'un contrôleur CRUD :

```
php artisan make:controller MonController --resource
```

Création d'une requête

```
php artisan make:request StoreXXXRequest
```

Installation de *Breeze* :

```
composer require laravel/breeze --dev
php artisan breeze:install
npm install
npm run build
```

Installation de *Jestream*/*Livewire* :

```
composer require laravel/jetstream
php artisan jetstream:install livewire
npm install
npm run build
```