Cours Laravel 6 – les données – les ressources d'API

laravel.sillo.org/cours-laravel-6-les-donnees-les-ressources-api/

bestmomo

September 6, 2019

Laravel permet de bâtir facilement des API à partir de ressource Eloquent. Voyons cela dans cet article.

Les API REST

REST (Representational State Transfer) est un style d'architecture pour constituer des API. Pour Laravel une API REST est :

- organisée autours de ressources
- sans persistance (aucune session)
- orientée client-serveur
- pouvant être mis en cache (donc une requête doit retourner toujours les mêmes résultats)
- retourner du JSON…

Si vous n'êtes pas trop sûr de vos connaissances concernant les API REST je vous conseille ce cours.

Pour continuer avec notre application de films on peut imaginer ce **endpoint** pour les informations de tous les films :

```
GET /api/films
[
  {
    id: 1,
    title: 'Aut numquam.'
  },
  {
    id: 2,
    title: 'Sint incidunt consequatur.'
  }
  ...
]
```

Ou ce **endpoint** pour les informations concernant un film en particulier :

```
GET /api/films/2
{
  id: 2,
  name: 'Sint incidunt consequatur.'
}
```

On peut imaginer d'autres endpoints pour modifier, ajouter, supprimer…

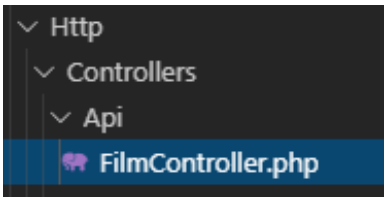
Il ne reste plus qu'à voir comment on réalise ça avec Laravel mais vous en avez désormais une idée déjà assez précise avec ce qu'on a déjà vu dans les précédents articles.

Le contrôleur

On va créer le contrôleur :

```
php artisan make:controller Api/FilmController --resource
```

On le retrouve rangé dans le dossier **app/Http/Controllers/Api** créé par la même occasion :



Avec évidemment le même code que celui qu'on a eu quand on avait déjà créé une ressource (et Laravel se débrouille tout seul pour les espaces de nom) :

```
<?php

namespace App\Http\Controllers\Api;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FilmController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        //
    }

    /**
     * Show the form for creating a new resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function create()
    {
        //
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
```

```
    public function store(Request $request)
    {
        //
    }

    /**
     * Display the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function show($id)
    {
        //
    }

    /**
     * Show the form for editing the specified resource.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function edit($id)
    {
        //
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  int  $id
     * @return \Illuminate\Http\Response
     */
    public function destroy($id)
    {
        //
    }
}
```

On va supprimer les méthodes **create** et **edit** qui n'ont pas lieu d'être pour une API.

On peut aussi directement générer un contrôleur sans ces deux méthodes avec l'option – ***-api***

Et on va coder les autres méthodes :

```
<?php

namespace App\Http\Controllers\Api;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
use App\Film;

class FilmController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        return Film::all();
    }

    /**
     * Store a newly created resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        Film::create($request->all());
    }

    /**
     * Display the specified resource.
     *
     * @param  \App\Film $film
     * @return \Illuminate\Http\Response
     */
    public function show(Film $film)
    {
        return $film;
    }

    /**
     * Update the specified resource in storage.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Film $film
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Film $film)
    {
        $film->update($request->all());
```

```
    }

    /**
     * Remove the specified resource from storage.
     *
     * @param  \App\Film $film
     * @return \Illuminate\Http\Response
     */
    public function destroy(Film $film)
    {
        $film->delete();
    }
}
```

Évidemment ça ressemble énormément à ce qu'on a fait précédemment et c'est normal puisqu'il s'agit du même genre de traitement sur les mêmes données.

Les routes

Pour les routes on adopte aussi une ressource dans le fichier **routes/api.php** :

```
Route::namespace('Api')->group(function() {
    Route::apiResource('films', 'FilmController');
});
```

Ce qui donne ces routes :

```
GET|HEAD  | api/films                 | films.index   | App\Http\Controllers\Api\FilmController@index   | api
POST      | api/films                 | films.store   | App\Http\Controllers\Api\FilmController@store   | api
GET|HEAD  | api/films/{film}          | films.show    | App\Http\Controllers\Api\FilmController@show    | api
PUT|PATCH | api/films/{film}          | films.update  | App\Http\Controllers\Api\FilmController@update  | api
DELETE    | api/films/{film}          | films.destroy | App\Http\Controllers\Api\FilmController@destroy | api
```

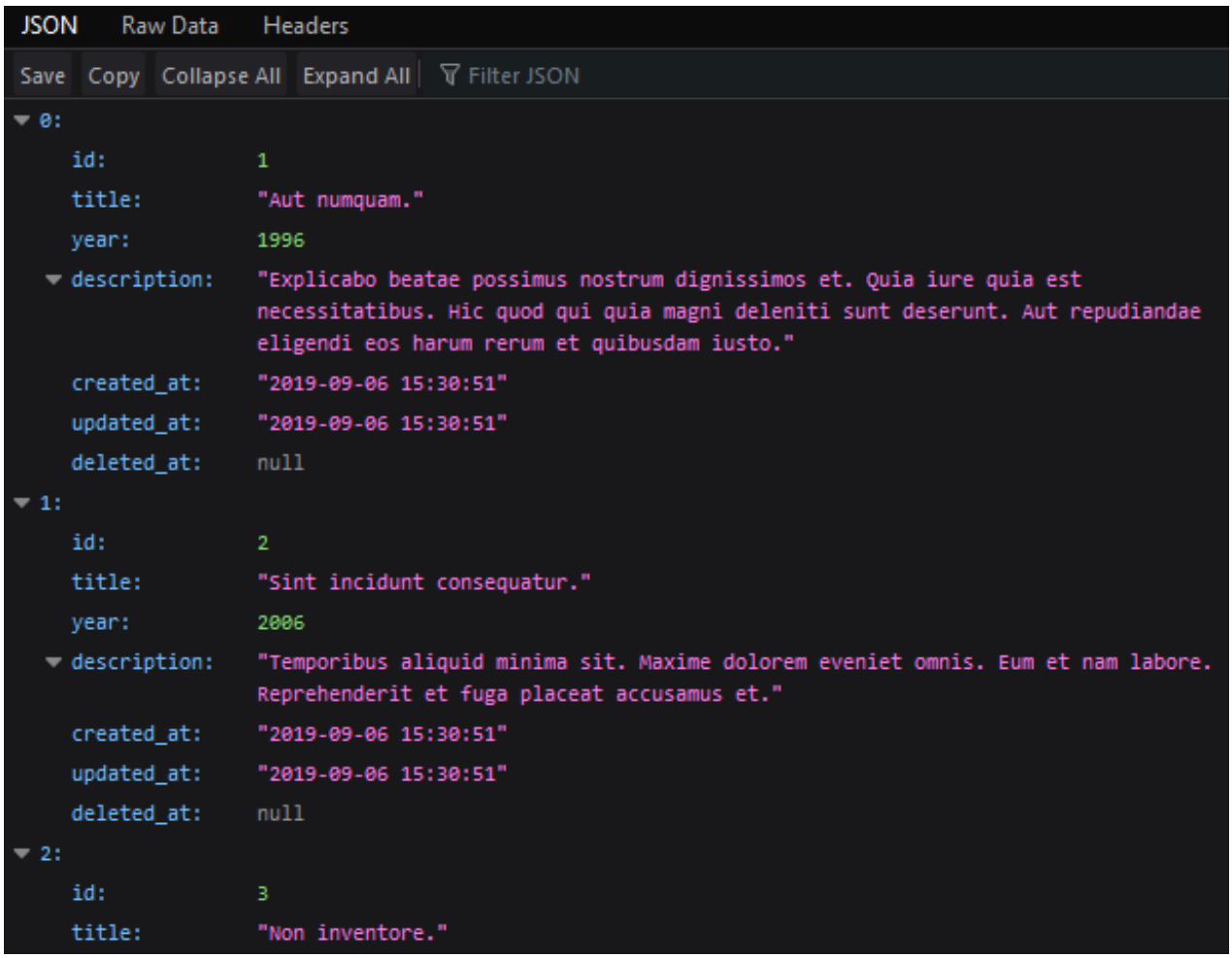
Le fait de déclarer ces routes dans le fichier **routes/api.php** a pour effet de leur appliquer le middleware **api** au lieu de **web**. Si vous regardez dans le fichier **app/Http/Kernel.php** vous allez voir la différence que ça génère :

```
protected $middlewareGroups = [
    'web' => [
        \App\Http\Middleware\EncryptCookies::class,
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
        \Illuminate\Session\Middleware\StartSession::class,
        // \Illuminate\Session\Middleware\AuthenticateSession::class,
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,
        \App\Http\Middleware\VerifyCsrfToken::class,
        \Illuminate\Routing\Middleware\SubstituteBindings::class,
    ],

    'api' => [
        'throttle:60,1',
        'bindings',
    ],
];
```

Comme on est sans état plus de cookie, de session, de protection CSRF…

Le fonctionnement

Maintenant avec l'url **...api/films** on obtient une réponse JSON (Laravel convertit automatiquement la réponse en JSON sans qu'on lui demande) :

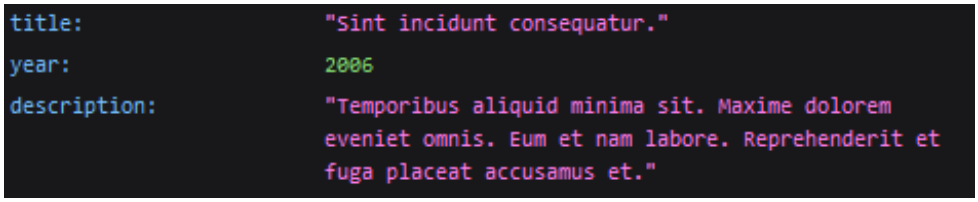


On obtient toutes les colonnes non cachées, c'est à dire celles qui ne sont pas définies dans la propriété **$hidden** du modèle. Et comme on n'a pas créé cette propriété on récupère toutes les colonnes. Alors on va juste renvoyer tire, année et description en ajoutant la propriétée dans le modèle **Film** :

```
protected $hidden = ['id', 'created_at', 'updated_at', 'deleted_at'];
```

*On dispose aussi de la propriété **$visible** qui est exactement l'inverse, on précise alors seulement les colonne qu'on veut voir.*

Avec l'url **.../api/films/2** on obtient :

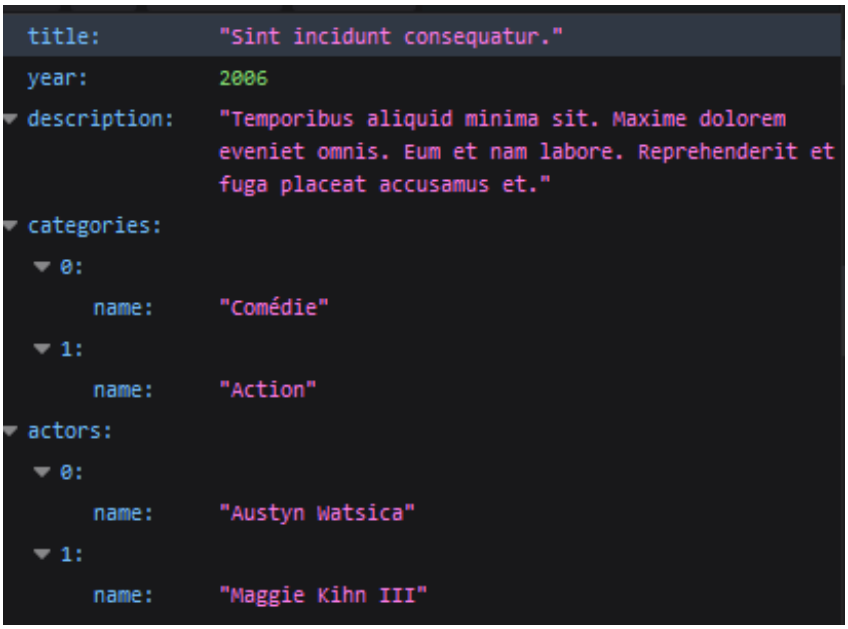


Mais pas seulement ! Comme pour notre applicationon a prévu le chargement automatique des catégories et des acteurs on les récupère également et avec toutes

leurs colonnes ! Comme ce n'est pas très judicieux on va ajouter cette ligne pour les deux modèles (**Actor** et **Category**) :

```
protected $visible = ['name'];
```

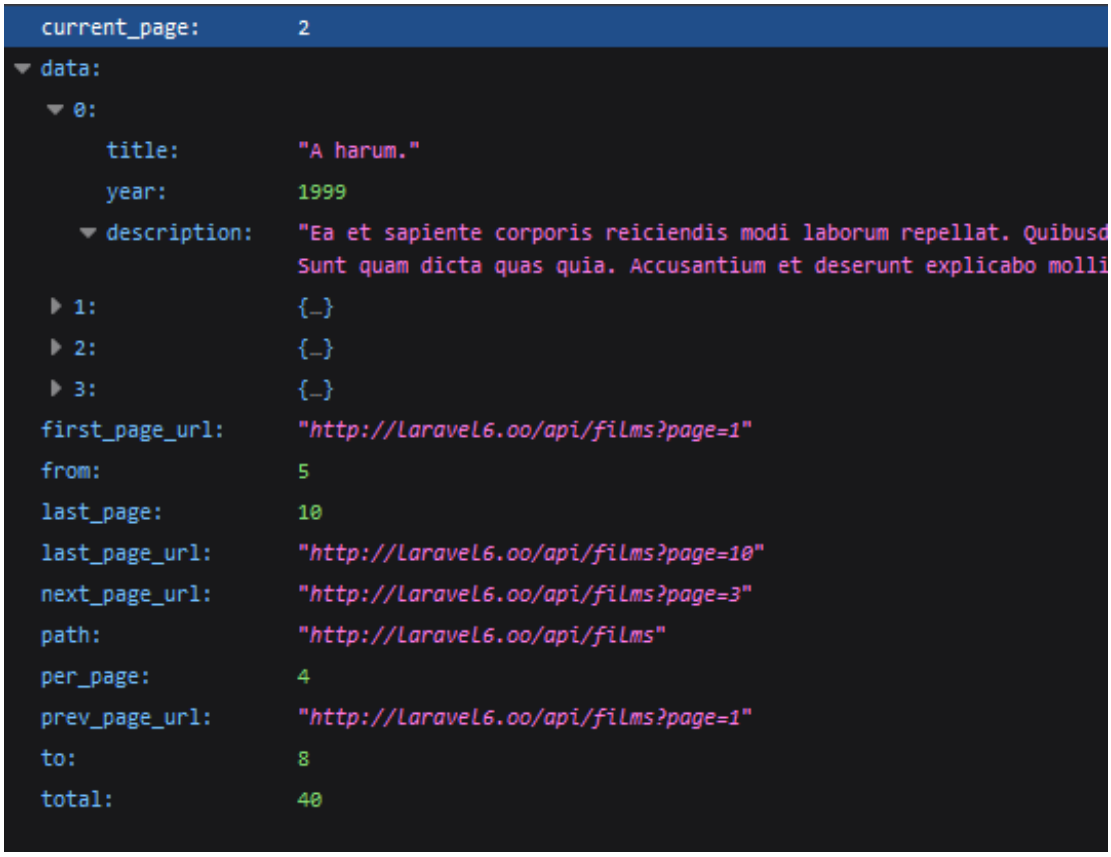
Maintenant on obtient ce résultat :



On pourrait aussi imaginer utiliser une pagination pour cette API. Il suffit de changer la méthode **index** :

```
public function index()
{
   return Film::paginate(4);
}
```

On peut alors demander la page qu'on veut, par exemple la seconde avec l'url **... api/films?page=2** :



On ne reçoit que les informations de la page 2 et en plus des renseignements concernant la pagination.

On peut imaginer aussi filtrer, ordonner…

Je ne traite pas dans ce chapitre de l'authentification spécifique aux API. Je vous invite à regarder la documentation de Passport sur le sujet.

Les ressources d'API

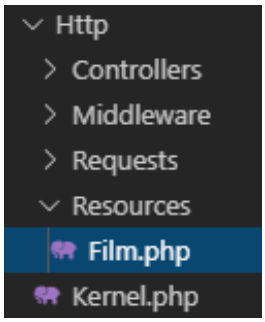
On a vu ci-dessus comme il est facile de retrouner une réponse JSON à partir des données d'un modèle. C'est parfait tant qu'on veut une simple sérialisation des valeurs des colonnes. Mais parfois on veut effectuer quelques traitements intermédiaires (ça correspond au design pattern transformer). Imaginons que nous voulons retourner seulement les 10 premiers mots de la description…

On utilise Artisan pour créer la ressource :

```
php artisan make:resource Film
```

On se retrouve avec la ressource ici :

Avec ce code :




```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class Film extends JsonResource
{
   /**
    * Transform the resource into an array.
    *
    * @param  \Illuminate\Http\Request  $request
    * @return array
    */
   public function toArray($request)
   {
      return parent::toArray($request);
   }
}
```

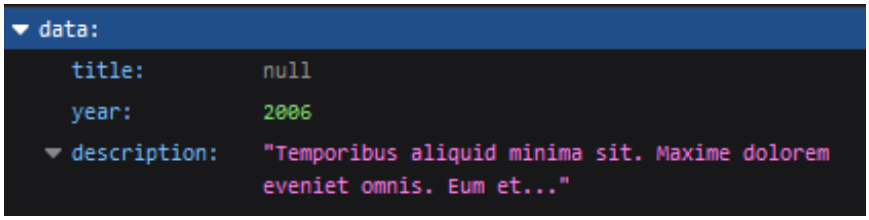
Là on se contente de tout sérialiser, ça ne va donc pas changer grand chose à ce qu'on avait précédemment. Dans le contrôleur on utilise la ressource, par exemple pour la méthode **show** :

```
use App\Http\Resources\Film as FilmResource;

...

public function show(Film $film)
{
   return new FilmResource($film);
}
```

Maintenant pour l'url **.../api/films/2** on obtient le même résultat que précédemment. Maintenant changeons la ressource :

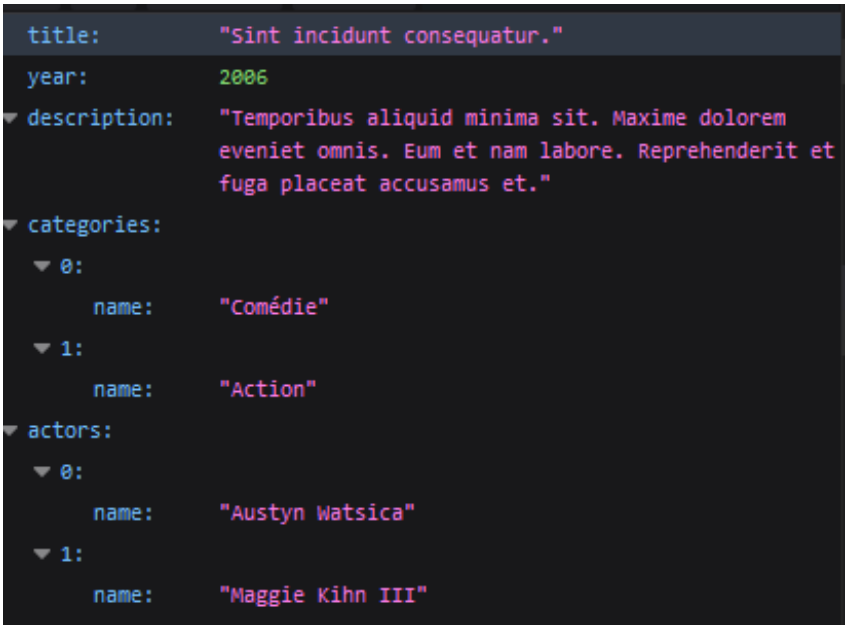
```
public function toArray($request)
{
   return [
      'title' => $this->name,
      'year' => $this->year,
      'description' => Str::words($this->description, 10),
   ];
}
```

Maintenant pour l'url **.../api/films/2** on obtient :



On voit qu'on n'a plus que les 10 premiers mots de la description. On a bien effectué un traitement intermédiaire sur les valeurs retournées. Mais on remarque aussi qu'on ne récupère plus les catégories et les acteurs ! On a juste ce qu'on a demandé. Mais personne ne nous empêche d'en demander plus :

```
public function toArray($request)
{
   return [
      'title' => $this->name,
      'year' => $this->year,
      'description' => Str::words($this->description, 10),
      'categories' => $this->categories,
      'actors' => $this->actors,
   ];
}
```



Et maintenant on récupère bien les éléments en relation.

Il y aurait encore beaucoup à dire sur les ressources et les API mais vous avez maintenant une bonne base de départ.

En résumé

> Laravel permet de créer facilement une API et de permettre des transformations avec des ressources.