**API Ressource**

Comment sérialiser nos modèles en JSON ? Où encore comment définir les champs à exposer dans notre modèle ce qui est très important, car lorsqu’on retourne un objet, automatiquement, **Laravel** va le convertir en JSON ce qui affichera tous les **champs** de notre model dans l’API.

**Création:**

**php artisan make:resource PropertyRessource**

Histoire de pouvoir tester nos implémentations concernant l’API, nous allons créer une nouvelle route **api /api/biens.** De ce fait nous ajouterons cette route dans **api.php** de même, il va nous falloir un controller pour gérer les actions de cette route que nous placerons dans le namespace **API.**

**php artisan make:controller API\\PropertyController**

Dans notre controller, dans la fonction index, nous allons retourner les 5 premiers biens.

public function index(){

return Property::limit(5)->get();

}

Nous remarquons que cela sera formaté en json et va afficher tous les champs.

Si on souhaite convertir l’ensemble de nos biens en utilisant cette ressource, nous aurons:

public function index(){

return PropertyRessource::collection(Property::limit(5)->get()) ;

}

}

Cela créera une clé **data** et à l’intérieur de cette clé **data,** nous aurons l’ensemble de nos ressources.

La petite particularité est que dans notre ressource nous avons une methode **toArray()** cette méthode nous permettra de retourner les champs que l’on veut afficher.

public function toArray(Request $request): array

{

return parent::toArray($request);

}

Pour définir les champs à afficher on peut procéder comme suit:

public function toArray(Request $request): array

{

return [

'id' =>$this->id,

'title' => $this->title

];

}

Nous pouvons accéder directement aux propriétés de nos biens à travers cette instance. Cela se fait de façon magique. Une approche serait d’utiliser la **propriété resource,** propriété qui contient notre model; nous pouvons le voir en faisant **$this->resource.** et d'accéder aux propriétés via.

/\*\*

\* Transform the resource into an array.

\*@property Property $resource Cela permet à notre éditeur de comprendre que

\* cette propriété est de type Property

\* @return array<string, mixed>

\*/

public function toArray(Request $request): array

{

return [

'id' =>$this->resource->id,

'title' => $this->resource->title

];

}

Nous avons également la possibilité de sortir les relations liées.

Prenons un exemple pour la relatio **option:**

Nous pouvons avoir :

return [

'id' =>$this->resource->id,

'title' => $this->resource->title,

'options' => $this->resource->options

];

Cela va nous afficher toutes les options, effectivement parce qu’on a pas défini ce que l’on souhaite exposer. Donc on peut créer une **ressource OptionResource** et décider d’afficher juste l’**id** et le **name.** Nous aurons dans notre PropertyRessource finalement :

public function toArray(Request $request): array

{

return [

'id' =>$this->resource->id,

'title' => $this->resource->title,

'options' => OptionRessource::collection($this->resource->options)

];

}

Nous pouvons également conditionner certains affichages, par exemple vérifier si l’**user**  a le droit de faire quelque chose ou pas et afficher un champ en fonction de cela. De ce fait, vu qu'on a accès à la propriété **$request,** nous pouvons directement récupérer la session de l’**user** et vérifier s’il peut faire l’action avec la méthode **can().**

**$request->user()->can()....**

Nous pouvons également utiliser la méthode **when** pour conditionner l’affichage d’un champ. Cela prend la condition en premier paramètre et le champs en 2ème.

return [

'id' =>$this->resource->id,

'title' => $this->resource->title,

'price' =>$this->when(false, $this->resource->price),

'options' => OptionRessource::collection($this->resource->options)

Nous avons pris l’exemple avec la propriété **‘price’.**

Nous pouvons voir que le **price** est à **false**, il ne s’affiche pas mais à **true** il s’affiche dans le **JSON.** Nous avons pleines de methodes que nous pouvons tester comme : **whenHas, mergeWhen …**

De plus, la méthode **mergeWhen** nous permet d’envoyer un tableau si la condition est vérifiée. Exemple:

public function toArray(Request $request): array

{

return [

'id' =>$this->resource->id,

'title' => $this->resource->title,

$this->mergeWhen(false, [

'price' => $this->resource->price,

'surface' => $this->resource->surface

]),

'options' => OptionRessource::collection($this->resource->options)

];

}

***info:*** *en ce qui concerne les relations, les sous-ressources, il serait judicieux de ne pas les afficher directement mais de vérifier au paravant si elles sont loader bien avant. Exemple avec* ***option:***

***'options' => OptionRessource::collection($this->whenLoaded('options'))***

*De la sorte, cette relation ne sera chargée que si nous faisons de l’****eager-loading.***

*public function index(){*

*return PropertyRessource::collection(Property::limit(5)->with('options')->get()) ;*

*}*

Nous pouvons également sérialiser notre model sur une seule entrée en particulier et nous lui passons notre objet en paramètre: Exemple:

public function index(){

return new PropertyRessource(Property::find(1)) ;

}

Si nous voulons qu’il change le nom de clé dans notre **JSON** qui est **$data,** nous pouvons le faire en déclarant dans notre **PropertyRessource** une propriété static **wrap = ‘nomclé’.**

Nous pouvons également lui dire de nous faire un retour de pagination.

public function index(){

return Property::paginate(5);

Cela nous affichera même les liens des url des pages suivantes. Cela est très intéressant si nous voulons construire un système de pagination côté client.

Nous pouvons également les retourner sous forme de collections.

return PropertyRessource::collection(Property::paginate(5)) ;

Autre notion intéressante à connaître, nous avons les ressources de type **collection.**

Cela va nous permettre de spécifier comment on va renvoyer nos collections d’objets.

**Pour générer:**

**php artisan make:resource PropertyCollection**

**NB:** si je n’avais pas mis de suffixe collection, j’aurais été obligé de mettre un drapeau devant le nom de la collection : **–collection**

Dans ce cas, notre **PropertyCollection** implémente le **RessourceCollection** à la différence de **JsonRessource** pour **PropertyRessource.**

Autre chose à noter est que là nous avons accès à la propriété **collection** au lieu de **resource:**

**return [**

**'data' => $this->collection,**

**'links' => [**

**'url' => 'https://gts-tech.com'**

**]**

**];**

de ce fait dans notre controller au lieu de retourner les données par **PropertyRessource** nous le ferons à travers **PropertyCollection.**

**public function index(){**

**//return PropertyRessource::collection(Property::limit(5)->with('options')->get());**

**return new PropertyCollection(Property::limit(5)->with('options')->get()) ;**

**return PropertyRessource::collection(Property::paginate(5)) ;**

**}**

Autre chose très intéressante est, lorsque l’on travaille avec une collection, elle recherche automatiquement la ressource liée, raison pour laquelle il est important de respecter les conventions de nommage, car en cherchant, il enlèvera juste le suffixe collection à la fin du nom de la collection pour voir s’il y a une ressource correspondante.

Si on a plusieurs **ressources**, nous pouvons lui préciser laquelle on veut utiliser à travers une propriété particulière:

**public $collect = NomResource::class**

Si nous ne voulons pas qu’il nous mette le préfixe **data** lors de l’affichage de la collection, nous pouvons le faire dans le **AppServiceProvider,** dans la méthode **boot()** nous rajoutons cela.

/\*\*

\* Bootstrap any application services.

\*/

public function boot(): void

{

Paginator::useBootstrapFive();

//Nous rajoutons cela

JsonResource::withoutWrapping();

}

Il faut noter que même nous notons cela, si nous avons défini un **wrap** comme dans notre cas où nous avons mis :

public static $wrap = 'property';

ou dans notre collection où nous avons défini le ‘**data’.**

**'data' => $this->collection,**

**Conclusion sur l'utilisation des API Resources dans Laravel**

L'utilisation des API Resources dans Laravel est une pratique puissante et flexible pour sérialiser les modèles en JSON. Cela permet un contrôle granulaire sur les données exposées via l'API, améliorant la sécurité, la performance, et la clarté du code. Voici les points clés et les conclusions que nous pouvons tirer :

**\*\*1. Contrôle de la Sérialisation :\*\***

- En utilisant les API Resources, vous pouvez définir exactement quels champs de votre modèle doivent être exposés via l'API. Cela évite de divulguer des informations sensibles ou inutiles.

- La méthode `toArray()` permet de personnaliser les champs renvoyés, ce qui offre une flexibilité maximale.

**\*\*2. Gestion des Relations :\*\***

- Les relations peuvent être sérialisées en utilisant des sous-ressources, ce qui permet de structurer les données de manière hiérarchique et propre.

- Il est essentiel d'utiliser la méthode `whenLoaded` pour éviter les requêtes supplémentaires inutiles, optimisant ainsi les performances.

**\*\*3. Conditionnement de l'Affichage :\*\***

- Les méthodes telles que `when` et `mergeWhen` permettent d'afficher conditionnellement certains champs en fonction des permissions ou autres critères logiques. Cela améliore la sécurité et l'efficacité de l'API.

**\*\*4. Collections de Ressources :\*\***

- Les `ResourceCollections` permettent de définir comment les collections d'objets doivent être sérialisées. Cela est particulièrement utile pour les réponses paginées et les listes d'objets.

- En utilisant des collections, vous pouvez ajouter des métadonnées telles que des liens de pagination ou des URL personnalisées.

**\*\*5. Pagination et Ressources Individuelles :\*\***

- L'utilisation de `paginate()` avec les API Resources permet de structurer les réponses paginées de manière cohérente et facile à utiliser pour les clients de l'API.

- Vous pouvez également sérialiser des modèles individuels de manière cohérente en utilisant les API Resources.

**\*\*6. Personnalisation Globale :\*\***

- Vous pouvez désactiver le préfixe `data` pour toutes les réponses JSON en utilisant `JsonResource::withoutWrapping()`. Cela vous donne plus de contrôle sur la structure de la réponse JSON.

**Recommandations et Bonnes Pratiques**

**\*\*1. Utiliser les API Resources par Défaut :\*\***

- Pour toute API, il est recommandé d'utiliser les API Resources pour contrôler la sortie JSON. Cela garantit que seules les données nécessaires sont exposées.

**\*\*2. Respecter les Conventions de Nommage :\*\***

- Respecter les conventions de nommage facilite la maintenance et l'évolution du code, surtout lorsque l'on travaille avec des collections et des sous-ressources.

**\*\*3. Optimiser les Requêtes avec Eager Loading :\*\***

- Utiliser `with()` pour charger les relations nécessaires en une seule requête, et `whenLoaded` pour conditionner leur inclusion dans la réponse JSON.

**\*\*4. Tester les Implémentations :\*\***

- Tester systématiquement les implémentations API pour s'assurer que les données renvoyées sont correctes, complètes, et sécurisées.

**\*\*5. Documentation et Clarté du Code :\*\***

- Documenter les API Resources et les contrôleurs pour faciliter la compréhension du code et la collaboration avec d'autres développeurs.

En conclusion, l'utilisation des API Resources dans Laravel permet de créer des APIs robustes, sécurisées, et faciles à maintenir. En adoptant les bonnes pratiques mentionnées, vous pouvez améliorer significativement la qualité et la performance de vos applications web.