# Les API avec API-platform

## Préambule

Pour mettre en place notre API et comprendre son fonctionnement, nous allons créer un système de commentaires sur les produits de notre base.

Pour cela, créez une nouvelle table dans la base Northwind, que l'on nommera `comments`.

Cette table comprendra les propriétés suivantes :

- content (text)
- date (datetime)
- updateDate (datetime)

Ajouter ensuite une relation entre les 2 tables, en partant du principe qu'un produit peut avoir plusieurs commentaires, mais qu'un commentaire est associé à un seul et unique produit.

De la même façon, une relation doit être établie entre la table `user` et la table `comment` : un utilisateur peut poster plusieurs commentaires, mais un commentaire est posté par un seul et unique utilisateur.

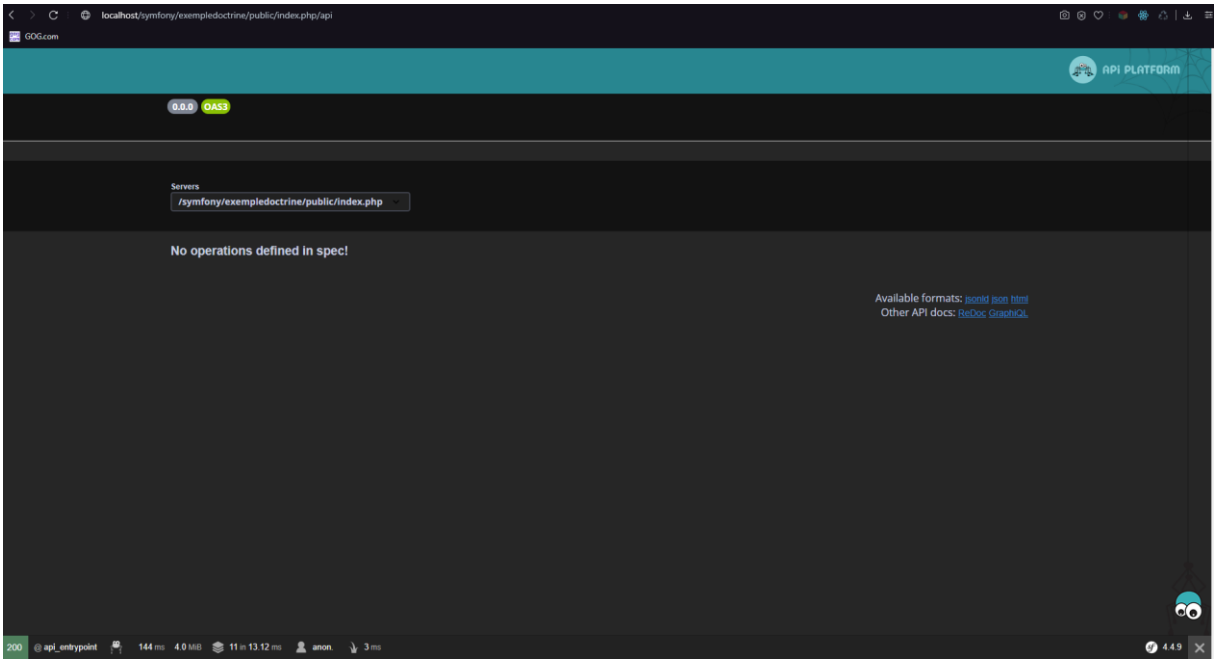
Organisez votre crud que l'on puisse ajouter un commentaire sur la page de détail du produit.

## Installation d'API-Platform

Pour installer API-Platform, ouvrez la console à la racine du projet et tapez la commande suivante :

```
composer req api
```

Pour vérifier la bonne installation du composant, rendez-vous sur la route `/api`. Vous devriez avoir l'affichage suivant :




C'est ici que vous aurez accès à la documentation de votre api.

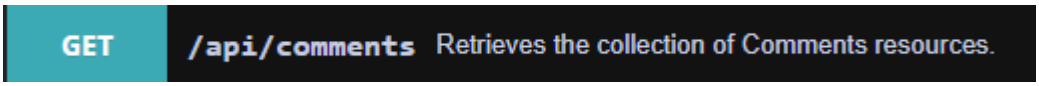
Il nous faut maintenant définir un point d'entré sur l'entité `Comments` pour pouvoir récupérer les commentaires. Nous allons pour cela utiliser les annotations.

Dans les annotations de la classe `Comments`, nous allons ajouter l'annotation `@ApiRessource()`.

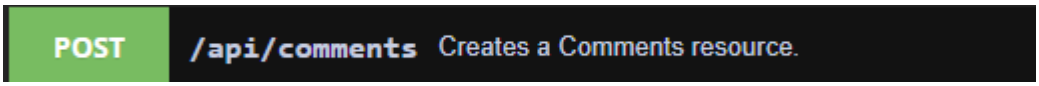
```php
<?php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;
use App\Repository\CommentsRepository;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass=CommentsRepository::class)
 * @ApiResource()
 */
class Comments
{
    ...
}
```

Actualisons la documentation de notre API.

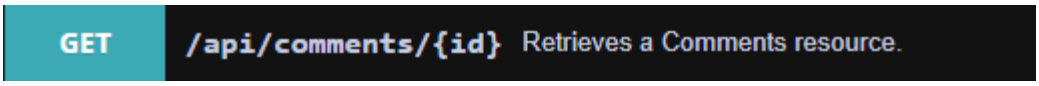
Vous y trouverez les différentes méthodes d'accès à nos commentaires :

- 

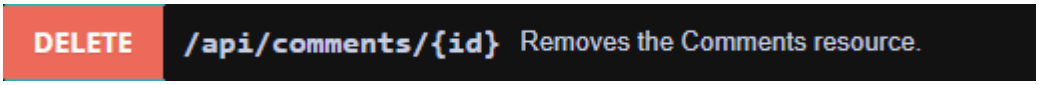

Va permettre de récupérer tous les commentaires.

- 

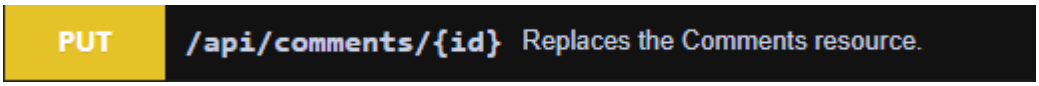

Permet de créer un commentaire.

- 

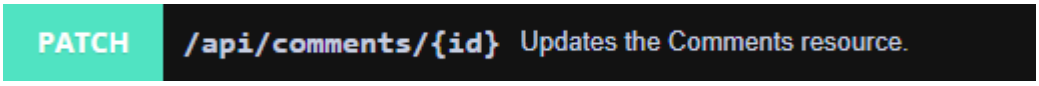

Permet de récupérer un commentaire en particulier.

- 


Permet de supprimer un commentaire.

- 


Permet de modifier un commentaire.

- 


Permet enfin de modifier un champ en particulier.

Vous avez la possibilité de filtrer les points d'entrés en passant des paramètres dans l'annotation :

- `collectionOperations={""}` permet de filtrer les méthodes sur tous les commentaires.

- `itemOperations={""}` permet de filtrer les méthodes sur un commentaire en particulier.

Par exemple, nous voulons uniquement garder les méthodes pour récupérer tous les commentaires, ainsi qu'un unique commentaire. Nous noterons les paramètres comme ceci :

```
/**
 * @ORM\Entity(repositoryClass=CommentsRepository::class)
 * @ApiResource(
 *      collectionOperations={"get"},
 *      itemOperations={"get"}
 * )
 */
```

De la même manière, nous pouvons uniquement garder les suppressions, ajout, etc.

## Test et configuration de l'API

Sur la doc de notre API, si nous cliquons sur la première ligne, nous allons retrouver un exemple de réponse, entièrement documentée à partir de l'entité.

En cliquant sur `Try it out`, on remarque que l'API récupère tous les champs. Pour éviter que toutes les données soient exposées, nous avons la possibilité de créer des groupes permettant de sélectionner les données que nous voulons récupérer.

Pour cela, il suffit de préciser dans les annotations de l'entité de cette façon les données que nous souhaitons récupérer :

```
/**
* @ORM\Column(type="text"})
* @Groups({"read:comment"})
*/
```

L'annotation `@Groups({"read:comment"})` permet de définir un groupe pour une propriété, et ici nous avons nommé ce groupe `read:comment`.

Pour afficher les groupes définis, il suffit de le préciser dans l'annotation de la classe :

```
/**
 * @ORM\Entity(repositoryClass=CommentsRepository::class)
 * @ApiResource(
 *      normalizationContext={"groups"={"read:comment"}},
 *      collectionOperations={"get"},
 *      itemOperations={"get"}
 * )
 */
```

### Application

Définissez un groupe pour les différents champs que vous voulez récupérer sur votre entité et observez le résultat sur l'API.

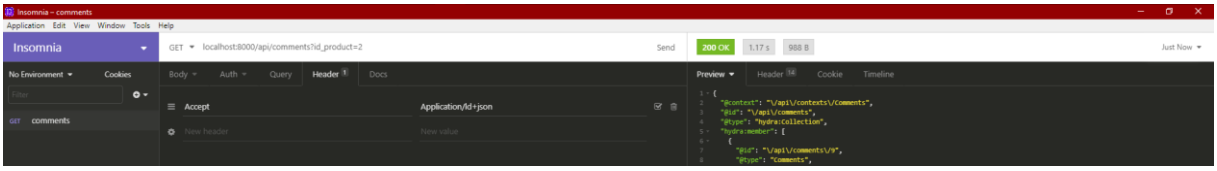
A l'aide de la documentation, afficher les résultats par date d'ajout décroissante.

## Alternative

Pour tester votre API, vous avez la possibilité d'utiliser un logiciel tiers (Postman, Insomnia). L'utilisation d'un autre logiciel permet plus de manipulation et de personnalisation.

Pour la suite, nous utiliserons Insomnia.

A l'aide de ce logiciel et de sa documentation, afficher le résultat précédent.




## Mettre en place un filtre de recherche

Si on veut filtrer l'affichage des commentaires selon un article, nous allons changer l'url de cette manière :

```
localhost:8000/api/comments?products=2
```

si on exécute cette requête, on obtient un résultat erroné. Il nous faut au préalable définir un filtre.

Pour cela, nous allons ajouter un filtre de recherche dans les annotations de la classe (entité) `Comments` :

```
@ApiFilter(SearchFilter::class, properties={"product": "exact"})
```

Si vous avez une erreur, n'oubliez l'import de `SearchFilter`.

Dans insomnia testez avec la requête suivante :

```
localhost:8000/api/comments?product=2
```

(Si vous utiliser le serveur Symfony, adaptez si vous utiliser un autre type de serveur, de même pour `id_product`, choisissez-en où il y a plusieurs commentaires).

Vous avez aussi la possibilité de créer vos propres filtres.

## Pagination

Pour éviter un trop gros affichage d'informations, API-platform nous propose un système de pagination.

Pour le mettre en place, il suffit d'ajouter dans l'annotation `@ApiRessource` :

```
/**
* @ApiRessource(
*     paginationItemsPerPage=2
* )
*/
```

Testez encore une fois pour observer le résultat. Nous avons bien 2 commentaire qui sont récupérés par page. En remplaçant `application/json` par `application/ld+json`. Nous obtenons des informations supplémentaires, comme le nombre de page, la page suivante et la page précédente.

Pour accéder à la page suivante, utiliser l'url suivante (toujours dans Insomnia) :

```
localhost:8000/api/comments?id_product=2&page=2.
```

## Relation

Il serait intéressant de pouvoir récupérer le nom de l'utilisateur qui a poster le commentaire.

Pour cela, rendons-nous dans l'entité `User.php`.

Dans les annotations de la propriété `email`, nous allons ajouter le même groupe que pour les propriétés de l'entité `Comments`.

Testez pour obtenir le résultat suivant :

```
[
  {
    "id": 2,
    "content": "string",
    "date": "2020-06-23T14:30:51+00:00",
    "user": {
      "id": 2,
      "email": "azerty@aze.rty"
    }
  },
  {
    "id": 9,
    "content": "test2",
    "date": "2020-06-19T08:30:04+00:00",
    "user": {
      "id": 2,
      "email": "azerty@aze.rty"
    }
  }
]
```

## Récupération d'un commentaire selon son id

Toujours dans Insomnia, nous allons dupliquer la requête précédente et nous la renommons `comments/{id}`. Toujours en gardant la méthode `GET`, nous modifions l'url de cette façon : `localhost:8000/api/comments/2` (2 étant l'id du commentaire demandé).

Ici, on pourrait vouloir retrouver le nom du produit sur lequel le commentaire a été mis.

Pour cela, nous allons recréer un autre groupe dans l'annotation de la propriété `product` et sur la propriété `product_name` de l'entité `Products.php`, par exemple : `@Groups({"read:comment:full"})`.

Il suffit ensuite de changer le contexte de normalisation dans `@ApiRessource` de la manière suivante :

```
/**
 * @ORM\Entity(repositoryClass=CommentsRepository::class)
 * @ApiResource(
 *     itemOperations={
 *      "get"={
 *          "normalization_context"={"groups"={"read:comment", "read:commen
t:full"}}
 *          }
 *     }
 * )
 */
```

Cette méthode permet de définir et d'appeler différents groupes selon les besoins.

Il était possible, sans changer le contexte de normalisation, de simplement ajouter le groupe `@Groups({"read:comment"})` sur la propriété `product_name` de l'entité `Products.php`. Tout dépend de vos besoins...

## Remarque

Si vous utiliser le même groupe sur plusieurs entités (par exemple `User` et `Comments`) vous risquez de rencontrer cette erreur :


```
    "type": "https:\/\/tools.ietf.org\/html\/rfc2616#section-10",
    "title": "An error occurred",
    "detail": "The total number of joined relations has exceeded the specified maximum. Raise the limit if necessary with the \"api_platform.eager_loading.max_joins\" configuration key (https:\/\/api-platform.com\/docs\/core\/performance\/#eager-loading), or limit the maximum serialization depth using the \"enable_max_depth\" option of the Symfony serializer (https:\/\/symfony.com\/doc\/current\/components\/serializer.html#handling-serialization-depth).",
    "trace": [
```


Ce qu'il se passe :

- Vous appelez un commentaire, qui comprend ses informations plus celle de l'utilisateur.
- Dans les informations de l'utilisateur nous retrouvons les différents commentaires qu'il a posté.
- Dans ces commentaires on retrouve les informations de l'utilisateur.
- Etc ...

Pour pallier à ce problème, il faut indiquer le nombre maximal de répétition acceptée.

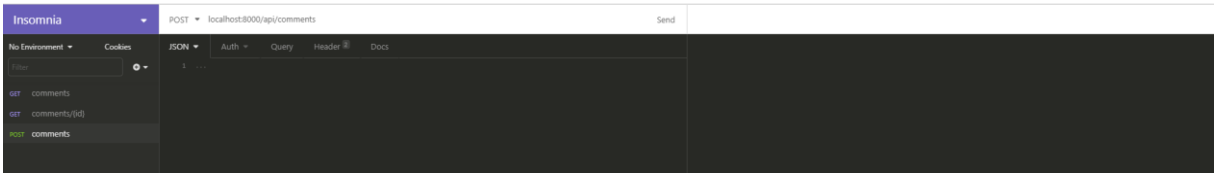
On doit ajouter, à `@ApiResource`, `"enable_max_depth"=true`, et ajouter `@MaxDepth(1)` en annotation de la propriété concernée de l'entité `User`.

Vous devriez voir apparaître cette notation : `"\/api\/comments\/2"`.

On appelle ceci une IRI. C'est l'adresse qui permet d'identifier une ressource.

## Ajout d'un commentaire

Commençons par dupliquer une des méthodes déjà existantes pour avoir le résultat suivant :




Pour rappel, POST permet de faire un ajout.

Pour tester un ajout de commentaire, placez-vous dans la partie Json et entrez le code suivant :

```
{
 "content" : "Un nouveau commentaire",
 "date" : "2020-06-24",
 "user" : "/api/users/1"
}
```

Pour `User`, nous utilisons l'IRI de notre objet. Symfony s'attend à ce que nous lui fournissions un objet pour l'utilisateur.

Attention : si vous utilisez des groupes différents pour vos propriétés, n'oubliez pas d'insérer la ligne suivant dans `@ApipResource` :

```
collectionOperations={
       "post"={
          "normalization_context"={"groups"={"read:comment",
"read:comment:full"}}
     }
   }
```

De plus, pour que vos différentes entités soient reconnues, il faut ajouter dans chacune d'elle `@ApiResource()`.

Vous devriez obtenir, dans la fenêtre de droite, un résultat semblable à celui-ci :

```
{
  "id": 24,
  "content": "Un nouveau commentaire",
  "date": "2020-06-24T00:00:00+00:00",
  "product": {
    "ProductName": "jhkghjkghj"
  },
  "user": {
    "id": 1,
    "email": "toto@toto.com"
  }
}
```

Il est également possible de récupérer l'utilisateur directement par notre application Symfony (puisque que notre API est directement générée et liée à cette dernière) par l'intermédiaire d'un contrôleur.

Pour cela, créons un nouveau contrôler : `/Controller/Api/CommentCreateController.php`.

Dans ce contrôleur, nous allons commencer par récupérer le composant `security` (qui permet de récupérer l'utilisateur) :

```
 private $security;
    // récupération de l'utilisateur
    public function __construct(Security $security)
    {
        // récupération de la connection
        $this->security = $security;
    }
```

Puis nous allons pouvoir ensuite définir que l'utilisateur connecté est celui qui a émis le `POST` :

```
 public function __invoke(Comments $data) // $data => donnée envoyé à l'API
    {
        // Définissions de l'utilisateur qui a émis le commentaire comme ét
ant celui qui est connecté
       $data->setUser($this->security->getUser());
       return $data;
    }
```

ATTENTION `$data` ne peut être changé. Ce paramètre regroupe les données qui transit. En outre, il est possible de le typer. Ici, on précise qu'on attend un commentaire.

Maintenant il faut préciser à notre entité que lorsqu'un `POST` est fait, il faut utiliser ce contrôleur.

Dans `CollectionOperations`, `"post"={}`, nous allons ajouter la ligne suivante :

```
*  "controller"=App/Controller/Api/CommentCreateController::class
```

Testez le résultat avec l'exemple précédent, sans spécifier la propriété `user`.

## Modification d'un commentaire

Commençons par définir le contexte de normalisation permettant la modification. Pour cela, rendons-nous sur notre entité, et définissez un groupe pour chacune des propriétés que vous avez à modifier. Si un groupe est déjà défini sur cette propriété, vous pouvez noter les choses de cette façon :

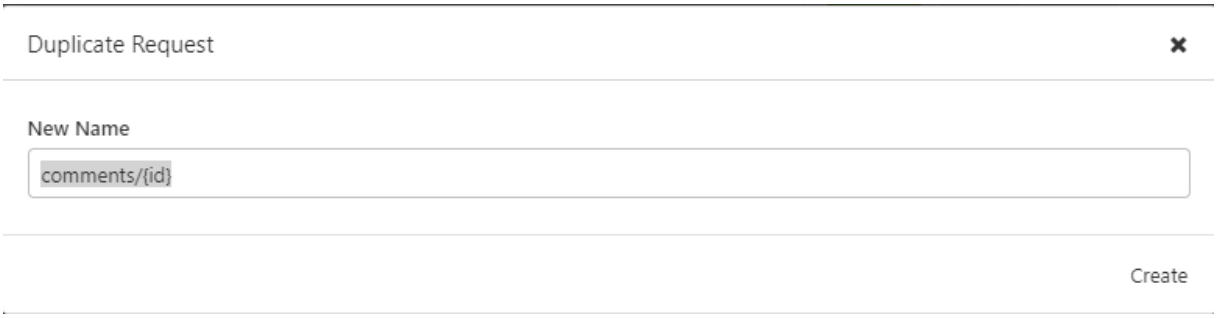
```
/**
     * @ORM\Column(type="text", nullable=true)
     * @Groups({"read:comment", "update:comment"})
     */
    private $content;
```

Ensuite définissons le contexte de normalisation dans `@APIRessource` :

```
/**
 * @ORM\Entity(repositoryClass=CommentsRepository::class)
 * @ApiResource(
 *      itemOperations={
 *         "put"={
 *          "denormalization_context"={"groups"="update:comment"}
 *         },
 *      },
 */
```

`denormalization_context` permet de permet de préciser ce qui est autorisé.

Nous pouvons maintenant dupliquer une des méthodes précédemment créées sur Insomnia.




Changeons la méthode d'accès par `PUT`. `PUT` permet la modification d'une entrée de la base.

Pour tester cette méthode, cliquez sur l'onglet `JSON` et testez le code suivant :

```
{
 "content" : "modification du commentaire",
 "edit": "2020-07-07"
}
```

## Exercice

En suivant les indications pour ajout, lecture, modification, et avec l'aide de la documentation, paramétrez l'entité pour la suppression d'une entrée dans la base de données.

## Sécurité

Maintenant que les bases de l'API sont en place, il serait intéressant de sécuriser le tout, evitant ainsi qu'une personne non connectée puisse ajouter un commentaire, ou qu'une personne connectée puisse modifier le commentaire de quelqu'un d'autre.

La première méthode consiste à ajouter une clé à `"post"={}` et à lui préciser quelle(s) autorisation(s) doit(vent) être prise(s) en compte pour pouvoir faire un `POST`.

Dans le cas d'un ajout de commentaire (pour notre exemple), nous voulons simplement que l'utilisateur soit connecté. Pour cela, nous allons donc inscrire :

```
*     collectionOperations={
*       "post"={
*          "security"="is_granted('IS_AUTHENTICATED_FULLY', object)"
*           }
*     }
```

La seconde méthode est d'utiliser un Voter. Cela va nous permettre de définir certaines autorisations pour des actions précises.

Prenons le cas d'une modification. Il ne faut pas qu'un utilisateur puisse modifier le commentaire d'un autre utilisateur.

### Application

En suivant la ressource sur les voters, établissez un voter pour filtrer la modification. Vous devrez y vérifier que l'utilisateur est bien une instance de `users`, et que `$subject` est une instance de `Comments`.

Une fois le voter établi, il ne reste plus qu'à ajouter une clé `security` de la même façon que pour la méthode `POST` :

```
 * "security"="is_granted('EDIT_COMMENT', object)"
```

(Si vous utiliser le serveur Symfony, adaptez si vous utiliser un autre type de serveur, de même pour `id_product`, choisissez-en où il y a plusieurs commentaires).

Vous avez aussi la possibilité de créer vos propres filtres.

## Pagination

Pour éviter un trop gros affichage d'informations, API-platform nous propose un système de pagination.

Pour le mettre en place, il suffit d'ajouter dans l'annotation `@ApiRessource` :

```
/**
* @ApiRessource(
*     paginationItemsPerPage=2
* )
*/
```

Testez encore une fois pour observer le résultat. Nous avons bien 2 commentaire qui sont récupérés par page. En remplaçant `application/json` par `application/ld+json`. Nous obtenons des informations supplémentaires, comme le nombre de page, la page suivante et la page précédente.

Pour accéder à la page suivante, utiliser l'url suivante (toujours dans Insomnia) : `localhost:8000/api/comments?id_product=2&page=2`.

## Relation

Il serait intéressant de pouvoir récupérer le nom de l'utilisateur qui a poster le commentaire.

Pour cela, rendons-nous dans l'entité `User.php`.

Dans les annotations de la propriété `email`, nous allons ajouter le même groupe que pour les propriétés de l'entité `Comments`.

Testez pour obtenir le résultat suivant :


```
[
  {
    "id": 2,
    "content": "string",
    "date": "2020-06-23T14:30:51+00:00",
    "product": {
      "id": 7,
      "ProductName": "jhkghjkghj",
      "picture": "7.jpeg"
    },
    "user": {
      "id": 2,
      "email": "azerty@aze.rty"
    }
  },
  {
    "id": 9,
    "content": "test2",
    "date": "2020-06-19T08:30:04+00:00",
    "product": {
      "id": 2,
      "ProductName": "fghfdgh",
      "picture": "2.jpeg"
    },
    "user": {
      "id": 2,
      "email": "azerty@aze.rty"
    }
  }
]
```