DOCUMENTATION API REST (Symfony API Platform)

I. API REST

1. Définition d'API

En informatique, API est l'acronyme d'Application Programming Interface, que l'on traduit en français par interface de programmation applicative. L'API peut être résumée à une solution informatique qui permet à des applications de communiquer entre elles et de s'échanger mutuellement des services ou des données. Il s'agit en réalité d'un ensemble de fonctions qui facilitent, via un langage de programmation, l'accès aux services d'une application.
Dans le domaine d'internet, l'API permet aux développeurs de pouvoir utiliser un programme sans avoir à se soucier du fonctionnement complexe d'une application. Les API peuvent par exemple être utilisées pour déclencher des campagnes publicitaires d'e-mailing de façon automatique sans avoir à passer par la compréhension d'une telle application (c'est le cas avec l'API AdWords de Google, par exemple). On les retrouve aujourd'hui dans de nombreux logiciels, en particulier dans les systèmes d'exploitation, les serveurs d'applications, dans le monde du graphisme, les bases de données, l'open data, etc.
Il existe deux grands protocoles de communication sur lesquels s'adossent les API : Simple Object Access Protocol (SOAP) et Representational State Transfer (REST). Le second s'est désormais largement imposé face au premier car il est plus flexible. Il a donné naissance aux API dites REST ou RESTful.

2) Protocole HTTP

a) Définition

L'http(Hypertext Transfer Protocol) désigne dans le langage informatique un protocole de communication entre un client et un serveur pour le World Wide Web. On le traduit littéralement en français par protocole de transfert hypertexte.

Inventé dans les années 1990 par Tim Berners-Lee, le protocole http établit une liaison entre un ordinateur (client) et un serveur Web. Le premier, via un navigateur Web, envoie une requête au second qui lui apporte une réponse presque instantanée. Autrement dit, le protocole de communication http est ce qui permet à un internaute d'accéder à un contenu (une page Web, un fichier CSS, etc.), en commandant au serveur d'effectuer une action.

b) Fonctionnement

- Verbes:

HTTP définit un ensemble de méthodes de requête qui indiquent l'action que l'on souhaite réaliser sur la ressource indiquée. Bien qu'on rencontre également des noms (en anglais), ces méthodes sont souvent appelées verbes HTTP. Chacun d'eux implémente une sémantique différente mais certaines fonctionnalités courantes peuvent être partagées par différentes méthodes (e.g. une méthode de requête peut être sûre (safe), idempotente ou être mise en cache (cacheable)).

GET
La méthode GET demande une représentation de la ressource spécifiée. Les requêtes GET doivent uniquement être utilisées afin de récupérer des données.
HEAD
La méthode HEAD demande une réponse identique à une requête GET pour laquelle on aura omis le corps de la réponse (on a uniquement l'en-tête).
POST
La méthode POST est utilisée pour envoyer une entité vers la ressource indiquée. Cela entraîne généralement un changement d'état ou des effets de bord sur le serveur.
PUT
La méthode PUT remplace toutes les représentations actuelles de la ressource visée par le contenu de la requête.

DELETE
La méthode DELETE supprime la ressource indiquée.
CONNECT
La méthode CONNECT établit un tunnel vers le serveur identifié par la ressource cible.

OPTIONS
La méthode OPTIONS est utilisée pour décrire les options de communications avec la ressource visée.
TRACE
La méthode TRACE réalise un message de test aller/retour en suivant le chemin de la ressource visée.

PATCH
La méthode PATCH est utilisée pour appliquer des modifications partielles à une ressource.

- **Codes:**

Les codes http, aussi appelés codes statut ou codes d’état, apportent une réponse à une requête effectuée sur un navigateur par un internaute. Ces codes fournissent

une information sur le statut d'une URL par exemple, d'un site internet ou d'une requête.

Ils sont classés en cinq catégories : chacune indique un type de statut différent et impacte d'une manière ou d'une autre le référencement de votre plateforme, ainsi que le comportement des visiteurs.

Les codes 100
Les codes 100 informent le client (c'est-à-dire au navigateur et à fortiori, à l'internaute) que la requête est actuellement en cours de traitement. Dans ce cas, aucun message ne sera affiché dans le navigateur.

Les codes 200
Cette catégorie indique au client que sa requête a abouti. Aussi, la navigation pourra être entreprise sans aucun problème. L'internaute n'est alors pas en mesure de voir le code http, puisqu'il n'a accès qu'au contenu de l'URL qui l'intéresse. Le code 200 OK indique donc au navigateur que les informations ont été traitées correctement par le serveur.

Les codes 300
Dans ce cas précis, la requête a été prise en compte par le serveur mais ne peut pas aboutir, sans l'action du client. Aussi, une procédure complémentaire doit être engagée, pour profiter du contenu de la page. Ces codes http sont principalement utilisés pour les redirections.

Le code 301 : dans ce cas, le navigateur doit rediriger l'internaute vers une autre page. Le webmaster a créé une redirection, pour envoyer le visiteur d'une page internet à une autre. Néanmoins, cela ne signifie pas que le contenu sera identique. L'internaute ne pourra pas voir de message de redirection, simplement la page de destination. Il s'agit ici d'un changement d'adresse définitif. Aussi, le code 301 peut être employé lorsqu'un site internet a changé de nom de domaine, par exemple.
Le code 302 : si le code précédemment mentionné concerne les changements d'adresse définitifs, celui-ci est utilisé pour les changements temporaires. Aussi, l'URL ancienne reste toujours valable, mais ne pourra être consultée pour le moment, tant que le webmaster ne lève pas la redirection. Encore une fois, l'internaute ne constatera pas ce message. Il se rendra automatiquement, grâce à son navigateur, sur la page de redirection concernée.
Les codes 400
Ces codes http concernent une erreur de la part du client. La requête a été convenablement réceptionnée, toutefois celle-ci ne peut être traitée. Vous connaissez très certainement le code 404, qui est généré lorsque l'URL demandée

n’est pas correctement rédigée. Nous verrons également qu’une autre interprétation est possible.

Le code 403 interdit : ici, le client n’est pas en mesure d’accéder à l’URL de son choix, car son accès nécessite une authentification, une autorisation mise en place par le webmaster.
Le code 404 non trouvé : s’il est possible que l’URL ait été mal rédigée, l’erreur 404 concerne également les problèmes de liens morts ou brisés. L’adresse demandée n’existe plus. Dans ce cas, le visiteur est dirigé vers une page d’erreur 404 générée automatiquement, ou personnalisée par le webmaster. Celui-ci est alors en mesure de lui proposer une solution, par exemple le retour sur la page d’accueil du site ou vers un contenu similaire à celui qui n’existe plus.
Les codes 500
Dans ce cas, c’est le serveur qui est défaillant. Celui-ci n’a pas été en mesure de traiter la requête envoyée par le client.

Le code 500 erreur interne du serveur : cette erreur, de nature inattendue, indique que le problème est interne. Aussi, l’administrateur devra en prendre connaissance et réaliser les travaux de maintenance appropriés.
Le code 503 service indisponible : ce message apparaît lorsqu’un serveur n’est plus en mesure de répondre à la requête du client. C’est le cas, lorsqu’un site reçoit trop de visites et ne possède pas les capacités de toutes les traiter. Toutefois, il s’agit d’un dysfonctionnement qui sera pris en compte très rapidement par l’administrateur du site.

3) Architecture Rest basé sur le protocole HTTP

REST (Representational State Transfer) ou RESTful est un style d’architecture permettant de construire des applications (Web, Intranet, Web Service). Il s’agit d’un ensemble de conventions et de bonnes pratiques à respecter et non d’une technologie à part entière. L’architecture REST utilise les spécifications originelles du protocole HTTP, plutôt que de réinventer une surcouche (comme le font SOAP ou XML-RPC par exemple).

Règle n°1 : l’URI comme identifiant des ressources
Règle n°2 : les verbes HTTP comme identifiant des opérations
Règle n°3 : les réponses HTTP comme représentation des ressources
Règle n°4 : les liens comme relation entre ressources
Règle n°5 : un paramètre comme jeton d’authentification
Les 5 règles à suivre pour implémenter REST
Règle n°1 : l’URI comme identifiant des ressources

REST se base sur les URI (Uniform Resource Identifier) afin d'identifier une ressource. Ainsi une application se doit de construire ses URI (et donc ses URL) de manière précise, en tenant compte des contraintes REST. Il est nécessaire de prendre en compte la hiérarchie des ressources et la sémantique des URL pour les éditer :

Quelques exemples de construction d'URL avec RESTful :
Liste des livres

NOK : http://mywebsite.com/book
OK : http://mywebsite.com/books

Filtre et tri sur les livres

NOK : http://mywebsite.com/books/filtre/policier/tri/asc
OK : http://mywebsite.com/books?fitlre=policier&tri=asc

Affichage d'un livre

NOK : http://mywebsite.com/book/display/87
OK : http://mywebsite.com/books/87

Tous les commentaires sur un livre

NOK : http://mywebsite.com/books/comments/87
OK : http://mywebsite.com/books/87/comments

Affichage d'un commentaire sur un livre

NOK : http://mywebsite.com/books/comments/87/1568
OK : http://mywebsite.com/books/87/comments/1568

En construisant correctement les URI, il est possible de les trier, de les hiérarchiser et donc d'améliorer la compréhension du système.

L'URL suivante peut alors être décomposée logiquement :

http://mywebsite.com/books/87/comments/1568 => un commentaire pour un livre
http://mywebsite.com/books/87/comments => tous les commentaires pour un livre
http://mywebsite.com/books/87 => un livre
http://mywebsite.com/books => tous les livres

Règle n°2 : les verbes HTTP comme identifiant des opérations
La seconde règle d'une architecture REST est d'utiliser les verbes HTTP existants plutôt que d'inclure l'opération dans l'URI de la ressource. Ainsi, généralement pour une ressource, il y a 4 opérations possibles (CRUD) :

Créer (create)
Afficher (read)
Mettre à jour (update)
Supprimer (delete)
HTTP propose les verbes correspondant :

Créer (create) => POST
Afficher (read) => GET
Mettre à jour (update) => PUT
Supprimer (delete) => DELETE
Exemple d'URL pour une ressource donnée (un livre par exemple) :

Créer un livre

NOK : GET http://mywebsite.com/books/create
OK : POST http://mywebsite.com/books

Afficher

NOK : GET http://mywebsite.com/books/display/87
OK : GET http://mywebsite.com/books/87

Mettre à jour

NOK : POST http://mywebsite.com/books/editer/87
OK : PUT http://mywebsite.com/books/87

Supprimer

NOK : GET http://mywebsite.com/books/87/delete
OK : DELETE http://mywebsite.com/books/87

Le serveur peut renvoyer les opérations acceptées sur une ressource via l'entête HTTP Allow.

Règle n°3 : les réponses HTTP comme représentation des ressources

Il est important d’avoir à l’esprit que la réponse envoyée n’est pas une ressource, c’est la représentation d’une ressource. Ainsi, une ressource peut avoir plusieurs représentations dans des formats divers : HTML, XML, CSV, JSON, etc.

C’est au client de définir quel format de réponse il souhaite reçevoir via l’entête Accept. Il est possible de définir plusieurs formats.

Quelques exemples :

Réponse en HTML

GET /books
Host: mywebsite.com
Accept: text/html

Réponse en XML

GET /books
Host: mywebsite.com
Accept: application/xml

Règle n°4 : les liens comme relation entre ressources
Les liens d’une ressource vers une autre ont tous une chose en commun : ils indiquent la présence d’une relation. Il est cependant possible de la décrire afin d’améliorer la compréhension du système. Pour expliciter cette description et indiquer la nature de la relation, l’attribut rel doit être spécifier sur tous les liens. Ainsi l’IANA donne une liste de relation parmi lesquelles :

contents
edit
next
last
payment
etc.
La liste complète sur le site de l’IANA :
http://www.iana.org/assignments/link-relations/link-relations.xml

On peut alors parler d’hypermedias.

Exemple de réponse en XML d'une liste paginée de livres :

<?xml>

```
<search>
<link rel="self" title="self"
href="http://mywebsite.com/books?q=policier&page=1&c=5" />
<link rel="next" title="next"
href="http://mywebsite.com/books?q=policier&page=2&c=5" />
<link rel="last" title="last"
href="http://mywebsite.com/books?q=policier&page=4&c=5" />
<book>
//...
</book>
</search>
```

Règle n°5 : un paramètre comme jeton d'authentification
C'est un des sujets les plus souvent abordé quand on parle de REST : comment authentifier une requête ? La réponse est très simple et est massivement utilisée par des APIs renommées (Google, Yahoo, etc.) : le jeton d'authentification.

Chaque requête est envoyée avec un jeton (token) passé en paramètre $_GET de la requête. Ce jeton temporaire est obtenu en envoyant une première requête d'authentification puis en le combinant avec nos requêtes.

Ainsi, on peut construire le scénario suivant :

1. demande d'authentification

```
GET /users/123/authenticate?pass=lkdnssdf54d47894f5123002fds2sd360s0

<?xml>
<user>
<id>123</id>
<name>Nicolas Hachet</name>
</user>
<token>
fsd531gfd5g5df31fdg3g3df45
</token>
```

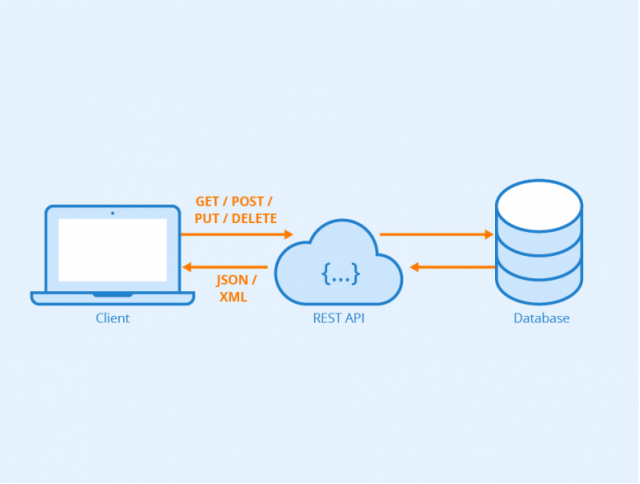
2. accès aux ressources

Cet token est ensuite utilisé pour générer un hash de la requête de cette façon :

```
hash = SHA1(token + requete)
hash = SHA1(fsd531gfd5g5df31fdg3g3df45 + "GET /books")
hash = 456894ds4q15sdq156sd1qsd1qsd156156
```

C’est ce hash qui est passé comme jeton afin de valider l’authentification pour cette requête:

GET /books?user=123&hash=456894ds4q15sdq156sd1qsd1qsd156156



4) Contraintes de l’architecture API REST

Six contraintes architecturales définissent un système REST. Ces contraintes restreignent la façon dont le serveur peut traiter et répondre aux requêtes du client afin que, en agissant dans ces contraintes, le système gagne des propriétés non fonctionnelles désirables, telles que la performance, l'extensibilité, la simplicité, l'évolutivité, la visibilité, la portabilité et la fiabilité3. Un système qui viole une de ces contraintes ne peut pas être considéré comme adhérant à l'architecture REST.

Client–serveur
Les responsabilités sont séparées entre le client et le serveur. Découpler l'interface utilisateur du stockage des données améliore la portabilité de l'interface utilisateur sur plusieurs plateformes. L'extensibilité du système se retrouve aussi améliorée par la simplification des composants serveurs. Mais peut-être encore plus essentiel pour le Web, la séparation permet aux composants d'évoluer indépendamment, supportant ainsi les multiples domaines organisationnels nécessaires à l'échelle d'Internet.

Sans état
La communication client–serveur s'effectue sans conservation de l'état de la session de communication sur le serveur entre deux requêtes successives. L'état de la session est conservé par le client et transmis à chaque nouvelle requête. Les requêtes du client contiennent donc toute l'information nécessaire pour que le

serveur puisse y répondre. La visibilité des interactions entre les composants s'en retrouve améliorée puisque les requêtes sont complètes. La tolérance aux échecs est également plus grande. De plus, le fait de ne pas avoir à maintenir une connexion permanente entre le client et le serveur permet au serveur de répondre à d'autres requêtes venant d'autres clients sans saturer l'ensemble de ses ports de communication, ce qui améliore l'extensibilité du système.

Cependant une exception usuelle à ce mode sans état est la gestion de l'authentification du client, afin que celui-ci n'ait pas à renvoyer ces informations à chacune de ses requêtes.

Avec mise en cache
Les clients et les serveurs intermédiaires peuvent mettre en cache les réponses. Les réponses doivent donc, implicitement ou explicitement, se définir comme pouvant être mise en cache ou non, afin d'empêcher les clients de récupérer des données obsolètes ou inappropriées en réponse à des requêtes ultérieures. Une mise en cache bien gérée élimine partiellement voire totalement certaines interactions client–serveur, améliorant davantage l'extensibilité et la performance du système.

En couches
Article détaillé : Architecture en couches.
Un client ne peut habituellement pas dire s'il est connecté directement au serveur final ou à un serveur intermédiaire. Les serveurs intermédiaires peuvent améliorer l'extensibilité du système en mettant en place une répartition de charge et un cache partagé. Ils peuvent aussi renforcer les politiques de sécurité.

Avec code à la demande (facultative)
Les serveurs peuvent temporairement étendre ou modifier les fonctionnalités d'un client en lui transférant du code exécutable. Par exemple par des applets Java ou des scripts JavaScript. Cela permet de simplifier les clients en réduisant le nombre de fonctionnalités qu'ils doivent mettre en œuvre par défaut et améliore l'extensibilité du système. En revanche, cela réduit aussi la visibilité de l'organisation des ressources. De ce fait, elle constitue une contrainte facultative dans une architecture REST.

Interface uniforme
La contrainte d'interface uniforme est fondamentale dans la conception de n'importe quel système REST. Elle simplifie et découple l'architecture, ce qui permet à chaque composant d'évoluer indépendamment. Les quatre contraintes de l'interface uniforme sont les suivantes.

Identification des ressources dans les requêtes

Chaque ressource est identifiée dans les requêtes, par exemple par un URI dans le cas des systèmes REST basés sur le Web. Les ressources elles-mêmes sont conceptuellement distinctes des représentations qui sont retournées au client. Par exemple, le serveur peut envoyer des données de sa base de données en HTML, XML ou JSON, qui sont des représentations différentes de la représentation interne de la ressource.
Manipulation des ressources par des représentations
Chaque représentation d'une ressource fournit suffisamment d'informations au client pour modifier ou supprimer la ressource.
Messages auto-descriptifs
Chaque message contient assez d'information pour savoir comment l'interpréter. Par exemple, l'interpréteur à invoquer peut être décrit par un type de médias.
Hypermédia comme moteur d'état de l'application (HATEOAS)
Après avoir accédé à un URI initial de l'application — de manière analogue aux humains accédant à la page d'accueil d'un site web —, le client doit être en mesure d'utiliser dynamiquement les hyperliens fournis par le serveur pour découvrir toutes les autres actions possibles et les ressources dont il a besoin pour poursuivre la navigation. Il n'est pas nécessaire pour le client de coder en dur cette information concernant la structure ou la dynamique de l'application.

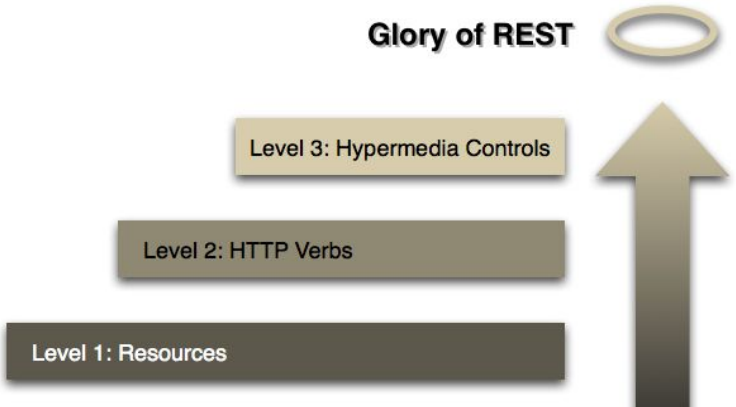
4) Le Modèle de Maturité de Richardson

Le modèle de maturité Richardson classe l'API en fonction de sa maturité RESTful. Il est proposé par Leonard Richardson . Le modèle de maturité Richardson est un moyen de classer votre API selon les contraintes de REST. Il décompose l'élément principal de l'approche REST en quatre niveaux (0 à 3).

Il existe quatre niveaux:

- Niveau 0: Le marais de POX
- Niveau 1: Ressources
- Niveau 2: Verbes HTTP
- Niveau 3: Contrôle hypermédia

Par exemple, un niveau supérieur est plus RESTful par rapport à un niveau inférieur. Ce n'est que lorsqu'une API atteint le niveau 4 que nous la considérons comme une API RESTful.



Niveau 0: Le marais de POX

Le niveau 0 est également appelé POX (Plain Old XML). Au niveau 0, HTTP est utilisé uniquement comme protocole de transport utilisé comme interaction à distance. Il ne prend pas les avantages de HTTP comme les différentes méthodes HTTP et le cache HTTP. Pour obtenir et publier les données, nous envoyons une demande au même URI, et seule la méthode POST peut être utilisée. Ces API utilisent un seul URI et une méthode HTTP appelée POST. En bref, il expose les services Web SOAP dans le style REST.

Par exemple, il peut y avoir de nombreux clients pour une entreprise particulière. Pour tous les différents clients, nous n'avons qu'un seul point final. Pour effectuer l'une des opérations comme obtenir, supprimer, mettre à jour, nous utilisons la même méthode POST.

Pour obtenir les données: http: // localhost: 8080 / customer

Pour publier les données: http: // localhost: 8080 / customer

Dans les deux URI ci-dessus, nous avons utilisé le même URI et la même méthode pour obtenir et publier les clients.

Niveau 1: Ressources

Lorsqu'une API peut faire la distinction entre différentes ressources, elle peut être au niveau 1. Elle utilise plusieurs URI. Où chaque URI est le point d'entrée vers une ressource spécifique. Il expose les ressources avec l'URI approprié. Le niveau 1 s'attaque à la complexité en décomposant d'énormes points de terminaison de service en plusieurs **points de terminaison différents** . Il utilise également une seule méthode HTTP POST pour récupérer et créer des données.

Par exemple, si nous voulons une liste de produits spécifiques, nous passons par l'URI http: // localhost: 8080 / products. Si nous voulons un produit spécifique, nous passons par l'URI http: // localhost: 8080 / products / mobile.

N'oubliez pas les points suivants lors de la création d'un URI:

- Utilisez le domaine et le sous-domaine pour regrouper ou partitionner logiquement les ressources.
- Utilisez **/** pour indiquer une relation hiérarchique.

- Utilisation **,** et **;** pour indiquer des relations non hiérarchiques.
- Utilisez **-** et **_** pour améliorer la lisibilité.
- Utilisez **&** pour séparer les paramètres.
- Évitez d'inclure **des extensions de fichier** .

Niveau 2: Verbes HTTP

Le niveau 2 indique qu'une API doit utiliser les propriétés du protocole pour gérer l'évolutivité et les échecs. Au niveau 2, **les verbes HTTP** corrects sont utilisés avec chaque demande. Cela suggère que pour être vraiment RESTful, les verbes HTTP doivent être utilisés dans l'API. Pour chacune de ces demandes, le code de réponse HTTP correct est fourni.

Nous n'utilisons pas une seule méthode POST pour toutes les demandes. Nous utilisons la méthode **GET** lorsque nous demandons une ressource et utilisons la méthode **DELETE** lorsque nous voulons supprimer une ressource. Utilisez également les codes de réponse du protocole d'application.

Par exemple, pour obtenir les clients, nous envoyons une demande avec l'URI http: // localhost: 8080 / customers, et le serveur envoie une réponse correcte **200 OK** .

Le tableau suivant montre les verbes HTTP et leur utilisation:

Verbes	Sécurité et idempotence	Usage
AVOIR	Y / Y	Il récupère les informations.
PUBLIER	N / N	Il est utilisé pour effectuer diverses actions sur le serveur, telles que créer une nouvelle ressource et mettre à jour une ressource existante, ou apporter un mélange de modifications à une ou plusieurs ressources.
SUPPRIMER	NEW YORK	Il est utilisé pour supprimer une ressource.
METTRE	NEW YORK	Il est utilisé pour mettre à jour ou remplacer une ressource existante ou pour créer une nouvelle ressource avec un URI spécifié par le client.
TÊTE	Y / Y	Il est utilisé pour récupérer les mêmes en-têtes que celui de la réponse GET mais sans aucun corps dans la réponse.
OPTIONS	Y / Y	Il est utilisé pour rechercher la liste des méthodes HTTP prises en charge par n'importe quelle ressource ou pour envoyer une requête ping au serveur.
TRACE	Y / Y	Il est utilisé pour le débogage, qui fait écho aux en-têtes arrière qu'il a reçus.

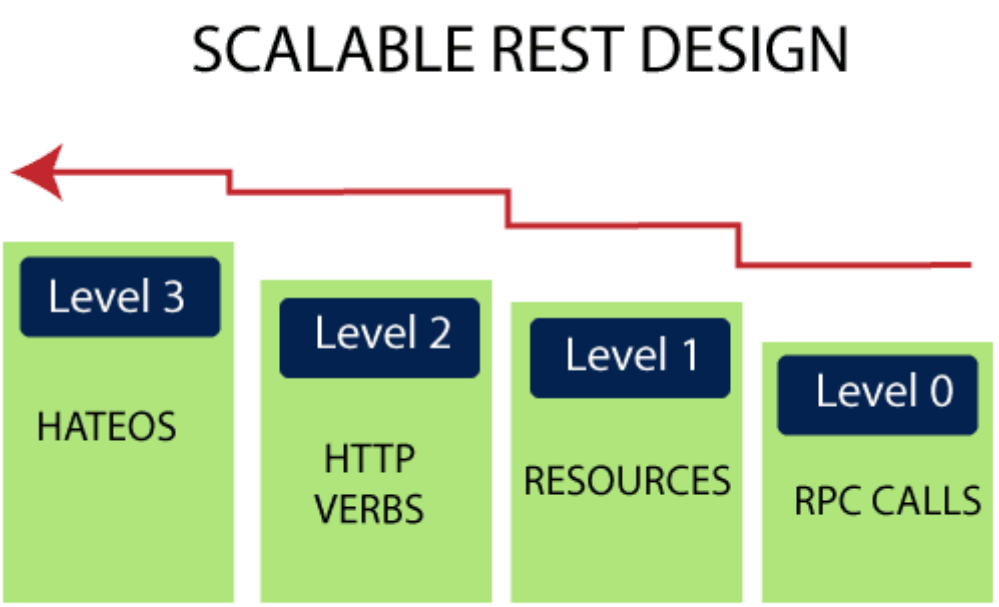
Modèle de maturité Richardson

Le modèle de maturité Richardson classe l'API en fonction de sa maturité RESTful. Il est proposé par **Leonard Richardson** . Le modèle de maturité Richardson est un moyen de classer votre API selon les contraintes de REST. Il décompose l'élément principal de l'approche REST en **quatre** niveaux (0 à 3).

Il existe quatre niveaux:

- Niveau 0: Le marais de POX
- Niveau 1: Ressources
- Niveau 2: Verbes HTTP
- Niveau 3: Contrôle hypermédia

Par exemple, un niveau supérieur est plus RESTful par rapport à un niveau inférieur. Ce n'est que lorsqu'une API atteint le niveau 4 que nous la considérons comme une API RESTful.



Niveau 0: Le marais de POX

Le niveau 0 est également appelé POX (Plain Old XML). Au niveau 0, HTTP est utilisé uniquement comme protocole de transport utilisé comme interaction à distance. Il ne prend pas les avantages de HTTP comme les différentes méthodes HTTP et le cache HTTP. Pour obtenir et publier les données, nous envoyons une demande au même URI, et seule la méthode POST peut être utilisée. Ces API utilisent un seul URI et une méthode HTTP appelée POST. En bref, il expose les services Web SOAP dans le style REST.

Par exemple, il peut y avoir de nombreux clients pour une entreprise particulière. Pour tous les différents clients, nous n'avons qu'un seul point final. Pour effectuer l'une des opérations comme obtenir, supprimer, mettre à jour, nous utilisons la même méthode POST.

Pour obtenir les données: http: // localhost: 8080 / customer

Pour publier les données: http: // localhost: 8080 / customer

Dans les deux URI ci-dessus, nous avons utilisé le même URI et la même méthode pour obtenir et publier les clients.

Niveau 1: Ressources

Lorsqu'une API peut faire la distinction entre différentes ressources, elle peut être au niveau 1. Elle utilise plusieurs URI. Où chaque URI est le point d'entrée vers une ressource spécifique. Il expose les ressources avec l'URI approprié. Le niveau 1 s'attaque à la complexité en décomposant d'énormes points de terminaison de service en plusieurs **points de terminaison différents** . Il utilise également une seule méthode HTTP POST pour récupérer et créer des données.

Par exemple, si nous voulons une liste de produits spécifiques, nous passons par l'URI http: // localhost: 8080 / products. Si nous voulons un produit spécifique, nous passons par l'URI http: // localhost: 8080 / products / mobile.

N'oubliez pas les points suivants lors de la création d'un URI:

- Utilisez le domaine et le sous-domaine pour regrouper ou partitionner logiquement les ressources.

- Utilisez **/** pour indiquer une relation hiérarchique.
- Utilisation **,** et **;** pour indiquer des relations non hiérarchiques.
- Utilisez **-** et **_** pour améliorer la lisibilité.
- Utilisez **&** pour séparer les paramètres.
- Évitez d'inclure **des extensions de fichier** .

Niveau 2: Verbes HTTP

Le niveau 2 indique qu'une API doit utiliser les propriétés du protocole pour gérer l'évolutivité et les échecs. Au niveau 2, **les verbes HTTP** corrects sont utilisés avec chaque demande. Cela suggère que pour être vraiment RESTful, les verbes HTTP doivent être utilisés dans l'API. Pour chacune de ces demandes, le code de réponse HTTP correct est fourni.

Nous n'utilisons pas une seule méthode POST pour toutes les demandes. Nous utilisons la méthode **GET** lorsque nous demandons une ressource et utilisons la méthode **DELETE** lorsque nous voulons supprimer une ressource. Utilisez également les codes de réponse du protocole d'application.

Par exemple, pour obtenir les clients, nous envoyons une demande avec l'URI http: // localhost: 8080 / customers, et le serveur envoie une réponse correcte **200 OK** .

Le tableau suivant montre les verbes HTTP et leur utilisation:

Verbes	Sécurité et idempotence	Usage
AVOIR	Y / Y	Il récupère les informations.

PUB LIER	N / N	Il est utilisé pour effectuer diverses actions sur le serveur, telles que créer une nouvelle ressource et mettre à jour une ressource existante, ou apporter un mélange de modifications à une ou plusieurs ressources.
SUP PRI MER	NEW YORK	Il est utilisé pour supprimer une ressource.
MET TRE	NEW YORK	Il est utilisé pour mettre à jour ou remplacer une ressource existante ou pour créer une nouvelle ressource avec un URI spécifié par le client.
TÊT E	Y / Y	Il est utilisé pour récupérer les mêmes en-têtes que celui de la réponse GET mais sans aucun corps dans la réponse.
OPTI ONS	Y / Y	Il est utilisé pour rechercher la liste des méthodes HTTP prises en charge par n'importe quelle ressource ou pour envoyer une requête ping au serveur.
TRA CE	Y / Y	Il est utilisé pour le débogage, qui fait écho aux en-têtes arrière qu'il a reçus.

Niveau 3: Contrôles hypermédia

Le niveau 3 est le niveau le plus élevé. C'est la combinaison du niveau 2 et HATEOAS. Il fournit également un support pour HATEOAS. Il est utile pour l'auto-documentation.

Par exemple, si nous envoyons une demande GET pour les clients, nous obtiendrons une réponse pour les clients au format JSON avec Hypermedia auto-documenté.

II. La Sérialisation et la Désérialisation

La sérialisation est le processus de conversion d'un objet en un flux d'octets pour stocker l'objet ou le transmettre à la mémoire, à une base de données, ou dans un fichier. Son principal objectif est d'enregistrer l'état d'un objet afin de pouvoir le recréer si nécessaire.
En outre c'est un processus par lequel nous changeons la forme d'un objet tout en conservant ses caractéristiques afin de le rendre transférable plus facilement à une entité. La désérialisation est, par conséquent, le rétablissement à sa forme première d'un objet sérialisé.

1) Notion de Encode et Decode

L'encodage d'un message est la production du message. C'est un système de significations codées.
Le décodage d'un message est de savoir comment un membre du public est en mesure de comprendre et d'interpréter le message. Il est un processus d'interprétation et de traduction des informations codées sous une forme compréhensible

2) Notion de normalisation et dénormalisation

La normalisation est le processus qui permet d'optimiser un modèle logique afin de le rendre non redondant. Ce processus conduit à la fragmentation des données dans plusieurs tables.
Alors que la dénormalisation est le processus consistant à regrouper plusieurs tables liées par des références, en une seule table, en réalisant statiquement les opérations de jointure adéquates.
L'objectif de la dénormalisation est d'améliorer les performances de la BD en recherche sur les tables considérées, en implémentant les jointures plutôt qu'en les calculant.
Un schéma doit être dénormalisé lorsque les performances de certaines recherches sont insuffisantes et que cette insuffisance à pour cause des jointures.
La dénormalisation peut également avoir un effet néfaste sur les performances :

- En mise à jour

Les données redondantes devant être dupliquées plusieurs fois.

- En contrôle supplémentaire

Les moyens de contrôle ajoutés (triggers, niveaux applicatifs, etc.) peuvent être très coûteux.

- En recherche ciblée

Certaines recherches portant avant normalisation sur une "petite" table et portant après sur une "grande" table peuvent être moins performantes après qu'avant.

III. Notion de Fixtures

Les fixtures ou données de tests permettent au développeur d'initialiser les données de la base de données dans un état connu. Par exemple, pour l'entité User, il serait possible de charger plusieurs utilisateurs avec différents rôles et caractéristiques. On pourrait créer un administrateur, un utilisateur standard et un utilisateur avec un accès désactivé. Par la suite, il serait possible d'écrire des tests fonctionnels afin de vérifier que notre page d'authentification (login) fonctionne bien avec tous ces types d'utilisateurs.

Dans bien des cas, il faudra recharger les fixtures entre chaque test. De cette manière, les effets de bord seront évités. Un état stable et connu sera initialisé entre chaque test.

Avec Symfony et Doctrine, il est facile de mettre en place des fixtures. Les fixtures héritent de la classe AbstractFixture et sont déposées dans le dossier /DataFixtures/ORM. Si l'ordre de chargement des fixtures est important, il est aussi possible d'implémenter l'interface OrderedFixtureInterface.

Configuration du bundle DoctrineFixturesBundle
Avant de faciliter la gestion des fixtures, il est conseillé de mettre en place le DoctrineFixturesBundle. Voici la procédure à suivre :

DoctrineFixturesBundle (The Symfony Bundles Documentation)
http://symfony.com/doc/current/bundles/DoctrineFixturesBundle/index.html
Exemple de fixture pour Doctrine et Symfony.

Le code suivant présente une classe qui permet de charger des entités Cegep dans la base de données.

On débute par définir l'espace de nom (namespace), par la suite, on indique les classes à charger avec le mot clé use, finalement, la classe est définie. Cette classe doit hériter de AbstractFixture et fournir une méthode load. C'est dans la méthode load que les entités seront créées et sauvegardées vers la base de données.

Si plusieurs classes de fixtures sont nécessaires, il sera peut-être important de définir l'ordre de chargement (créer les utilisateurs avant de créer les commentaires

associés à ces utilisateurs). L'interface OrderedFixtureInterface vous obligera à définir la méthode getOrder qui devra retourner l'ordre de chargement de votre classe.

/src/Acme/Bundle/ApiBundle/DataFixtures/ORM/LoadCegepData.php

```
<?php
namespace Acme\AppBundle\DataFixtures\ORM;

use Acme\AppBundle\Entity\Cegep;
use Doctrine\Common\DataFixtures\AbstractFixture;
use Doctrine\Common\DataFixtures\OrderedFixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;

class LoadCegepData
        extends AbstractFixture
        implements OrderedFixtureInterface
{

 /**
 * Load data fixtures with the passed EntityManager
 * @param ObjectManager $manager
 */
 public function load(ObjectManager $manager)
 {
  $cegep = new Cegep();
  $cegep->setName("Cégep de Sainte-Foy");
  $cegep->setAddress("123 chemin Ste-Foy");
  $cegep->setPostalCode("G1V 1T3");
  $cegep->setPhone("418-659-6600");
  $cegep->setWebUrl("http://www.cegep-ste-foy.qc.ca");
  $this->addReference('cegep-csf', $cegep);
  $manager->persist($cegep);

  $cegep = new Cegep();
  $cegep->setName("Cegep 1");
  $cegep->setAddress("1 rue");
  $cegep->setPostalCode("G1M 1P1");
  $cegep->setPhone("418 659 1111");
  $cegep->setWebUrl("www.cegep-1.qc.ca");
  $cegep->setCreatedAt(new \DateTime('now'));
  $manager->persist($cegep);

  $manager->flush();
 }

 /**
 * Get the order of this fixture
```

```
 * @return integer
 */
 public function getOrder()
 {
   return 1;
 }

}
```

IV. POSTMAN

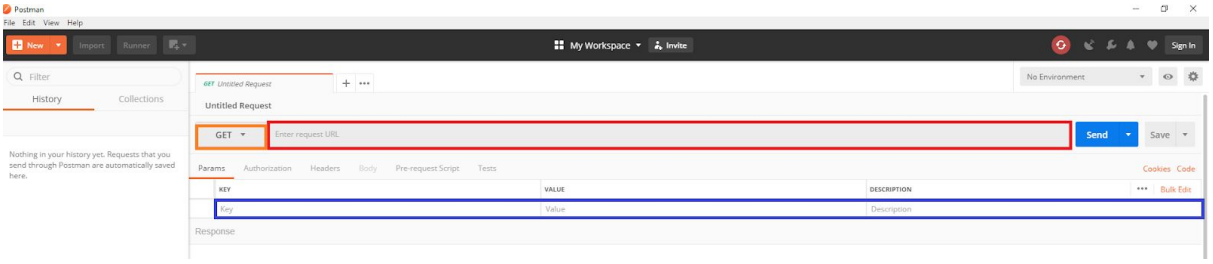
Parmi les nombreuses solutions pour interroger ou tester webservices et API, Postman propose de nombreuses fonctionnalités, une prise en main rapide et une interface graphique agréable.
Il permet de construire et d'exécuter des requêtes HTTP, de les stocker dans un historique afin de pouvoir les rejouer, mais surtout de les organiser en Collections. Cette classification permet notamment de regrouper des requêtes de façon « fonctionnelle » (par exemple enchaînement d'ajout d'item au panier, ou bien un processus d'identification).

Postman assure également la gestion des Environnements, qui permet de contextualiser des variables et d'exécuter des requêtes ou des séries de requêtes dans différents configurations (typiquement : dev, recette, prod).

- Installation

Aller sur https://www.postman.com/downloads/, cliquer sur télécharger l'application et choisissez votre architecture d'ordinateur.
Une fois le téléchargement fini, lancer l'installateur, l'installation devrait se faire automatiquement et PostMan vas se lancer tout seul.

Voici son interface graphique :



Ecole informatique - SUPINFO

Accéder à Open-Campus

ACCUEIL

Installer et utiliser PostMan
Par Baptiste GILLET Publié le 16/10/2018 à 19:50:04 Noter cet article: (0 votes)
Avis favorable du comité de lecture

Baptiste GILLET
Promotion: M.Sc. 2

Campus de Lille

Introduction
Installation
Utiliser PostMan
Conclusion
Introduction
PostMan est un logiciel permettant de tester vos API mais aussi de les enregistrer.

C'est-à-dire qu'au lieu de tester vos API sur votre navigateur et de découvrir vos informations non formalisées, vous allez utiliser postman tester les requêtes, les voir sous le bon format et aussi enregistrer la requête.

Installation
Aller sur PostMan, cliquer sur télécharger l'application et choisissez votre architecture d'ordinateur.

Une fois le téléchargement fini, lancer l'installateur, l'installation devrait se faire automatiquement et PostMan vas se lancer tout seul.

Utiliser PostMan

Voici l'application PostMan, vous pourrez remarquer 3 cadres

Cadre Orange

Méthode de la requête (GET, POST, PUT, DELETE, ...)

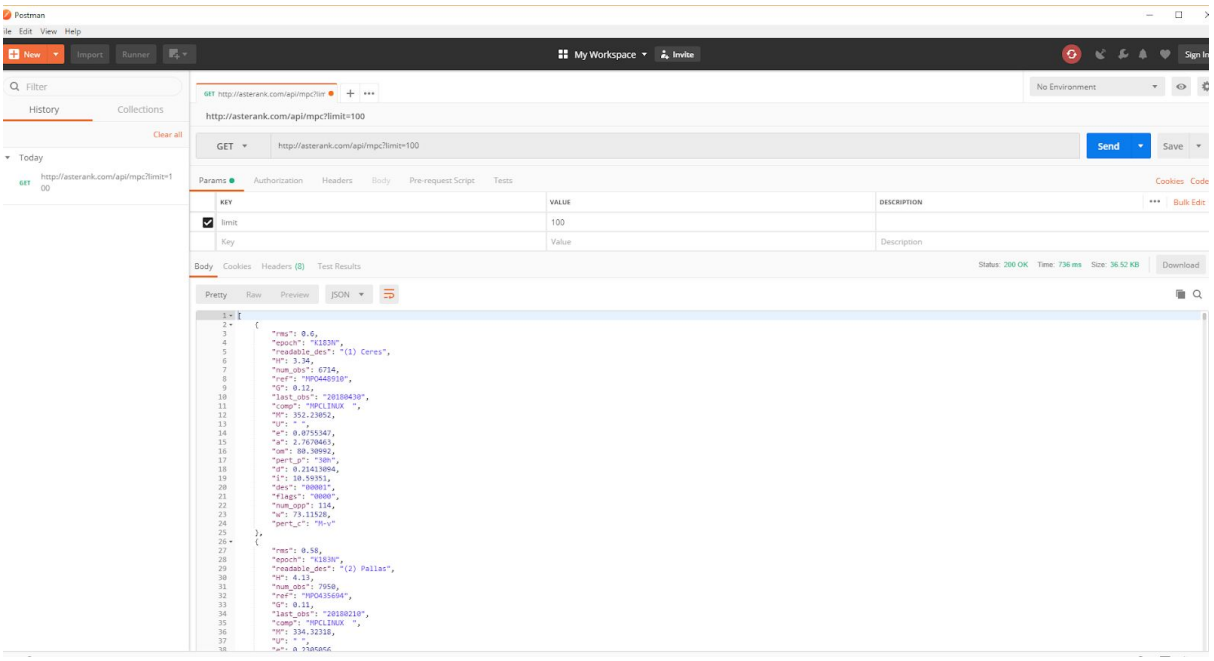
Cadre Rouge

La requête

Cadre Bleu

Les paramètres de la requête

Insérer votre requête (http://asterank.com/api/mpc?limit=100) dans la barre de recherche et vous obtiendrer cela:



V. Bundle API PlatForm

- Installation:

vous pouvez utiliser Composer pour installer le bundle autonome dans un projet Symfony Flex existant:

composer require api

Il n'y a pas d'options de configuration obligatoires bien que de nombreux paramètres soient disponibles .

- Filter:

API Platform Core fournit un système générique pour appliquer des filtres et des critères de tri sur les collections.
Les filtres sont des services et ils peuvent être liés à une ressource de deux manières:
Par exemple, avoir une déclaration de service de filtre:

```yaml
# api/config/services.yaml
services:
    # ...
    offer.date_filter:
        parent: 'api_platform.doctrine.orm.date_filter'
        arguments: [ { dateProperty: ~ } ]
        tags:  [ 'api_platform.filter' ]
        # The following are mandatory only if a _defaults section is defined
        # You may want to isolate filters in a dedicated file to avoid adding them
        autowire: false
        autoconfigure: false
        public: false
```

Nous lions le filtre offer.date_filter à la ressource comme ceci:

PHP YAML XML

```php
<?php
// api/src/Entity/Offer.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;

/**
 * @ApiResource(attributes={"filters"={"offer.date_filter"}})
 */
class Offer
{
    // ...
}
```

- SearchFilter:

Si la prise en charge Doctrine ORM ou MongoDB ODM est activée, l'ajout de filtres est aussi simple que l'enregistrement d'un service de filtrage dans le api/config/services.yamlfichier et l'ajout d'un attribut à votre configuration de ressources.

Les supports de filtres exact, partial, start, endet les word_startstratégies correspondantes:

- partial stratégie utilise LIKE %text%pour rechercher les champs qui contiennent text.
- start stratégie utilise LIKE text%pour rechercher les champs commençant par text.
- end stratégie utilise LIKE %textpour rechercher les champs qui se terminent par text.
- word_start stratégie utilise LIKE text% OR LIKE % text% pour rechercher des champs contenant des mots commençant par text.

Ajoutez la lettre i au filtre si vous souhaitez qu'elle soit insensible à la casse. Par exemple ipartial ou iexact. Notez que cela utilisera la LOWER fonction et aura un impact sur les performances **s'il n'y a pas d'index approprié** .

L'insensibilité à la casse peut déjà être appliquée au niveau de la base de données en fonction du **classement** utilisé. Si vous utilisez MySQL, notez que le utf8_unicode_ci classement couramment utilisé (et son frère utf8mb4_unicode_ci) sont déjà insensibles à la casse, comme indiqué par la _ci partie dans leurs noms.

Remarque: Les filtres de recherche avec la exact stratégie peuvent avoir plusieurs valeurs pour la même propriété (dans ce cas, la condition sera similaire à une clause SQL IN).

Syntaxe: ?property[]=foo&property[]=bar

Dans l'exemple suivant, nous verrons comment autoriser le filtrage d'une liste d'offres de commerce électronique:

```
<?php
// api/src/Entity/Offer.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;
use ApiPlatform\Core\Annotation\ApiFilter;
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\SearchFilter;

/**
 * @ApiResource()
 * @ApiFilter(SearchFilter::class, properties={"id": "exact", "price": "exact", "description":
"partial"})
 */
class Offer
{
    // ...
}
```

http://localhost:8000/api/offers?price=10 retournera toutes les offres avec un prix exact 10. http://localhost:8000/api/offers?description=shirt renverra toutes les offres avec une description contenant le mot "chemise".

Les filtres peuvent être combinés ensemble: http://localhost:8000/api/offers?price=10&description=shirt

- RangeFilter

Le filtre de plage vous permet de filtrer par une valeur inférieure à, supérieure à, inférieure ou égale, supérieure ou égale et entre deux valeurs.

Syntaxe: ?property[<lt|gt|lte|gte|between>]=value

Activez le filtre:

```php
<?php
// api/src/Entity/Offer.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiFilter;
use ApiPlatform\Core\Annotation\ApiResource;
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\RangeFilter;

/**
 * @ApiResource
 * @ApiFilter(RangeFilter::class, properties={"price"})
 */
class Offer
{
    // ...
}
```

Étant donné que le point final de collecte est /offers, vous pouvez filtrer le prix avec la requête suivante: /offers?price[between]=12.99..15.99.

Il renverra toutes les offres priceentre 12,99 et 15,99.

Vous pouvez filtrer les offres en joignant deux valeurs, par exemple: /offers?price[gt]=12.99&price[lt]=19.99.

- OrderFilter

Le filtre d'ordre permet de trier une collection en fonction des propriétés données.

Syntaxe: ?order[property]=<asc|desc>

Activez le filtre:

```php
<?php
// api/src/Entity/Offer.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiFilter;
use ApiPlatform\Core\Annotation\ApiResource;
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\OrderFilter;

/**
 * @ApiResource
 * @ApiFilter(OrderFilter::class, properties={"id", "name"}, arguments=
{"orderParameterName"="order"})
 */
class Offer
{
    // ...
}
```

Étant donné que le point final de collecte est /offers, vous pouvez filtrer les offres par nom en ordre croissant, puis par ID dans l' ordre décroissant avec la requête suivante: /offers?order[name]=desc&order[id]=asc.

Par défaut, chaque fois que la requête ne spécifie pas la direction de manière explicite (par exemple /offers?order[name]&order[id]:), les filtres ne seront appliqués que si vous configurez une direction d'ordre par défaut à utiliser:

```php
<?php
// api/src/Entity/Offer.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiFilter;
use ApiPlatform\Core\Annotation\ApiResource;
use ApiPlatform\Core\Bridge\Doctrine\Orm\Filter\OrderFilter;

/**
 * @ApiResource
 * @ApiFilter(OrderFilter::class, properties={"id": "ASC", "name": "DESC"})
 */
class Offer
{
    // ...
}
```

- PropertyFilter

Le filtre de propriétés ajoute la possibilité de sélectionner les propriétés à sérialiser (ensembles de champs épars).

Syntaxe: ?properties[]=<property>&properties[<relation>][]=<property>

Vous pouvez ajouter autant de propriétés que vous le souhaitez.

Activez le filtre:

```php
<?php
// api/src/Entity/Book.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiFilter;
use ApiPlatform\Core\Annotation\ApiResource;
use ApiPlatform\Core\Serializer\Filter\PropertyFilter;

/**
 * @ApiResource
 * @ApiFilter(PropertyFilter::class, arguments={"parameterName": "properties",
"overrideDefaultProperties": false, "whitelist": {"allowed_property"}})
 */
class Book
{
    // ...
}
```

Trois arguments sont disponibles pour configurer le filtre:

- parameterName est le nom du paramètre de requête (par défaut properties)
- overrideDefaultProperties permet de remplacer les propriétés de sérialisation par défaut (par défaut false)
- whitelist liste blanche des propriétés pour éviter une exposition incontrôlée des données (par défaut null pour autoriser toutes les propriétés)

Étant donné que le point final de collecte est /books, vous pouvez filtrer les propriétés de sérialisation avec la requête suivante: /books?properties[]=title&properties[]=author. Si vous voulez inclure certaines propriétés du imbriquée document « auteur », utilisez: /books?properties[]=title&properties[author][]=name.

- Pagination

API Platform Core prend en charge nativement les collections paginées. La pagination est activée par défaut pour toutes les collections. Chaque collection contient 30 éléments par page. L'activation de la pagination et le nombre d'éléments par page peuvent être configurés à partir de:

- côté serveur (global ou par ressource)

- côté client, via un paramètre GET personnalisé (désactivé par défaut)

Lors de l'émission d'une `GET`demande sur une collection contenant plus d'une page (ici `/books`), une **collection Hydra** est retournée. Il s'agit d'un document JSON (-LD) valide contenant des éléments de la page demandée et des métadonnées.

```
{
  "@context": "/contexts/Book",
  "@id": "/books",
  "@type": "hydra:Collection",
  "hydra:member": [
    {
      "@id": "/books/1",
      "@type": "http://schema.org/Book",
      "name": "My awesome book"
    },
    {
        "_": "Other items in the collection..."
    },
  ],
  "hydra:totalItems": 50,
  "hydra:view": {
    "@id": "/books?page=1",
    "@type": "hydra:PartialCollectionView",
    "hydra:first": "/books?page=1",
    "hydra:last": "/books?page=2",
    "hydra:next": "/books?page=2"
  }
}
```

Les liens hypermédia vers la première, la dernière, la page précédente et la page suivante de la collection sont affichés ainsi que le nombre total d'éléments dans la collection.

Le nom du paramètre de page peut être modifié avec la configuration suivante:

```
# api/config/packages/api_platform.yaml
api_platform:
    collection:
        pagination:
            page_parameter_name: _page
```

- Api Subressources

Une sous-ressource est une collection ou un élément qui appartient à une autre ressource. La plate-forme API facilite la création de telles opérations.
Le point de départ d'une sous-ressource doit être une relation sur une ressource existante. Par exemple, créons deux entités (Question, Réponse) et mettons en place une sous-ressource pour `/question/42/answer` nous donner la réponse à la question 42

```php
<?php
// api/src/Entity/Answer.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ApiResource
 */
class Answer
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column
     */
    public $content;

    /**
     * @ORM\OneToOne(targetEntity="Question", mappedBy="answer")
     */
    public $question;

    public function getId(): ?int
    {
        return $this->id;
    }

    // ...
}

// api/src/Entity/Question.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;
use ApiPlatform\Core\Annotation\ApiSubresource;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 * @ApiResource
 */
class Question
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @ORM\Column
     */
    public $content;

    /**
```

```
     * @ORM\OneToOne(targetEntity="Answer", inversedBy="question")
     * @ORM\JoinColumn(referencedColumnName="id", unique=true)
     * @ApiSubresource
     */
    public $answer;

    public function getId(): ?int
    {
        return $this->id;
    }

    // ...
}
```

Notez que tout ce que nous avions à faire est de mettre en place @ApiSubresourcela Question::answer relation. Étant donné que la answer relation est un à un, nous savons que cette sous-ressource est un élément. Par conséquent, la réponse ressemblera à ceci:

```
{

  "@context": "/contexts/Answer",

  "@id": "/answers/42",

  "@type": "Answer",

  "id": 42,

  "content": "Life, the Universe, and Everything",

  "question": "/questions/42"

}
```

Si vous placez la sous-ressource sur une relation qui est à plusieurs, vous récupérerez une collection.

Enfin et surtout, les sous-ressources peuvent être imbriquées, ce qui /questions/42/answer/comments permettra d'obtenir la collection de commentaires pour la réponse à la question 42.

Remarque: seules les GETopérations sont prises en charge pour le moment

- **itemOperations - CollectionOperations**

itemOperations et **collectionOperations** sont des tableaux contenant une liste d'opérations. Chaque opération est définie par une clé correspondant au nom de l'opération qui peut être tout ce que vous voulez et un tableau de propriétés comme

valeur. Si une liste vide d'opérations est fournie, toutes les opérations sont désactivées.

Si le nom de l' opération correspond à une des méthodes HTTP prises en charge (GET, POST, PUT, PATCH ou DELETE), correspondant method propriété sera automatiquement ajouté.

```php
<?php
// api/src/Entity/Book.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;

/**
 * ...
 * @ApiResource(
 *     collectionOperations={"get"},
 *     itemOperations={"get"}
 * )
 */
class Book
{
    // ...
}
```

L'exemple précédent peut également être écrit avec une définition de méthode explicite:

```php
<?php
// api/src/Entity/Book.php

namespace App\Entity;

use ApiPlatform\Core\Annotation\ApiResource;

/**
 * ...
 * @ApiResource(
 *     collectionOperations={"get"={"method"="GET"}},
 *     itemOperations={"get"={"method"="GET"}}
 * )
 */
class Book
{
    // ...
}
```

API Platform Core est suffisamment intelligent pour enregistrer automatiquement la route Symfony applicable référençant une action CRUD intégrée simplement en

spécifiant le nom de la méthode comme clé ou en vérifiant la méthode HTTP explicitement configurée.

Si vous ne souhaitez pas autoriser l'accès à l'élément de ressource (c'est-à-dire que vous ne voulez pas d' `GET`opération d'élément), au lieu de l'omettre complètement, vous devez plutôt déclarer une `GET`opération d'élément qui renvoie HTTP 404 (Introuvable), de sorte que la ressource l'article peut toujours être identifié par un IRI. Par exemple:

```php
<?php
// api/src/Entity/Book.php

namespace App\Entity;

use ApiPlatform\Core\Action\NotFoundAction;
use ApiPlatform\Core\Annotation\ApiResource;

/**
 * @ApiResource(
 *     collectionOperations={
 *         "get",
 *     },
 *     itemOperations={
 *         "get"={
 *             "controller"=NotFoundAction::class,
 *             "read"=false,
 *             "output"=false,
 *         },
 *     },
 * )
 */
class Book
{
}
```

V. Gestion de l’authentification

- Notion de JWT (Json Web Token)

  est un standard ouvert défini dans la RFC 75191. Il permet l'échange sécurisé de jetons (tokens) entre plusieurs parties. Cette sécurité de l’échange se traduit par la vérification de l’intégrité des données à l’aide d’une signature numérique.

Un jeton se compose de trois parties :

- Un *en-tête* (header), utilisé pour décrire le jeton. Il s'agit d'un objet JSON.
- Une *charge utile* (payload) qui représente les informations embarquées dans le jeton. Il s'agit également d'un objet JSON.
- Une *signature* numérique.

En-tête

```
{"typ": "jwt", "alg": "HS512"}
```

Charge utile

```
{"name":"Wikipedia","iat":1525777938}
```

Dans l'exemple ci-dessus, on voit dans l'en-tête que le jeton est un JSON Web Token (JWT) et que l'algorithme utilisé pour la signature est HMAC-SHA512.

signature numérique

Pour obtenir la signature, il faut tout d'abord encoder séparément l'en-tête et la charge utile avec Base64url défini dans la RFC 46482. Ensuite, on les concatène ensemble en les séparant avec un point. On obtient la signature de ce résultat avec l'algorithme choisi. Cette signature est ajoutée au résultat de la même manière (encodée et séparée par un point).

À noter que pour l'encodage en Base64url, le caractère de remplissage '=' n'est pas obligatoire et ne sera pas utilisé dans la création du JSON Web Token pour faciliter la transmission dans une URL.

À partir de l'exemple ci-dessus, voici les différentes étapes pour obtenir un JSON Web Token.

Encodage de l'en-tête

```
eyJ0eXAiOiAiand0IiwgImFsZyI6ICJIUzUxMiJ9
```

Encodage de la charge utile

```
eyJuYW1lIjoiV2lraXBlZGlhIiwiaWF0IjoxNTI1Nzc3OTM4fQ
```

Concaténation des deux éléments, séparation par un point

```
eyJ0eXAiOiAiand0IiwgImFsZyI6ICJIUzUxMiJ9.eyJuYW1lIj
oiV2lraXBlZGlhIiwiaWF0IjoxNTi1Nzc3OTM4fQ
```

Obtention de la signature avec l'algorithme HMAC-SHA512

```
HMACSHA512(concatenation, 'ma super clé secrète')
```

Encodage de la signature (toujours avec Base64url)

```
iu0aMCsaepPy6ULphSX5PT32oPvKkM5dPl131knIDq9Cr8OUzzA
CsuBnpSJ_rE9XkGjmQVawcvyCHLiM4Kr6NA
```

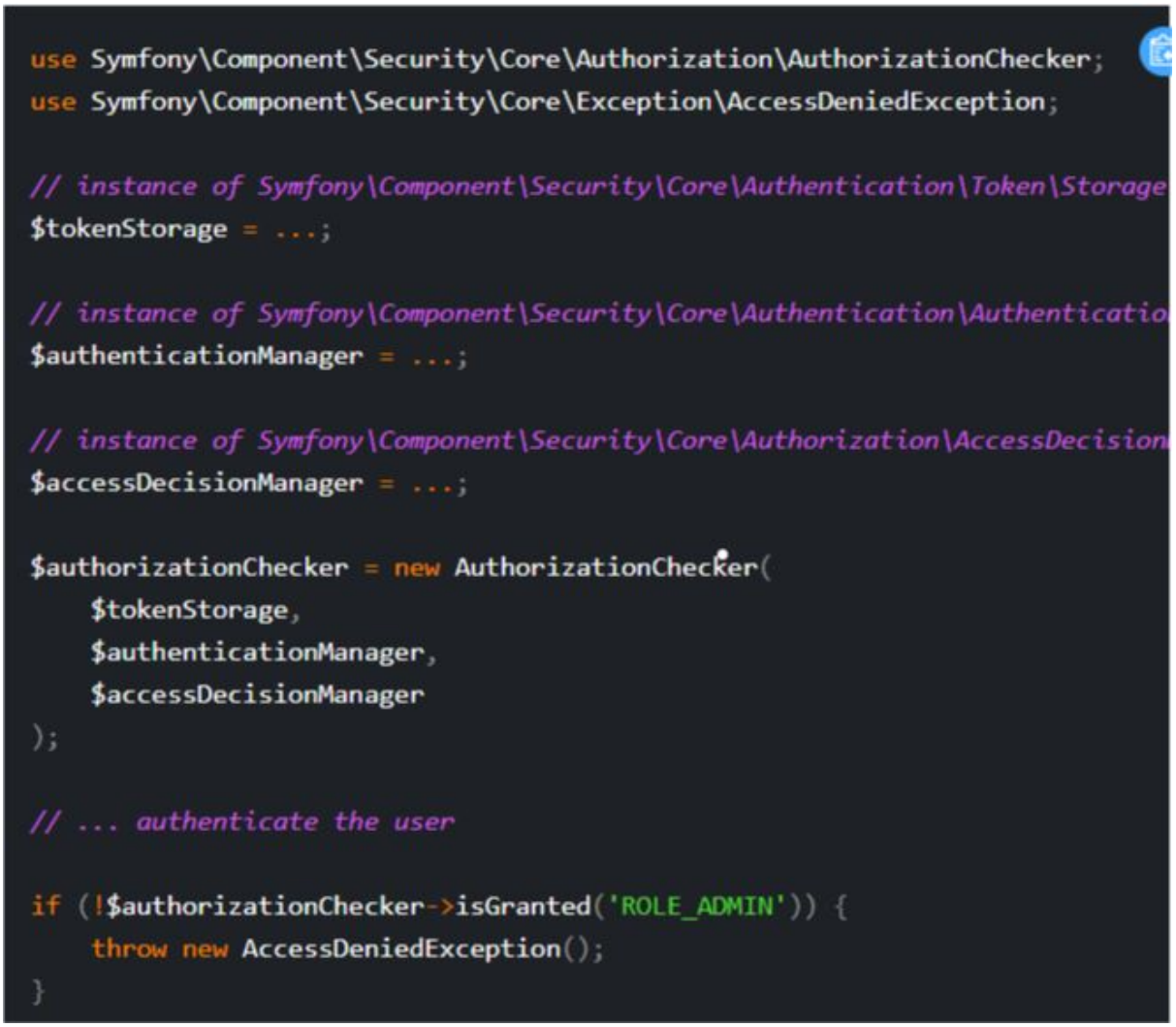
Concaténation des deux éléments, séparation par un point

```
eyJ0eXAiOiAiand0IiwgImFsZyI6ICJIUzUxMiJ9.eyJuYW1lIj
oiV2lraXBlZGlhIiwiaWF0IjoxNTI1Nzc3OTM4fQ.iu0aMCsaep
Py6ULphSX5PT32oPvKkM5dPl131knIDq9Cr8OUzzACsuBnpSJ_r
E9XkGjmQVawcvyCHLiM4Kr6NA
```

- Firewall

En informatique , un firewall ou pare - feu est un système de sécurité réseau qui surveille et contrôle le trafic réseau entrant et sortant en fonction de règles de sécurité.

Un pare-feu établit généralement une barrière entre un réseau interne approuvé et un réseau externe non approuvé, comme Internet.



```php
use Symfony\Component\Security\Core\Authorization\AuthorizationChecker;
use Symfony\Component\Security\Core\Exception\AccessDeniedException;

// instance of Symfony\Component\Security\Core\Authentication\Token\Storage
$tokenStorage = ...;

// instance of Symfony\Component\Security\Core\Authentication\Authenticatio
$authenticationManager = ...;

// instance of Symfony\Component\Security\Core\Authorization\AccessDecision
$accessDecisionManager = ...;

$authorizationChecker = new AuthorizationChecker(
    $tokenStorage,
    $authenticationManager,
    $accessDecisionManager
);

// ... authenticate the user

if (!$authorizationChecker->isGranted('ROLE_ADMIN')) {
    throw new AccessDeniedException();
}
```

Un firewall pour les requêtes HTTP

L'authentification d'un utilisateur se fait par le pare-feu. Une application peut avoir plusieurs zones sécurisées, de sorte que le pare-feu est configuré à l'aide d'une carte de ces zones sécurisées. Pour chacune de ces zones, la carte contient un matcher de requête et une collection d'auditeurs. L'outil de correspondance des demandes permet au pare-feu de savoir si la demande actuelle pointe vers une zone sécurisée. Il est ensuite demandé aux auditeurs si la demande en cours peut être utilisée pour authentifier l'utilisateur

```php
use Symfony\Component\HttpFoundation\RequestMatcher;
use Symfony\Component\Security\Http\Firewall\ExceptionListener;
use Symfony\Component\Security\Http\FirewallMap;

$firewallMap = new FirewallMap();

$requestMatcher = new RequestMatcher('^/secured-area/');

// array of callables
$listeners = [...];

$exceptionListener = new ExceptionListener(...);

$firewallMap->add($requestMatcher, $listeners, $exceptionListener);
```

La carte du pare-feu sera donnée au pare-feu comme premier argument, avec le répartiteur d'événements utilisé par HttpKernel:

```php
use Symfony\Component\HttpKernel\KernelEvents;
use Symfony\Component\Security\Http\Firewall;

// the EventDispatcher used by the HttpKernel
$dispatcher = ...;

$firewall = new Firewall($firewallMap, $dispatcher);

$dispatcher->addListener(
    KernelEvents::REQUEST,
    [$firewall, 'onKernelRequest']
);
```

Le pare-feu est enregistré pour écouter l' kernel.request événement qui sera distribué par le HttpKernel au début de chaque demande qu'il traite. De cette façon, le pare-feu peut empêcher l'utilisateur d'aller plus loin que ce qui est autorisé.

Firewall Config

Les informations sur un pare-feu donné, telles que son nom, son fournisseur, son contexte, son point d'entrée et son URL d'accès refusé, sont fournies par les instances de la FirewallConfig classe.

Cet objet est accessible via la méthode de la classe et via la méthode de la classe.getFirewallConfig(Request $request)FirewallMapget Config()FirewallContext

Firewall Listeners

Lorsque le pare-feu est informé de l' kernel.requestévénement, il demande à la carte du pare-feu si la demande correspond à l'une des zones sécurisées. La première zone sécurisée qui correspond à la demande renverra un ensemble d'écouteurs de pare-feu correspondants (qui sont chacun appelables). Ces auditeurs seront tous invités à traiter la demande en cours. Cela signifie essentiellement: savoir si la demande actuelle contient des informations permettant à l'utilisateur d'être authentifié (par exemple, l'écouteur d'authentification HTTP de base vérifie si la demande a un en-tête appelé PHP_AUTH_USER).

Écouteur d'exceptions

Si l'un des écouteurs lance un AuthenticationException, l'écouteur d'exception fourni lors de l'ajout de zones sécurisées à la carte du pare-feu se déclenchera.

L'écouteur d'exceptions détermine ce qui se passe ensuite, en fonction des arguments qu'il a reçus lors de sa création. Il peut démarrer la procédure d'authentification, peut-être demander à l'utilisateur de fournir à nouveau ses informations d'identification (alors qu'il n'a été authentifié que sur la base d'un cookie "souvenez-vous de moi"), ou transformer l'exception en un AccessDeniedHttpException, ce qui aboutira éventuellement à un "HTTP / 1.1 403: Accès refusé ».

Points d'entrée

Lorsque l'utilisateur n'est pas du tout authentifié (c'est-à-dire lorsque le stockage de jetons n'a pas encore de jeton), le point d'entrée du pare-feu sera appelé pour «démarrer» le processus d'authentification. Un point d'entrée doit mettre en œuvre AuthenticationEntryPointInterface, qui n'a qu'une seule méthode: start(). Cette méthode reçoit l' Request objet actuel et l'exception par

laquelle l'écouteur d'exceptions a été déclenché. La méthode doit renvoyer un Response objet. Cela pourrait être, par exemple, la page contenant le formulaire de connexion ou, dans le cas de l'authentification HTTP de base, une réponse avec un en- WWW-Authenticate tête, qui invitera l'utilisateur à fournir son nom d'utilisateur et son mot de passe.

Flux: pare-feu, authentification, autorisation ¶

J'espère que vous pouvez maintenant voir un peu comment fonctionne le «flux» du contexte de sécurité:

1. Le pare-feu est enregistré en tant qu'auditeur sur l' kernel.request événement;
2. Au début de la demande, le pare-feu vérifie la carte du pare-feu pour voir si un pare-feu doit être actif pour cette URL;
3. Si un pare-feu est trouvé dans la carte pour cette URL, ses écouteurs sont avertis;
4. Chaque auditeur vérifie si la demande actuelle contient des informations d'authentification - un auditeur peut (a) authentifier un utilisateur, (b) lancer un AuthenticationException, ou (c) ne rien faire (car il n'y a aucune information d'authentification sur la demande);
5. Une fois qu'un utilisateur est authentifié, vous utiliserez l' autorisation pour refuser l'accès à certaines ressources.

access_control

Définit la protection de sécurité des URL de votre application. Il est utilisé par exemple pour déclencher l'authentification de l'utilisateur lors de la tentative d'accès au backend et pour permettre aux utilisateurs anonymes d'accéder à la page du formulaire de connexion.

1. Options de correspondance

Symfony crée une instance de RequestMatcher pour chaque access_control entrée, qui détermine si oui ou non un contrôle d'accès donné doit être utilisé sur cette demande. Les access_control options suivantes sont utilisées pour la correspondance:

- path: une expression régulière (sans délimiteurs)
- ip ou ips: les masques de réseau sont également pris en charge
- port: un nombre entier
- host: une expression régulière

- methods: une ou plusieurs méthodes

Prenons l' access_control exemple des entrées suivantes:

```yaml
# config/packages/security.yaml
security:
    # ...
    access_control:
        - { path: '^/admin', roles: ROLE_USER_IP, ip: 127.0.0.1 }
        - { path: '^/admin', roles: ROLE_USER_PORT, ip: 127.0.0.1, port: 8080 }
        - { path: '^/admin', roles: ROLE_USER_HOST, host: symfony\.com$ }
        - { path: '^/admin', roles: ROLE_USER_METHOD, methods: [POST, PUT] }
```

Pour chaque demande entrante, Symfony décidera laquelle access_control utiliser en fonction de l'URI, de l'adresse IP du client, du nom d'hôte entrant et de la méthode de demande. Rappelez - vous, la première règle correspondant est utilisé, et si ip, port, host ou method ne sont pas spécifiés pour une entrée, qui access_control correspond à tout ip, port, host ou method:

URI	IP	PORT	HÔTE	MÉTHODE	access_control	Pourquoi?
`/admin/user`	127.0.0.1	80	exemple.com	AVOIR	règle # 1 (`ROLE_USER_IP`)	Les correspondances URI `path` et IP correspondent `ip` .
`/admin/user`	127.0.0.1	80	symfony.com	AVOIR	règle # 1 (`ROLE_USER_IP`)	Le `path` et `ip` correspondent toujours. Cela correspondrait également à l' `ROLE_USER_HOST` entrée, mais *seule* la **première** `access_control` correspondance est utilisée.
`/admin/user`	127.0.0.1	8080	symfony.com	AVOIR	règle # 2 (`ROLE_USER_PORT`)	Le `path` , `ip` et `port` match.
`/admin/user`	168.0.0.1	80	symfony.com	AVOIR	règle # 3 (`ROLE_USER_HOST`)	Le `ip` ne correspond pas à la première règle, donc la deuxième règle (qui correspond) est utilisée.
`/admin/user`	168.0.0.1	80	symfony.com	PUBLIER	règle # 3 (`ROLE_USER_HOST`)	La deuxième règle correspond toujours. Cela correspondrait également à la troisième règle (`ROLE_USER_METHOD`), mais seule la **première** correspondance `access_control` est utilisée.
`/admin/user`	168.0.0.1	80	exemple.com	PUBLIER	règle # 4 (`ROLE_USER_METHOD`)	Le `ip` et `host` ne correspondent pas aux deux premières entrées, mais le troisième – `ROLE_USER_METHOD` – correspond à et est utilisé.
`/foo`	127.0.0.1	80	symfony.com	PUBLIER	ne correspond à aucune entrée	Cela ne correspond à aucune `access_control` règle, car son URI ne correspond à aucune des `path` valeurs.

VII. API platform et Doctrine Events

Doctrine , l'ensemble des bibliothèques PHP utilisées par Symfony pour travailler avec les bases de données, fournit un système d'événements léger pour mettre à jour les entités pendant l'exécution de l'application. Ces événements, appelés événements de cycle de vie , permettent

d'effectuer des tâches telles que *«mettre à jour la propriété createdAt automatiquement juste avant la persistance d'entités de ce type»* .

Déclencheurs Doctrine événements avant / après l' exécution des opérations de l'entité la plus courante (par exemple prePersist/postPersist, preUpdate/postUpdate) et aussi sur d' autres tâches courantes (par exemple loadClassMetadata, onClear).

Il existe différentes façons d'écouter ces événements de la Doctrine:

- **Les rappels du cycle de vie** , ils sont définis comme des méthodes sur les classes d'entités et ils sont appelés lorsque les événements sont déclenchés;
- **Écouteurs et abonnés du cycle de vie** , ce sont des classes avec des méthodes de rappel pour un ou plusieurs événements et elles sont appelées pour toutes les entités;
- **Écouteurs d'entité** , ils sont similaires aux écouteurs de cycle de vie, mais ils ne sont appelés que pour les entités d'une certaine classe.

Ce sont les **inconvénients et avantages** de chacun:

- Les rappels ont de meilleures performances car ils ne s'appliquent qu'à une seule classe d'entité, mais vous ne pouvez pas réutiliser la logique pour différentes entités et ils n'ont pas accès aux services Symfony ;
- Les écouteurs et les abonnés du cycle de vie peuvent réutiliser la logique entre différentes entités et peuvent accéder aux services Symfony mais leurs performances sont moins bonnes car ils sont appelés pour toutes les entités;
- Les écouteurs d'entité ont les mêmes avantages que les écouteurs de cycle de vie et ils ont de meilleures performances car ils ne s'appliquent qu'à une seule classe d'entité.

Rappels Doctrine Lifecycle

Les rappels de cycle de vie sont définis comme des méthodes à l'intérieur de l'entité que vous souhaitez modifier. Par exemple, supposons que vous souhaitiez définir une createdAtcolonne

de date à la date actuelle, mais uniquement lorsque l'entité est persistée pour la première fois (c'est-à-dire insérée). Pour ce faire, définissez un rappel pour l'

```php
// src/Entity/Product.php
use Doctrine\ORM\Mapping as ORM;

// When using annotations, don't forget to add @ORM\HasLifecycleCall
// to the class of the entity where you define the callback

/**
 * @ORM\Entity()
 * @ORM\HasLifecycleCallbacks()
 */
class Product
{
    // ...

    /**
     * @ORM\PrePersist
     */
    public function setCreatedAtValue()
    {
        $this->createdAt = new \DateTime();
    }
}
```

prePersistévénement Doctrine:

Écouteurs du cycle de vie de la Doctrine

Les écouteurs du cycle de vie sont définis comme des classes PHP qui écoutent un seul événement Doctrine sur toutes les entités d'application. Par exemple, supposons que vous souhaitiez mettre à jour un index de recherche chaque fois qu'une nouvelle entité est conservée

dans la base de données. Pour ce faire, définissez un écouteur pour l' postPersist événement Doctrine:

```php
// src/EventListener/SearchIndexer.php
namespace App\EventListener;

use App\Entity\Product;
use Doctrine\Persistence\Event\LifecycleEventArgs;

class SearchIndexer
{
    // the listener methods receive an argument which gives you access to
    // both the entity object of the event and the entity manager itself
    public function postPersist(LifecycleEventArgs $args)
    {
        $entity = $args->getObject();

        // if this listener only applies to certain entity types,
        // add some code to check the entity type as early as possible
        if (!$entity instanceof Product) {
            return;
        }

        $entityManager = $args->getObjectManager();
        // ... do something with the Product entity
    }
}
```

L'étape suivante consiste à activer l'écouteur Doctrine dans l'application Symfony en créant un nouveau service pour celui-ci et en le marquant avec la doctrine.event_listener balise:

```yaml
# config/services.yaml
services:
    # ...

    App\EventListener\SearchIndexer:
        tags:
            -
                name: 'doctrine.event_listener'
                # this is the only required option for the lifecycle listener tag
                event: 'postPersist'

                # listeners can define their priority in case multiple listeners are associ
                # to the same event (default priority = 0; higher numbers = listener is run
                priority: 500

                # you can also restrict listeners to a specific Doctrine connection
                connection: 'default'
```

Écouteurs d'entité de doctrine

Les écouteurs d'entité sont définis comme des classes PHP qui écoutent un seul événement Doctrine sur une seule classe d'entité. Par exemple, supposons que vous souhaitiez envoyer des notifications chaque fois qu'une User entité est modifiée dans la base de données. Pour ce faire, définissez un écouteur pour l' postUpdate événement Doctrine:

```php
// src/EventListener/UserChangedNotifier.php
namespace App\EventListener;

use App\Entity\User;
use Doctrine\Persistence\Event\LifecycleEventArgs;

class UserChangedNotifier
{
    // the entity listener methods receive two arguments:
    // the entity instance and the lifecycle event
    public function postUpdate(User $user, LifecycleEventArgs $event)
    {
        // ... do something to notify the changes
    }
}
```

L'étape suivante consiste à activer l'écouteur Doctrine dans l'application Symfony en créant un nouveau service pour celui-ci et en le marquant avec la doctrine.orm.entity_listener balise:

```yaml
# config/services.yaml
services:
    # ...

    App\EventListener\UserChangedNotifier:
        tags:
            -
                # these are the options required to define the entity listener
                name: 'doctrine.orm.entity_listener'
                event: 'postUpdate'
                entity: 'App\Entity\User'

                # these are other options that you may define if needed

                # set the 'lazy' option to TRUE to only instantiate listeners when they ar
                # lazy: true

                # set the 'entity_manager' option if the listener is not associated to the
                # entity_manager: 'custom'

                # by default, Symfony looks for a method called after the event (e.g. post
                # if it doesn't exist, it tries to execute the '__invoke()' method, but yo
                # configure a custom method name with the 'method' option
                # method: 'checkUserChanges'
```

Doctrine Lifecycle

Les abonnés au cycle de vie sont définis comme des classes PHP qui implémentent l' Doctrine\Common\EventSubscriberinterface et qui écoutent un ou plusieurs événements Doctrine sur toutes les entités d'application. Par exemple, supposons que vous souhaitiez enregistrer toute l'activité de la base de données. Pour ce faire, définissez un abonné pour les événements postPersist, postRemove et postUpdate Doctrine:

```php
// src/EventListener/DatabaseActivitySubscriber.php
namespace App\EventListener;

use App\Entity\Product;
use Doctrine\Common\EventSubscriber;
use Doctrine\ORM\Events;
use Doctrine\Persistence\Event\LifecycleEventArgs;

class DatabaseActivitySubscriber implements EventSubscriber
{
    // this method can only return the event names; you cannot define a
    // custom method name to execute when each event triggers
    public function getSubscribedEvents()
    {
        return [
            Events::postPersist,
            Events::postRemove,
            Events::postUpdate,
        ];
    }

    // callback methods must be called exactly like the events they listen to;
    // they receive an argument of type LifecycleEventArgs, which gives you access
```

```php
    public function postPersist(LifecycleEventArgs $args)
    {
        $this->logActivity('persist', $args);
    }

    public function postRemove(LifecycleEventArgs $args)
    {
        $this->logActivity('remove', $args);
    }

    public function postUpdate(LifecycleEventArgs $args)
    {
        $this->logActivity('update', $args);
    }

    private function logActivity(string $action, LifecycleEventArgs $args)
    {
        $entity = $args->getObject();

        // if this subscriber only applies to certain entity types,
        // add some code to check the entity type as early as possible
        if (!$entity instanceof Product) {
            return;
        }

        // ... get the entity information and log it somehow
    }
}
```

L'étape suivante consiste à activer l'abonné Doctrine dans l'application Symfony en créant un nouveau service pour celui-ci et en le marquant avec la doctrine.event_subscriber balise:

```yaml
# config/services.yaml
services:
    # ...

    App\EventListener\DatabaseActivitySubscriber:
        tags:
            - { name: 'doctrine.event_subscriber' }
```

Si vous devez associer l'abonné à une connexion Doctrine spécifique, vous pouvez le faire dans la configuration du service:

```yaml
# config/services.yaml
services:
    # ...

    App\EventListener\DatabaseActivitySubscriber:
        tags:
            - { name: 'doctrine.event_subscriber', connection: 'default' }
```