

Beste de savoir

Créez une API REST avec Symfony 3

12 août 2019

Table des matières

| | |
|--|-----------|
| I. Un tour d’horizon des concepts REST | 6 |
| 1. REST en quelques mots | 8 |
| 1.1. Origine de REST | 8 |
| 1.2. Présentation de REST | 8 |
| 1.2.1. Principes REST | 8 |
| 1.2.2. Un peu de vocabulaire autour de REST | 10 |
| 2. Pourquoi utiliser REST | 11 |
| 2.1. Pourquoi REST | 11 |
| 2.1.1. Les avantages de l’architecture REST | 11 |
| 2.1.2. Popularisation | 12 |
| 2.2. REST appliqué au WEB avec le protocole HTTP | 13 |
| 2.2.1. La séparation Client-Serveur | 13 |
| 2.2.2. Sans état (Stateless) | 14 |
| 2.2.3. Le Cache | 14 |
| 2.2.4. La gestion des ressources | 14 |
| 2.3. Ce cours | 15 |
| 2.3.1. Notre application Web | 15 |
| 2.3.2. Description du besoin | 15 |
| 2.3.3. Technologies utilisées | 15 |
| II. Développement de l’API REST | 17 |
| 3. Notre environnement de développement | 18 |
| 3.1. Environnement technique | 18 |
| 3.1.1. Plateforme | 18 |
| 3.1.2. Symfony Installer | 18 |
| 3.1.3. Composer | 19 |
| 3.1.4. Installation de Git | 20 |
| 3.2. Création d’un projet Symfony | 20 |
| 3.2.1. Utilisons l’installateur de Symfony | 20 |
| 3.2.2. Configuration de Apache | 21 |
| 4. Premières interactions avec les ressources | 24 |
| 4.1. Lire une collection | 24 |
| 4.1.1. Notre première ressource | 24 |
| 4.1.2. Les collections dans REST | 24 |
| 4.1.3. Le nommage d’une ressource | 25 |

| | | |
|-----------|---|-----------|
| 4.1.4. | Accéder aux lieux déclarés dans l'application | 25 |
| 4.1.5. | Pratiquons avec les utilisateurs | 31 |
| 4.2. | Lire une ressource | 35 |
| 4.2.1. | Accéder à un seul lieu | 35 |
| 4.3. | Les codes de statut (status code) pour des messages plus expressifs | 38 |
| 4.3.1. | Quel code de statut utilisé ? | 38 |
| 4.3.2. | Gérer une erreur 404 | 39 |
| 5. | FOSRestBundle et Symfony à la rescousse | 43 |
| 5.1. | Installation de FOSRestBundle | 43 |
| 5.2. | Routage avec FOSRestBundle | 45 |
| 5.2.1. | Routage automatique | 45 |
| 5.2.2. | Routage manuel | 49 |
| 5.3. | Quid de l'attribut <code>_format</code> ? | 51 |
| 5.4. | Gestion des réponses avec FOSRestBundle | 53 |
| 5.4.1. | Configuration du gestionnaire de vue | 53 |
| 5.4.2. | La cerise sur le gâteau : Format automatique et réponse sans l'objet View | 56 |
| 5.5. | Pratiquons avec notre code | 58 |
| 6. | Créer et supprimer des ressources | 61 |
| 6.1. | Création d'une ressource | 61 |
| 6.1.1. | Quelle est la ressource cible ? | 61 |
| 6.1.2. | Quel verbe HTTP ? | 62 |
| 6.1.3. | Le corps de notre requête | 62 |
| 6.1.4. | Quel code de statut HTTP ? | 62 |
| 6.1.5. | Créer un nouveau lieu | 63 |
| 6.1.6. | Pratiquons avec les utilisateurs | 71 |
| 6.2. | Suppression d'une ressource | 74 |
| 6.2.1. | Suppression d'un lieu | 75 |
| 6.2.2. | Pratiquons avec les utilisateurs | 77 |
| 7. | Mettre à jour des ressources | 80 |
| 7.1. | Mise à jour complète d'une ressource | 80 |
| 7.1.1. | Quelle est la ressource cible ? | 80 |
| 7.1.2. | Quel verbe HTTP ? | 80 |
| 7.1.3. | Le corps de notre requête | 81 |
| 7.1.4. | Quel code de statut HTTP ? | 81 |
| 7.1.5. | Mise à jour d'un lieu | 81 |
| 7.1.6. | Pratiquons avec les utilisateurs | 84 |
| 7.2. | Mise à jour partielle d'une ressource | 86 |
| 7.2.1. | À la rencontre de PATCH | 86 |
| 7.2.2. | Mise à jour partielle d'un lieu | 87 |
| 7.2.3. | Pratiquons avec les utilisateurs | 91 |
| 7.2.4. | Gestion des erreurs avec FOSRestBundle | 92 |
| 7.3. | Notre application vu selon le modèle de Richardson | 92 |
| 8. | Relations entre ressources | 94 |
| 8.1. | Hiérarchie entre ressources : la notion de sous-ressources | 94 |
| 8.1.1. | Un peu de conception | 94 |

| | |
|---|------------|
| 8.1.2. Pratiquons avec les lieux | 95 |
| 8.2. Les groupes avec le sérialiseur de Symfony | 103 |
| 8.3. Mise à jour de la suppression d'une ressource | 108 |
| 9. TP : Le clou du spectacle - Proposer des suggestions aux utilisateurs | 110 |
| 9.1. Énoncé | 110 |
| 9.2. Détails de l'implémentation | 111 |
| 9.3. Travail préparatoire | 112 |
| 9.3.1. Gestion des thèmes pour les lieux | 112 |
| 9.3.2. Gestions des préférences | 119 |
| 9.4. Proposer des suggestions aux utilisateurs | 124 |
| 9.4.1. Calcul du niveau de correspondance | 124 |
| 9.4.2. Appel API pour récupérer les suggestions | 126 |
| 10. REST à son paroxysme | 131 |
| 10.1. Supporter plusieurs formats de requêtes et de réponses | 131 |
| 10.1.1. Cas des requêtes | 131 |
| 10.1.2. Cas des réponses | 132 |
| 10.1.3. La négociation de contenu | 134 |
| 10.2. L'Hypermédia | 137 |
| | |
| III. Amélioration de l'API REST | 139 |
| | |
| 11. Sécurisation de l'API 1/2 | 140 |
| 11.1. Connexion et déconnexion avec une API | 140 |
| 11.2. Login et mot de passe pour les utilisateurs | 141 |
| 11.3. Création d'un token | 148 |
| | |
| 12. Sécurisation de l'API 2/2 | 156 |
| 12.1. Exploitions le token grâce à Symfony | 156 |
| 12.2. Gestion des erreurs avec FOSRestBundle | 164 |
| 12.3. 401 ou 403, quand faut-il utiliser ces codes de statut ? | 169 |
| 12.4. Suppression d'un token ou la déconnexion | 169 |
| | |
| 13. Créer une ressource avec des relations | 173 |
| 13.1. Rappel de l'existant | 173 |
| 13.2. Création d'un lieu avec des tarifs | 173 |
| 13.2.1. Un peu de conception | 173 |
| 13.2.2. Implémentation | 174 |
| 13.3. Bonus : Une validation plus stricte | 179 |
| 13.3.1. Création d'un lieu avec deux prix de même type | 179 |
| 13.3.2. Validation personnalisée avec Symfony | 180 |
| | |
| 14. Quand utiliser les query strings ? | 184 |
| 14.1. Pourquoi utiliser les query strings ? | 184 |
| 14.2. Gestion des query strings avec FOSRestBundle | 185 |
| 14.2.1. L'annotation QueryParam | 185 |
| 14.2.2. Le listener | 185 |

| | |
|---|------------|
| 14.3. Paginer et Trier les réponses | 186 |
| 14.3.1. Paginer la liste de lieux | 186 |
| 14.3.2. Trier la liste des lieux | 190 |
| 15. JMSSerializer : Une alternative au sérialiseur natif de Symfony | 194 |
| 15.1. Pourquoi utiliser JMSSerializerBundle ? | 194 |
| 15.2. Installation et configuration de JMSSerializerBundle | 195 |
| 15.2.1. Installation de JMSSerializerBundle | 195 |
| 15.2.2. Configuration de JMSSerializerBundle | 195 |
| 15.2.3. Sérialiser les attributs même s'ils sont nuls | 197 |
| 15.3. Impact sur l'existant | 197 |
| 15.3.1. Tests de la configuration | 197 |
| 15.3.2. Mise à jour de nos règles de sérialisation | 198 |
| 16. La documentation avec OpenAPI (Swagger RESTFul API) | 201 |
| 16.1. Qu'est-ce que OpenAPI ? | 201 |
| 16.2. Rédaction de la documentation | 202 |
| 16.2.1. Quel outil pouvons-nous utiliser pour créer la documentation ? | 202 |
| 16.2.2. Structure de base du fichier swagger.json | 202 |
| 16.2.3. Déclarer une opération avec l'API | 203 |
| 16.3. Installer et utiliser Swagger UI | 205 |
| 16.3.1. Installation de Swagger UI | 205 |
| 16.3.2. Utiliser notre documentation | 206 |
| 17. Automatiser la documentation avec NelmioApiDocBundle | 208 |
| 17.1. Installation de NelmioApiDocBundle | 208 |
| 17.2. L'annotation ApiDoc | 208 |
| 17.2.1. Configuration | 208 |
| 17.2.2. Intégration avec FOSRestBundle | 210 |
| 17.2.3. Définir le type des réponses de l'API | 210 |
| 17.2.4. Définir le type des payloads des requêtes | 212 |
| 17.2.5. Gérer plusieurs codes de statut | 213 |
| 17.3. Étendre NelmioApiDocBundle | 215 |
| 17.3.1. Pourquoi étendre le bundle ? | 215 |
| 17.3.2. Correction du format de sortie des réponses en erreur | 215 |
| 17.4. Le bac à sable | 216 |
| 17.4.1. Configuration du bac à sable | 216 |
| 17.4.2. Documentation pour la création de token | 217 |
| 17.4.3. Tester le bac à sable | 217 |
| 17.5. Générer une documentation compatible OpenAPI | 218 |
| 18. FAQ | 220 |
| 18.1. Comment générer des pages HTML depuis l'application Symfony 3 ? | 220 |
| 18.1.1. Utiliser plusieurs règles dans le <code>format_listener</code> | 220 |
| 18.1.2. Configurer le <code>zone_listener</code> | 221 |
| 18.2. Comment autoriser l'accès à certaines urls avec notre système de sécurité ? | 222 |

REST s'est imposé dans le monde du web comme étant un paradigme approuvé et éprouvé pour concevoir des APIs (Application Programming Interface).

Table des matières

De grandes entreprises comme [Github](#) , Facebook ([Graph](#)) ou [YouTube](#) l'utilisent pour fournir des APIs largement utilisées pour accéder à leurs services.

À l'ère des sites web en Single Page Applications et des applications mobiles (Android, IOS ou encore Windows Phone), savoir développer une API est devenu incontournable.

Pourquoi utiliser REST plutôt qu'une autre technologie ou architecture ? Quels avantages cela peut-il nous apporter ? Comment développer une API REST avec Symfony ?

Tout au long de ce cours, nous allons apprendre à mettre en œuvre les principes de REST pour développer rapidement une application web fiable et extensible avec le framework Symfony et l'un de ses bundles phares *FOSRestBundle*.

Première partie

Un tour d'horizon des concepts REST

I. Un tour d'horizon des concepts REST

Nous allons explorer les origines de REST et l'ensemble des concepts et contraintes qui sont autour.

?

Pourquoi utiliser REST plutôt qu'une autre technologie ou architecture ? Quelles avantages cela peut-il nous apporter ?

1. REST en quelques mots

1.1. Origine de REST

REST (Representational State Transfer) est un style d'architecture pour les systèmes hypermédia distribués, créé par un dénommé Roy Thomas Fielding en 2000 et décrit dans le chapitre 5 de sa thèse de doctorat intitulée [Architectural Styles and the Design of Network-based Software Architectures](#) [↗](#) (Les styles d'architecture et la conception de l'architecture des applications réseaux).

[Roy Fielding](#) [↗](#) est un ingénieur en informatique qui a notamment travaillé sur les spécifications du protocole HTTP Hypertext Transfert Protocol. Il est aussi connu comme l'un des membres fondateur de la fondation Apache.

1.2. Présentation de REST

1.2.1. Principes REST

Le style d'architecture REST représente un ensemble de contraintes qui régissent une application réseau. Chacune de ces contraintes décrit un concept qu'une application qui se veut RESTful doit implémenter.



Le terme RESTful (anglicisme) est un adjectif qui qualifie une application qui suit les principes REST.

1.2.1.1. Client-Serveur (Client-Server)

La première contrainte est la séparation des responsabilités entre le client et le serveur. Le serveur s'occupe de la gestion des règles métier et des données et le client se concentre sur l'interface utilisateur (une interface peut être une page web, une application mobile etc.). En séparant le client et le serveur, la portabilité et la scalabilité de notre application sont grandement améliorées. Chaque composant pourra aussi évoluer séparément. Nous pouvons imaginer un site web qui refait toute sa charte graphique sans que le code côté serveur ne soit modifié.

I. Un tour d'horizon des concepts REST

1.2.1.2. Sans état (Stateless)

Une autre contrainte est la notion de "Sans état" ou Stateless en anglais. La communication entre le client et le serveur doit se faire sans dépendre d'un contexte lié au serveur. Chaque requête du client contient toutes les informations nécessaires à son traitement. Ainsi, plusieurs instances de serveurs peuvent traiter indifféremment les requêtes de chaque client.

1.2.1.3. Cache

Afin d'améliorer les performances de l'application, le serveur doit ajouter *une étiquette de cache* à toutes les réponses. Cette étiquette décrit les possibilités de mise en cache ou non des données renvoyées par le serveur.

1.2.1.4. Interface uniforme (Uniform Interface)

Une des fonctionnalités clés qui permet de distinguer une architecture REST est la mise en valeur d'une interface uniforme entre les différents composants.

REST repose sur 4 contraintes d'interface :

- l'identification de manière unique des ressources ;
- l'interaction avec les ressources via des représentations, chaque ressource disposant de sa présentation ;
- les messages auto-descriptifs, une réponse ou une requête contient toutes les informations permettant de décrire la nature des données qu'elle contient et les interactions possibles ;
- et, l'hypermédia en tant que moteur de l'état de l'application *HATEOAS*. L'état de l'application, les différentes interactions possibles entre client et le serveur doivent être décrites à travers les liens hypermédia dans les réponses du serveur.

Le terme lien hypermédia englobe des formulaires, les liens hypertextes ou plus généralement tout support numérique permettant une interaction.

En définissant une interface uniformisée, les différentes interactions avec le serveur sont facilement identifiables.

1.2.1.5. Organisation en couches (Layered System)

Les couches dans une application consistent en l'isolation des différents composants de l'application pour bien organiser leurs responsabilités. Chaque couche représente alors un système borné qui traite une problématique spécifique de notre application. Nous pouvons prendre comme exemple une couche dédiée au stockage des données mais qui n'a pas conscience de leur origine. Son unique rôle consiste à stocker des informations qui lui sont passées.

I. Un tour d'horizon des concepts REST

1.2.1.6. Code à la demande (Code-On-Demand)

Cette contrainte **optionnelle** permet l'extension des fonctionnalités du client en fournissant du code téléchargeable et exécutable. Cela nécessite quand même une certaine connaissance des clients qui exploitent l'application REST. Par exemple, une API pourrait fournir du code JavaScript que tous les clients web peuvent télécharger et exécuter pour effectuer des tâches complexes. Cela permet de faire évoluer un client sans avoir à le redéployer vu que le code exécuté vient du serveur. Il suffira juste de mettre à jour le serveur et le tour est joué.

1.2.2. Un peu de vocabulaire autour de REST

Ce style d'architecture introduit et utilise par la même occasion quelques notions qu'il faut impérativement comprendre.

1.2.2.1. Ressources et identifiants

Une interface REST gravite autour de ressources. À partir du moment où vous devez interagir avec une entité de votre application, créer une entité, la modifier, la consulter ou encore l'identifier de manière unique, vous avez pour la plupart des cas une ressource. Si par exemple, nous développons une application de commerce en ligne, un article disponible à la vente est une ressource. Une image décrivant cet article peut être une ressource. Pour référencer de manière unique cette ressource, REST utilise un identifiant. Cet identifiant sera alors utilisé par *tous* les différents composants de notre application afin d'interagir avec cette ressource. Notre article pourra ainsi avoir un numéro unique que les composants du panier, de paiement ou autres utiliseront pour désigner cet article.

1.2.2.2. Représentation d'une ressource

Une représentation désigne toutes les informations (données et métadonnées) utiles qui décrivent une ressource.

Notre article pourra donc être représenté par une page HTML (Hypertext Markup langage) contenant le nom de l'article, son prix, sa disponibilité etc. Et notre image décrivant un article, sa représentation désignera simplement les données en base64 et les métadonnées qui décrivent l'encodage utilisée pour l'image, le type de compression, etc.



En résumé, REST est un style d'architecture défini par un ensemble de contraintes qui régissent l'organisation d'une application et les procédés de communication entre un fournisseur de services (le serveur) et le consommateur (le client).

2. Pourquoi utiliser REST



Pourquoi utiliser REST plutôt d'une autre technologie ou architecture ?

Il existe plusieurs moyens permettant de communiquer entre des composants dans le cadre d'une architecture de type SOA (Service oriented Architecture). On peut citer le protocole SOAP (Simple Object Access Protocol) ou encore XML-RPC.

Ces technologies sont largement utilisées surtout dans un cadre d'entreprise, mais avec l'essor du web, elles ont commencé à montrer leurs limites.

REST étant conçu pour répondre à ce besoin spécifique - le web - ce style d'architecture théorisé par Roy Fielding présente intrinsèquement beaucoup d'avantages pour ce cas d'usage.

Dans cette partie de ce cours, nous allons donc voir les facilités et l'intérêt que REST pourrait nous apporter dans le cadre du développement d'une API web.

2.1. Pourquoi REST

2.1.1. Les avantages de l'architecture REST

Comme les différentes règles et design pattern appliqués en génie logiciel, les différentes contraintes qu'impose l'architecture REST permettent d'obtenir des applications de meilleure qualité.


On peut citer entre autres :

- un couplage plus faible entre le client et le serveur comparé aux méthodes du type RPC Remote Procedure Call comme SOAP ;
- une uniformisation des APIs (Application Programming Interface) pour une facilité d'utilisation ;
- une plus grande tolérance à la panne ;
- ou encore une application facilement portable et extensible.

2.1.2. Popularisation

Bien que la publication de la thèse de Roy Fielding date des années 2000, un livre de Leonard Richardson et Sam Ruby *RESTful Web Services*, sorti le 22 mai 2007, a popularisé le style d'architecture REST en proposant une méthodologie pour l'implémenter en utilisant le protocole HTTP Hypertext Transfert Protocol.

Comme vous l'aurez déjà remarqué, plus nous avançons dans les principes REST, plus le modèle devient contraignant. Dès lors, une application peut suivre ces principes sans pour autant remplir toutes les contraintes du REST.

Ainsi, lors de la conférence [QCon du 20 novembre 2008](#) , Richardson a présenté un modèle qui permet d'évaluer son application selon les principes REST. Ce modèle est connu sous le nom de : Modèle de maturité de Richardson.

2.1.2.1. Niveau 0 : RPC (Remote Procedure Call) via HTTP

Le protocole HTTP est utilisé pour appeler des méthodes du serveur. C'est le niveau des API Json RPC ou encore SOAP.

2.1.2.2. Niveau 1 : Identification des ressources

Les entités avec lesquels les interactions ont lieu sont identifiées en tant que ressources.

2.1.2.3. Niveau 2 : Utilisation des verbes HTTP

Les interactions avec le serveur se font avec plusieurs verbes HTTP différents en **respectant** leurs sémantiques. Les opérations avec une ressource se font via un même identifiant mais avec des verbes différents.

Par exemple, le verbe *GET* pour récupérer du contenu ou *DELETE* pour le supprimer. En l'occurrence, le Json RPC utilise le verbe *POST* pour toutes ces opérations et par conséquent ne respecte pas ce modèle.



Les verbes HTTP appelés aussi méthodes permettent de décrire avec une sémantique claire l'opération que nous voulons effectuer. Nous pouvons citer les plus courantes qui sont *GET*, *POST*, *PUT* et *DELETE*.

Pour finir les codes de statut du protocole permettent d'avoir des réponses plus expressives. Une réponse avec un code 404 permettra au client d'identifier que la ressource demandée n'existe pas. Nous verrons plus en détails quelles sont les méthodes et codes de statut que nous pouvons utiliser dans la suite de ce cours.

2.1.2.4. Niveau 3 : Contrôles hypermédia.

Comme déjà décrit dans la partie *Présentation de REST > Interface uniforme*, le contrôle hypermédia désigne l'état d'une application ou API avec un seul point d'entrée mais qui propose des éléments permettant de l'explorer et d'interagir avec elle. Un bon exemple est le site web. Si par exemple, nous accédons à YouTube, la page d'accueil nous propose des liens vers des vidéos ou encore un formulaire de recherche. Ces éléments hypermédia permettent ainsi de visualiser toutes sortes de contenus sans connaître au préalable les liens directs les identifiants.



FIGURE 2.1. – Modèle de maturité de Richardson

Notre objectif sera de se rapprocher le plus possible de l'architecture REST sans oublier les contraintes que le monde réel va nous imposer.

2.2. REST appliqué au WEB avec le protocole HTTP

Comme l'a dit Roy Fielding dans le chapitre 6 de sa thèse, l'objectif de REST était de créer un model architectural décrivant comment le web devrait fonctionner, le permettant de devenir ainsi une référence pour les protocoles web. REST a été conçu en évitant de violer les contraintes principales qui régissent le web.

As described in Chapter 4, the motivation for developing REST was to create an architectural model for how the Web should work, such that it could serve as the guiding framework for the Web protocol standards. REST has been applied to describe the desired Web architecture, help identify existing problems, compare alternative solutions, and ensure that protocol extensions would not violate the core constraints that make the Web successful.

Le protocole de transfert HTTP dispose de beaucoup de spécificités que nous pouvons donc mettre en oeuvre avec le style d'architecture REST. Nous verrons comment mettre à profit ces spécifications afin de remplir les exigences d'une application dite RESTful.

2.2.1. La séparation Client-Serveur

L'essence même du HTTP - protocole de transfert hypertexte - comme son nom l'indique est de permettre le transfert de données entre un client et un serveur. Dès lors, les applications web remplissent de-facto cette contrainte d'architecture.

L'utilisation de HTTP dans le cadre de REST pour une bonne isolation client-serveur est donc un choix judicieux et très largement répandu.

I. Un tour d'horizon des concepts REST

2.2.2. Sans état (Stateless)

Il suffit de consulter le résumé de la même [RFC 7231](#) pour voir que :

The Hypertext Transfer Protocol (HTTP) is a stateless application-level protocol for distributed, collaborative, hypertext information systems.

Le protocole de transfert hypertexte (HTTP) est un protocole **sans état** de la [couche application](#) (se référer au modèle OSI) pour les systèmes d'informations hypermédia distribués et collaboratifs.

Le protocole HTTP est stateless (sans état) par définition. Même si nous pouvons retrouver des applications web qui dérogent à cette contrainte, il faut retenir que HTTP a été pensé pour fonctionner sans état.

2.2.3. Le Cache

Là aussi, le protocole HTTP supporte nativement [la gestion du cache](#) via les entêtes comme *Cache-Control*, *Expires*, etc. Ces entêtes permettent de réutiliser une même réponse si le contenu est considéré comme étant à jour comme le préconise le style d'architecture REST afin d'améliorer les performances de notre application.

2.2.4. La gestion des ressources

2.2.4.1. Identification

Nous avons déjà défini une ressource dans le cadre de REST et pourquoi il fallait l'identifier de manière unique. Le protocole HTTP utilise là aussi une notion bien connue : l'URI (Uniform Resource Identifier). En effet, lorsque nous consultons la [RFC 2731](#) de HTTP 1.1, nous pouvons voir que une ressource est définie comme étant :

The target of an HTTP request is called a "resource". HTTP does not limit the nature of a resource; it merely defines an interface that might be used to interact with resources. Each resource is identified by a Uniform Resource Identifier (URI), as described in Section 2.7 of [RFC7230].

La cible d'une requête HTTP est appelé une « ressource ». HTTP ne met pas de limitation sur la nature d'une ressource ; il définit seulement une interface qui peut être utilisé pour interagir avec des ressources. Chacune de ces ressources est identifiée par une URI (Uniform Resource Identifier), comme décrit dans la section 2.7 de la [RFC7230].

2.2.4.2. Représentation

Une représentation est toute information destinée à refléter l'état passé, actuel ou voulu d'une ressource donnée.

I. Un tour d'horizon des concepts REST

For the purposes of HTTP, a "representation" is information that is intended to reflect a past, current, or desired state of a given resource, ...

RFC 7231 [↗](#)

Ainsi avec les URI et les représentations des réponses HTTP (html, xml, json, etc.), nous pouvons satisfaire la contrainte 4 d'interface uniforme de REST pour mettre en place notre application.

2.3. Ce cours

2.3.1. Notre application Web

Durant ce cours, nous allons développer une API permettant de gérer des idées et suggestions de sorties récréatives en se basant sur les concepts REST. Cette application va nous servir de fil conducteur pour ce cours et toutes ses fonctionnalités seront détaillées plus tard.

Les prérequis pour suivre ce cours, il faut des connaissances minimum de Symfony 2.7 à 3.* :

- créer une application avec Symfony ;
- Utiliser Doctrine 2 avec Symfony ;
- Utiliser l'injection de dépendances de Symfony.

Les objectifs de ce cours sont entre autres de :

- Comprendre l'architecture REST ;
- Mettre en place une API RESTful (Créer une API uniforme et facile à utiliser) ;
- Apprendre comment sécuriser une API (REST en particulier) ;
- Savoir utiliser les avantages de Symfony dans ses développements (Composants et Bundles).

2.3.2. Description du besoin

Nous allons mettre en place une application permettant de gérer des idées et suggestions de sorties récréatives. L'application dispose de plusieurs lieux (restaurants, centre de loisirs, cinéma etc) connus et réputés et de plusieurs utilisateurs avec leurs centres d'intérêt. L'objectif est de proposer un mécanisme permettant de proposer à chaque utilisateur une idée de sortie la plus pertinente en se basant sur ses préférences.

2.3.3. Technologies utilisées

Les exemples présentés se baseront sur Symfony 3 avec *FOSRestBundle*. Les tests de l'API se feront avec cURL (utilitaire en ligne de commande) et le logiciel Postman (extension du navigateur Chrome).

I. Un tour d'horizon des concepts REST

Le protocole HTTP se prête bien au jeu de REST. À l'heure actuelle, la plupart des API RESTful l'utilisent vu que les technologies pour l'exploiter sont très largement répandues.

Ici prend fin l'aparté sur la partie théorique de ce cours. La suite sera grandement axée sur la pratique, tous les concepts seront abordés en se basant sur des exemples concrets.

Nous allons donc voir comment appliquer les concepts et contraintes REST dans une application web. Cela nous offrira une API uniforme avec une prise en main facile. L'objectif est d'avoir à la fin de ce cours une API pleinement fonctionnelle.

Deuxième partie

Développement de l'API REST

3. Notre environnement de développement

Afin d'avoir un environnement de développement de référence pendant ce cours, nous allons voir ensemble les technologies qui seront utilisées et surtout à quelles versions.

Vous pourrez ainsi tester les différents codes qui seront fournis. Il est utile de rappeler que pour suivre ce cours vous devez avoir un minimum de connaissances en PHP et Symfony. Certains aspects de configuration comme l'installation de MySQL, Apache ou autres ne seront pas abordés. Si vous n'avez jamais procédé à l'installation de Symfony, il est préférable de se documenter sur le sujet avant de commencer ce cours.

3.1. Environnement technique

3.1.1. Plateforme

Nous avons ci-dessous un tableau récapitulatif des différentes technologies et la version utilisée. Le système d'exploitation utilisé importe peu et vous pouvez donc utiliser celui de votre choix pour suivre le reste du cours. Sur Windows, vous avez la suite [WAMP](#) et son équivalent [LAMP](#) sur Ubuntu.

| Technologie | Version | Exemples |
|-------------|---------|---------------|
| PHP | 7.0.x | 7.0.0, 7.0.3 |
| MySQL | 5.7.x | 5.7.0, 5.7.9 |
| Apache | 2.4.x | 2.4.0, 2.4.17 |

3.1.2. Symfony Installer

La méthode recommandée pour installer un projet Symfony est d'utiliser [l'installateur](#). Cet utilitaire nous permettra d'installer Symfony avec la version que nous souhaitons.

3.1.2.1. Installation sur Linux

Il suffit de lancer dans la console :

II. Développement de l'API REST

```
1 curl -Ls https://symfony.com/installer -o ~/bin/symfony
2 chmod a+x ~/bin/symfony
```

Cela téléchargera l'installateur et le placera dans le répertoire bin de l'utilisateur connecté.

3.1.2.2. Installation sous Windows :

Avant tout, il faut s'assurer que l'exécutable de PHP est bien disponible dans l'invite de commande. Des consignes d'installation sont disponibles sur [le site officiel de PHP](#) . Ensuite, Il suffit exécuter dans l'invite de commande :

```
1 c:\> php -r "readfile('https://symfony.com/installer');" > symfony
```

Ensuite, il est judicieux de créer un fichier `symfony.bat` contenant `@php "%~dp0symfony" %*`.



Il est possible de placer les fichiers `symfony` et `symfony.bat` dans un même dossier que vous rajouter dans le PATH avec les variables d'environnement de Windows afin d'accéder à la commande partout.

Une fois l'installation finie, lancer la commande `symfony` pour vérifier le bon fonctionnement du tout.

```
1 symfony
2 # réponse attendue
3 Symfony Installer (1.5.0)
4 =====
5
6 This is the official installer to start new projects based on the
7 Symfony full-stack framework.
```

3.1.3. Composer

Nous utiliserons [Composer](#) pour rajouter de nouvelles dépendances à notre projet.

Il suffit d'utiliser l'installateur disponible sur [le site officiel](#) .

Pour tester le bon fonctionnement, il faut lancer la commande `composer` :

II. Développement de l'API REST

Accédez à l'URL de vérification de Symfony et effectuez les correctifs si nécessaires <http://localhost:8000/config.php> .



FIGURE 3.1. – Page de vérification de la configuration de Symfony

3.2.1.2. Problème de certificats SSL

Sous Windows, il est possible de rencontrer des problèmes de certificats.

```
1 [GuzzleHttp\Exception\RequestException]
2 cURL error 60: SSL certificate problem: unable to get local issuer
   certificate
```

Pour corriger ce problème, il faut s'assurer que l'extension OpenSSL est activée et définir le chemin d'accès vers le fichier contenant les certificats racines.

Une liste de certificats est disponible sur <https://curl.haxx.se/ca/cacert.pem> . Pensez à le télécharger.

Commençons par identifier le fichier de configuration de PHP. Avec WAMP, ce fichier se situe dans le dossier d'installation (par exemple, D:\wamp64\bin\php\php7.0.0\php.ini). Il suffit maintenant de vérifier que la ligne concernant l'extension OpenSSL n'est pas commenté et de spécifier le chemin du fichier contenant les certificats racines.

```
1 extension=php_openssl.dll
2 [openssl]
3 openssl.cafile=D:\wamp64\bin\php\php7.0.0\cacert.pem
4 ;openssl.capath=
```

3.2.2. Configuration de Apache

Durant le reste du cours, j'accéderai à l'API en utilisant un virtual host apache personnalisé. Notre API sera donc disponible sur l'URL `http://rest-api.local`.

Pour ce faire, il faut [configurer un virtual host apache](#) et modifier le fichier host du système pour renseigner l'URL `rest-api.local`.

II. Développement de l'API REST



Le virtual host fourni est compatible avec Windows. Penser à remplacer `D:/wamp64/www/rest_api` par votre dossier d'installation et à effectuer les adaptations nécessaires pour un autre système d'exploitation.

```
1 <VirtualHost *:80>
2     ServerName rest-api.local
3
4     DocumentRoot "D:/wamp64/www/rest_api/web"
5
6     <Directory "D:/wamp64/www/rest_api/web">
7         DirectoryIndex app_dev.php
8         Require all granted
9         AllowOverride None
10
11         RewriteEngine On
12         RewriteCond %{REQUEST_FILENAME} -f
13         RewriteRule ^ - [L]
14         RewriteRule ^ app_dev.php [L]
15     </Directory>
16
17     # Ajuster le chemin vers les fichiers de logs à votre
18     # convenance
19     ErrorLog logs/rest-api-error.log
20     CustomLog logs/rest-api-access.log combined
21 </VirtualHost>
```



Le mode rewrite d'apache est obligatoire pour que ce virtual host fonctionne. Notez aussi que les requêtes seront redirigées directement vers `app_dev.php` avec cette configuration.

Ensuite sous Windows, éditez en tant qu'administrateur le fichier `C:\Windows\System32\drivers\etc\hosts` et sous Linux, éditez avec les droits root le fichier `/etc/hosts`, et rajouter une ligne :

```
1 127.0.0.1 rest-api.local
2 ::1 rest-api.local
```

Sous Windows, l'astuce consiste à lancer votre éditeur de texte en tant qu'administrateur avant d'ouvrir le fichier à éditer.

Maintenant en accédant à l'URL <http://rest-api.local/> , nous atteignons notre page web de bienvenue.



FIGURE 3.2. – Page d'accueil de notre futur site

Maintenant que nous avons un environnement de développement fonctionnel, nous allons mettre en place toutes les briques nécessaires pour avoir une API REST complète. Les choses sérieuses peuvent maintenant commencer. L'outil [Postman](#) [↗](#) sera utilisé pour effectuer tous les tests de notre API. Il est donc grandement recommandé de l'installer avant de continuer.

4. Premières interactions avec les ressources



Pourquoi parle-t-on tant des ressources ?

Au-delà de connaître la définition d'une ressource en REST, un des principaux problèmes lorsque nous développons une API est de savoir quelles sont les entités de notre projet qui sont éligibles. Dans cette partie du cours, nous allons donc voir comment bien identifier une ressource avec une expression du besoin plus claire et aussi comment les exposer avec une URI (Uniform Resource Identifier).

4.1. Lire une collection

4.1.1. Notre première ressource

Une interface REST gravite autour de ressources. À partir du moment où vous devez interagir avec une entité de votre application, créer une entité, la modifier, la consulter ou encore l'identifier de manière unique, vous avez pour la plupart des cas une ressource.

L'application que nous devons développer enregistre plusieurs **lieux** (monuments, centres de loisirs, châteaux, etc.) et fait des suggestions de sorties/visites à des **utilisateurs**.

Dans notre application, nous aurons donc un lieu avec éventuellement les informations permettant de le décrire (nom, adresse, thème, réputation, etc.).

Nous serons sûrement appelés à le consulter ou à l'éditer. Voici donc notre première ressource : un lieu.

Le choix des ressources dans une API REST est très important mais leur nommage l'est autant car c'est cela qui permettra d'avoir une API cohérente.

4.1.2. Les collections dans REST

A ce stade du cours, la notion de ressource doit être bien comprise. Mais il existe aussi une autre notion qui sera utile dans la conception d'une API REST : les collections.

Une collection désigne simplement un ensemble de ressources d'un même type. Dans notre cas, la liste de tous les lieux référencés dans l'application représente une collection. Et c'est idem pour la liste des utilisateurs.

4.1.3. Le nommage d'une ressource

Une règle d'or à respecter, c'est la cohérence. Il faut choisir des noms de ressources simples et suivre une même logique de nommage. Si par exemple, une ressource est nommée au pluriel alors elle doit l'être sur toute l'API et toutes les ressources doivent être aussi au pluriel. La casse est également très importante pour la cohérence. Il faudra ainsi respecter la même casse pour toutes les ressources.

Pour le cas de notre exemple, toutes nos ressources seront en minuscule, au pluriel et en anglais. C'est un choix assez répandu dans les différentes API publiques à notre disposition.

Donc pour une collection représentant les lieux à visiter, nous aurons **places**. Dans notre URL, nous aurons alors `rest-api.local/places`.

4.1.4. Accéder aux lieux déclarés dans l'application

Pour commencer, nous considérons qu'un lieu a un nom et une adresse. L'objectif est d'avoir un appel de notre API permettant d'afficher tous les lieux connus par notre application.

4.1.4.1. La sémantique HTTP

La ressource avec laquelle nous souhaitons interagir est **places**. Notre requête HTTP doit donc se faire sur l'URL `rest-api.local/places`.

?

Quelle méthode (ou verbe) HTTP utiliser : *GET*, *POST*, ou *DELETE* ?

Comme expliqué dans le modèle de maturité de Richardson, une API qui se veut RESTful doit utiliser les méthodes HTTP à bon escient pour interagir avec les ressources. Dans notre cas, nous voulons lire des données disponibles sur le serveur. Le protocole HTTP nous propose la méthode **GET** qui, selon la [RFC 7231](#), est la méthode de base pour récupérer des informations.



FIGURE 4.1. – Cinématique de récupération des lieux

4.1.4.2. Implémentation

Nous allons commencer par mettre en place notre appel API avec de fausses données, ensuite nous mettrons en place la persistance de celles-ci avec Doctrine.

Tout d'abord, il faut créer une entité nommée **Place** contenant un nom et une adresse :

II. Développement de l'API REST

```
1 # src/AppBundle/Entity/Place.php
2 <?php
3 namespace AppBundle\Entity;
4
5 class Place
6 {
7     public $name;
8
9     public $address;
10
11     public function __construct($name, $address)
12     {
13         $this->name = $name;
14         $this->address = $address;
15     }
16 }
```

Créons maintenant un nouveau contrôleur nommé *PlaceController* qui s'occupera de la gestion des lieux avec, pour l'instant, une seule méthode permettant de les lister.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14     /**
15      * @Route("/places", name="places_list")
16      * @Method({"GET"})
17      */
18     public function getPlacesAction(Request $request)
19     {
20         return new JsonResponse([
21             new Place("Tour Eiffel",
22                 "5 Avenue Anatole France, 75007 Paris"),
23             new Place("Mont-Saint-Michel",
24                 "50170 Le Mont-Saint-Michel"),
25             new Place("Château de Versailles",
26                 "Place d'Armes, 78000 Versailles"),
27         ]);
28 }
```

II. Développement de l'API REST

```
25     }
26 }
```

Un appel de type `GET` sur l'URL rest-api.local/places permet d'obtenir notre liste de lieux.

```
1  [
2  {
3    "name": "Tour Eiffel",
4    "address": "5 Avenue Anatole France, 75007 Paris"
5  },
6  {
7    "name": "Mont-Saint-Michel",
8    "address": "50170 Le Mont-Saint-Michel"
9  },
10 {
11  "name": "Château de Versailles",
12  "address": "Place d'Armes, 78000 Versailles"
13 }
14 ]
```

Avec Postman :



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 4.2. – Récupération des lieux avec Postman

Nous allons maintenant récupérer nos lieux depuis la base de données avec Doctrine. Rajoutons un identifiant aux lieux et mettons en place les annotations sur l'entité *Place*.

```
1  # src/AppBundle/Entity/Place.php
2  <?php
3  namespace AppBundle\Entity;
4
5  use Doctrine\ORM\Mapping as ORM;
6
7  /**
8   * @ORM\Entity()
9   * @ORM\Table(name="places")
10  */
11  class Place
12  {
13      /**
```

II. Développement de l'API REST

```
14     * @ORM\Id
15     * @ORM\Column(type="integer")
16     * @ORM\GeneratedValue
17     */
18     protected $id;
19
20     /**
21     * @ORM\Column(type="string")
22     */
23     protected $name;
24
25     /**
26     * @ORM\Column(type="string")
27     */
28     protected $address;
29
30     public function getId()
31     {
32         return $this->id;
33     }
34
35     public function getName()
36     {
37         return $this->name;
38     }
39
40     public function getAddress()
41     {
42         return $this->address;
43     }
44
45     public function setId($id)
46     {
47         $this->id = $id;
48         return $this;
49     }
50
51     public function setName($name)
52     {
53         $this->name = $name;
54         return $this;
55     }
56
57     public function setAddress($address)
58     {
59         $this->address = $address;
60         return $this;
61     }
62 }
```

II. Développement de l'API REST

Pour des raisons de clarté, nous allons aussi modifier le nom de notre base de données.

```
1 # app/config/parameters.yml
2 parameters:
3     database_host: 127.0.0.1
4     database_port: null
5     database_name: rest_api
6     database_user: root
7     database_password: null
```

Il ne reste plus qu'à créer la base de données et la table pour stocker les lieux.

```
1 php bin/console doctrine:database:create
2 # Réponse
3 # Created database `rest_api` for connection named default
4
5 php bin/console doctrine:schema:update --dump-sql --force
6 # Réponse
7 CREATE TABLE places (id INT AUTO_INCREMENT NOT NULL, name
8     VARCHAR(255) NOT NULL, address VARCHAR(255) NOT NULL, PRIMARY
9     KEY(id)) DE
10
11 Updating database schema...
12 Database schema updated successfully! "1" query was executed
```

Le jeu de données de test :

```
1 INSERT INTO `places` (`id`, `name`, `address`) VALUES (NULL,
2     'Tour Eiffel', '5 Avenue Anatole France, 75007 Paris'), (NULL,
3     'Mont-Saint-Michel', '50170 Le Mont-Saint-Michel'), (NULL,
4     'Château de Versailles', 'Place d'Armes, 78000 Versailles')
```

Nous disposons maintenant d'une base de données pour gérer les informations de l'application. Il ne reste plus qu'à changer l'implémentation dans notre contrôleur pour charger les données avec Doctrine.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
```

II. Développement de l'API REST

```
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14     /**
15      * @Route("/places", name="places_list")
16      * @Method({"GET"})
17      */
18     public function getPlacesAction(Request $request)
19     {
20         $places = $this->get('doctrine.orm.entity_manager')
21             ->getRepository('AppBundle:Place')
22             ->findAll();
23         /* @var $places Place[] */
24
25         $formatted = [];
26         foreach ($places as $place) {
27             $formatted[] = [
28                 'id' => $place->getId(),
29                 'name' => $place->getName(),
30                 'address' => $place->getAddress(),
31             ];
32         }
33
34         return new JsonResponse($formatted);
35     }
36 }
```

En testant à nouveau notre appel, nous obtenons :

```
1 [
2   {
3     "id": 1,
4     "name": "Tour Eiffel",
5     "address": "5 Avenue Anatole France, 75007 Paris"
6   },
7   {
8     "id": 2,
9     "name": "Mont-Saint-Michel",
10    "address": "50170 Le Mont-Saint-Michel"
11  },
12  {
13    "id": 3,
14    "name": "Château de Versailles",
15    "address": "Place d'Armes, 78000 Versailles"
16  }
17 ]
```


II. Développement de l'API REST

```
17 ]
```

Avec Postman :



FIGURE 4.3. – Récupération des lieux avec Postman

4.1.5. Pratiquons avec les utilisateurs

4.1.5.1. Objectif

Maintenant que le principe pour récupérer les informations d'une liste est expliqué, nous allons faire de même avec les utilisateurs. Nous considérerons que les utilisateurs ont un nom, un prénom et une adresse mail et que la ressource pour désigner une liste d'utilisateur est **users**.

L'objectif est de mettre en place un appel permettant de générer la liste des utilisateurs enregistrés en base. Voici le format de la réponse attendue :

```
1  [
2    {
3      "id": 1,
4      "firstname": "Ab",
5      "lastname": "Cde",
6      "email": "ab.cde@test.local"
7    },
8    {
9      "id": 2,
10     "firstname": "Ef",
11     "lastname": "Ghi",
12     "email": "ef.ghi@test.local"
13   }
14 ]
```

4.1.5.2. Implémentation

4.1.5.2.1. Configuration de doctrine Comme pour les lieux, nous allons commencer par créer l'entité *User* et la configuration doctrine qui va avec :

II. Développement de l'API REST

```
1 # src/AppBundle/Entity/User.php
2 <?php
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity()
9  * @ORM\Table(name="users")
10 */
11 class User
12 {
13     /**
14      * @ORM\Id
15      * @ORM\Column(type="integer")
16      * @ORM\GeneratedValue
17      */
18     protected $id;
19
20     /**
21      * @ORM\Column(type="string")
22      */
23     protected $firstname;
24
25     /**
26      * @ORM\Column(type="string")
27      */
28     protected $lastname;
29
30     /**
31      * @ORM\Column(type="string")
32      */
33     protected $email;
34
35     public function getId()
36     {
37         return $this->id;
38     }
39
40     public function setId($id)
41     {
42         $this->id = $id;
43     }
44
45     public function getFirstname()
46     {
47         return $this->firstname;
48     }
49 }
```

II. Développement de l'API REST

```
50     public function setFirstname($firstname)
51     {
52         $this->firstname = $firstname;
53     }
54
55     public function getLastname()
56     {
57         return $this->lastname;
58     }
59
60     public function setLastname($lastname)
61     {
62         $this->lastname = $lastname;
63     }
64
65     public function getEmail()
66     {
67         return $this->email;
68     }
69
70     public function setEmail($email)
71     {
72         $this->email = $email;
73     }
74 }
```

Mettons à jour la base de données :

```
1 php bin/console doctrine:schema:update --dump-sql --force
2 # Réponse
3 #> CREATE TABLE users (id INT AUTO_INCREMENT NOT NULL, firstname
   VARCHAR(255) NOT NULL, lastname VARCHAR(255) NOT NULL, email
   VARCHAR(255) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET
   utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
4
5 #>Updating database schema...
6 #>Database schema updated successfully! "1" query was executed
```

N'oublions pas le jeu de données de test :

```
1 INSERT INTO `users` (`id`, `firstname`, `lastname`, `email`) VALUES
   (NULL, 'Ab', 'Cde', 'ab.cde@test.local'), (NULL, 'Ef', 'Ghi',
   'ef.ghi@test.local');
```

4.1.5.2.2. Création du contrôleur pour les utilisateurs Nous allons créer un contrôleur dédié aux utilisateurs. Pour l'instant, nous aurons une seule méthode permettant de les lister.

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use AppBundle\Entity\User;
11
12 class UserController extends Controller
13 {
14     /**
15      * @Route("/users", name="users_list")
16      * @Method({"GET"})
17      */
18     public function getUsersAction(Request $request)
19     {
20         $users = $this->get('doctrine.orm.entity_manager')
21             ->getRepository('AppBundle:User')
22             ->findAll();
23         /* @var $users User[] */
24
25         $formatted = [];
26         foreach ($users as $user) {
27             $formatted[] = [
28                 'id' => $user->getId(),
29                 'firstname' => $user->getFirstname(),
30                 'lastname' => $user->getLastname(),
31                 'email' => $user->getEmail(),
32             ];
33         }
34
35         return new JsonResponse($formatted);
36     }
37 }
```

En regardant le code, nous pouvons déjà remarquer que le contrôleur *UserController* ressemble à quelques lignes près au contrôleur *PlaceController*. Vu qu'avec REST nous utilisons une interface uniforme pour interagir avec nos ressources, si l'opération que nous voulons effectuer est identique, il y a de forte chance que le code pour l'implémentation le soit aussi. Cela nous permettra donc de gagner du temps dans les développements.

En testant avec Postman :



FIGURE 4.4. – Récupération des utilisateurs avec Postman

4.2. Lire une ressource

4.2.1. Accéder à un seul lieu

4.2.1.1. Un peu de conception



Maintenant que nous savons comment accéder à un ensemble de ressource (une collection), comment faire pour récupérer un seul lieu ?

D'un point de vue sémantique HTTP, nous savons que pour lire du contenu, il faut utiliser la méthode `GET`. Le problème maintenant est de savoir comment identifier la ressource parmi toutes celles dans la collection.

Le point de départ est, en général, le nom de la collection (**places** pour notre cas). Nous devons donc trouver un moyen permettant d'identifier de manière unique un élément de cette collection. Il a une relation entre la collection et chacune de ses ressources.

Pour le cas des lieux, nous pouvons choisir l'identifiant auto-incrémenté pour désigner de manière unique un lieu. Nous pourrions dire alors que l'identifiant `1` désigne la ressource **Tour Eiffel**.

Pour la représenter dans une URL, nous avons deux choix :

- `rest-api.local/places?id=1`
- `rest-api.local/places/1`

On pourrait être tenté par la première méthode utilisant le query string `id`. Mais la [RFC 3986](#) spécifie clairement les query strings comme étant des composants qui contiennent des données *non-hiérarchiques*. Pour notre cas, il y a une relation hiérarchique claire entre une collection et une de ses ressources. Donc cette méthode est à proscrire.

Notre URL pour désigner un seul lieu sera alors `rest-api.local/places/1`. Et pour généraliser, pour accéder à un lieu, on aura `rest-api.local/places/{place_id}` où `{place_id}` désigne l'identifiant de notre lieu.

4.2.1.2. Implémentation

Mettons maintenant en œuvre un nouvel appel permettant de récupérer un lieu. Nous allons utiliser le contrôleur *PlaceController*.

II. Développement de l'API REST

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14
15     // code de getPlacesAction
16
17     /**
18      * @Route("/places/{place_id}", name="places_one")
19      * @Method({"GET"})
20      */
21     public function getPlaceAction(Request $request)
22     {
23         $place = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:Place')
25             ->find($request->get('place_id'));
26         /* @var $place Place */
27
28         $formatted = [
29             'id' => $place->getId(),
30             'name' => $place->getName(),
31             'address' => $place->getAddress(),
32         ];
33
34         return new JsonResponse($formatted);
35     }
36 }
```

Cette action est particulièrement simple et se passe de commentaires. Ce qu'il faut retenir c'est que la méthode renvoie une seule entité et pas une liste.

En testant, nous avons comme réponse :

```
1 {
2     "id": 1,
3     "name": "Tour Eiffel",
4     "address": "5 Avenue Anatole France, 75007 Paris"
5 }
```



FIGURE 4.5. – Récupération d'un lieu avec Postman

Nous pouvons rendre la configuration de la route plus stricte en utilisant l'attribut `requirements` de l'annotation `Route`. Puisque les identifiants des lieux sont des entiers, la déclaration de la route pourrait être `@Route("/places/{place_id}", requirements={"place_id" = "\d+"}, name="places_one")`.

4.2.1.3. Pratiquons avec les utilisateurs

Bis repetita, nous allons mettre en place une méthode permettant de récupérer les informations d'un seul utilisateur.

Comme pour les lieux, pour récupérer un utilisateur, il suffit de créer un nouvel appel `GET` sur l'URL `rest-api.local/users/{id}` où `{id}` désigne l'identifiant de l'utilisateur.

Pour cela, éditons le contrôleur `UserController` pour rajouter cette méthode.

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use AppBundle\Entity\User;
11
12 class UserController extends Controller
13 {
14
15     // code de getUsersAction
16
17     /**
18      * @Route("/users/{id}", name="users_one")
19      * @Method({"GET"})
20      */
21     public function getUserAction(Request $request)
22     {
23         $user = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:User')
25             ->find($request->get('id'));
26         /* @var $user User */
```

```
27
28     $formatted = [
29         'id' => $user->getId(),
30         'firstname' => $user->getFirstname(),
31         'lastname' => $user->getLastname(),
32         'email' => $user->getEmail(),
33     ];
34
35     return new JsonResponse($formatted);
36 }
37 }
```

Nous obtenons une belle réponse JSON :

```
1 {
2   "id": 1,
3   "firstname": "Ab",
4   "lastname": "Cde",
5   "email": "ab.cde@test.local"
6 }
```

A screenshot of a web browser showing a URL: <http://zestedesavoir.com/media/galleries/3183/>

FIGURE 4.6. – Récupération d'un utilisateur

4.3. Les codes de statut (status code) pour des messages plus expressifs

4.3.1. Quel code de statut utilisé?

Que se passe-t-il si nous essayons de récupérer un lieu inexistant ?

Vous remarquerez qu'avec le code actuel si le lieu recherché n'existe pas (par exemple rest-api.local/places/42), nous avons une belle erreur nous signifiant que la méthode `getId` ne peut être appelée sur l'objet `null` (`Fatal error: Call to a member function getId() on null`) et le code de statut de la réponse est une erreur `500`.



FIGURE 4.7. – Récupération d'un lieu inexistant

Ce comportement ne respecte pas la sémantique HTTP. En effet dans n'importe quel site, si vous essayez d'accéder à une page inexistante, vous recevez la fameuse erreur **404 Not Found** qui signifie que la ressource n'existe pas. Pour que notre API soit le plus RESTful possible, nous devons implémenter un comportement similaire.

Nous ne devons avoir une erreur **500** que dans le cas d'une erreur interne du serveur. Par exemple, s'il est impossible de se connecter à la base de données, il est légitime de renvoyer une erreur **500**.

De la même façon, lorsque la ressource est trouvée, nous devons renvoyer un code **200** pour signifier que tout s'est bien passé. Par chance, ce code est le code par défaut lorsqu'on utilise l'objet *JsonResponse* de Symfony. Nous avons donc déjà ce comportement en place.



FIGURE 4.8. – Cinématique de récupération des lieux avec le code de statut

4.3.2. Gérer une erreur 404

Pour notre cas, il est facile de gérer ce type d'erreurs. Nous devons juste vérifier que la réponse du repository n'est pas nulle. Au cas contraire, il faudra renvoyer une erreur **404** avec éventuellement un message détaillant le problème.

Pour un lieu, nous aurons donc :

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use Symfony\Component\HttpFoundation\Response;
11 use AppBundle\Entity\Place;
12
```

II. Développement de l'API REST

```
13 class PlaceController extends Controller
14 {
15     // ...
16
17     /**
18      * @Route("/places/{place_id}", name="places_one")
19      * @Method({"GET"})
20      */
21     public function getPlaceAction(Request $request)
22     {
23         $place = $this->get('doctrine.orm.entity_manager')
24                 ->getRepository('AppBundle:Place')
25                 ->find($request->get('place_id'));
26         /* @var $place Place */
27
28         if (empty($place)) {
29             return new JsonResponse(['message' =>
30                                     'Place not found'], Response::HTTP_NOT_FOUND);
31         }
32
33         $formatted = [
34             'id' => $place->getId(),
35             'name' => $place->getName(),
36             'address' => $place->getAddress(),
37         ];
38
39         return new JsonResponse($formatted);
40     }
41 }
```

Maintenant, une requête GET sur l'URL rest-api.local/places/42 nous renvoie une erreur 404 avec un message bien formaté en JSON. La constante `Response::HTTP_NOT_FOUND` vaut 404 et est une constante propre à Symfony.

La réponse contient un message en JSON :

```
1 {
2   "message": "Place not found"
3 }
```



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 4.9. – Récupération d'un lieu inexistant avec Postman

II. Développement de l'API REST

Pour un utilisateur, les modifications à effectuer restent identiques :

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use Symfony\Component\HttpFoundation\Response;
11 use AppBundle\Entity\User;
12
13 class UserController extends Controller
14 {
15     // ...
16
17     /**
18      * @Route("/users/{id}", name="users_one")
19      * @Method({"GET"})
20      */
21     public function getUserAction(Request $request)
22     {
23         $user = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:User')
25             ->find($request->get('id'));
26         /* @var $user User */
27
28         if (empty($user)) {
29             return new JsonResponse(['message' =>
30                 'User not found'], Response::HTTP_NOT_FOUND);
31         }
32
33         $formatted = [
34             'id' => $user->getId(),
35             'firstname' => $user->getFirstname(),
36             'lastname' => $user->getLastname(),
37             'email' => $user->getEmail(),
38         ];
39
40         return new JsonResponse($formatted);
41     }
42 }
```

Avec ces modifications, nous avons maintenant une gestion des erreurs propres et l'API respecte au mieux la sémantique HTTP.

II. Développement de l'API REST

Après cette première introduction, nous pouvons retenir qu'en REST les interactions ont lieu avec soit une collection soit une instance de celle-ci : une ressource.

Chaque opération peut alors être décrite comme étant une requête sur une URL bien identifiée avec un verbe HTTP adéquat. Le type de la réponse est décrit par un code de statut.

Voici un petit récapitulatif du mode de fonctionnement :

| Opération souhaitée | Verbe HTTP |
|---------------------|------------|
| Lecture | GET |

| Code statut | Signification |
|-------------|---|
| 200 | Tout s'est bien passé |
| 404 | La ressource demandée n'existe pas |
| 500 | Une erreur interne a eu lieu sur le serveur |

En résumé, chaque verbe est destiné à une action et la réponse est décrite en plus des données explicitées par un code de statut.

Pour concevoir une bonne API RESTful, il faut donc toujours se poser ces questions :

- Sur quelle ressource mon opération doit s'effectuer ?
- Quel verbe HTTP décrit le mieux cette opération ?
- Quelle URL permet d'identifier la ressource ?
- Et quel code de statut doit décrire la réponse ?

5. FOSRestBundle et Symfony à la rescousse

Force est de constater que le code dans nos contrôleurs est assez répétitifs. Toutes les réponses sont en JSON via l'objet *JsonResponse*, la logique de formatage de celles-ci est dupliqué et toutes les routes suivent un même modèle.

Nous avons là un schéma classique de code facilement factorisable et justement Symfony nous propose beaucoup d'outils via les composants et les bundles afin de gérer ce genre de tâches courantes et/ou répétitifs.

Nous allons donc utiliser les avantages qu'offre le framework Symfony à travers le bundle *FOSRestBundle* afin de mieux gérer les problématiques d'implémentation liées au contrainte REST et gagner ainsi en productivité.

5.1. Installation de FOSRestBundle

Comme pour tous les bundles de Symfony, la méthode la plus simple pour l'installateur est d'utiliser le gestionnaire de dépendances *Composer*. Pour les besoins du cours, nous allons installer la version ~ 2.1 (2.1.0 pour mon cas) qui apporte un support plus complet de Symfony 3. Depuis la console, il suffit de lancer la commande :

```
1 composer require friendsofsymfony/rest-bundle "^2.1"
2
3 # Réponse
4 #> ./composer.json has been updated
5 #> Loading composer repositories with package informatio
6 #> Updating dependencies (including require-dev)
7 #>   - Installing willdurand/jsonp-callback-validator (v
8 #>     Downloading: 100%
9 #>
10 #>   - Installing willdurand/negotiation (1.5.0)
11 #>     Downloading: 100%
12 #>
13 #>   - Installing friendsofsymfony/rest-bundle (2.1.0)
14 #>     Downloading: 100%
```

Ensuite, il suffit d'activer le bundle dans Symfony en éditant le fichier *AppKernel*.

II. Développement de l'API REST

```
1 # app/AppKernel.php
2 <?php
3
4 use Symfony\Component\HttpKernel\Kernel;
5 use Symfony\Component\Config\Loader\LoaderInterface;
6
7 class AppKernel extends Kernel
8 {
9     public function registerBundles()
10    {
11        $bundles = [
12            // ... D'autres bundles déjà présents
13            new
14                Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
15            new FOS\RestBundle\FOSRestBundle(),
16            new AppBundle\AppBundle(),
17        ];
18        // ...
19    }
20    // ...
21 }
```

À l'état actuel, l'installation n'est pas encore complète. Si nous lançons la commande `php bin/console debug:config fos_rest` une belle exception est affichée.

```
1 [InvalidArgumentException]
2 Neither a service called "jms_serializer.serializer" nor
3   "serializer" is available and no serializer is explicitly
   configured. You must either enable the JMSSerializerBundle,
   enable the Framework
4 Bundle serializer or configure a custom serializer.
```

En effet, pour traiter les réponses, ce bundle a besoin d'un outil de sérialisation.



La sérialisation est un processus permettant de convertir des données (une instance d'une classe, un tableau, etc.) en un format prédéfini. Pour le cas de notre API, la sérialisation est le mécanisme par lequel nos objets PHP seront transformés en un format textuel (JSON, XML, etc.).

Heureusement pour nous, l'installation standard de Symfony contient un composant de sérialisation que nous pouvons utiliser.

Par ailleurs, *FOSRestBundle* supporte le sérialiseur fourni par le bundle [JMSSerializerBundle](#) qui fournit plus de possibilités.

II. Développement de l'API REST

Mais pour nos besoins, le sérialiseur standard suffira largement. Nous allons donc l'activer en modifiant la configuration de base dans le fichier `app/config/config.yml`.

```
1 # app/config/config.yml
2 framework:
3     # ...
4     serializer:
5         enabled: true
```

Maintenant en retapant la commande `php bin/console debug:config fos_rest`, nous obtenons :

```
1 php bin/console debug:config fos_rest
2 Current configuration for extension with alias "fos_rest"
3 =====
4
5 fos_rest:
6     disable_csrf_role: null
7     access_denied_listener:
8         enabled: false
9     service: null
10    formats: { }
11    unauthorized_challenge: null
12    param_fetcher_listener:
13        enabled: false
14    ...
```

Et voilà !

Le bundle `FOSRestBundle` fournit un ensemble de fonctionnalités permettant de développer une API REST. Nous allons en explorer une bonne partie tout au long de ce cours. Mais commençons d'abord par le système de routage et de gestion des réponses.

5.2. Routage avec FOSRestBundle

Le système de [routage](#) avec ce bundle est assez complet et facile à prendre en main. Il existe un système basé sur des conventions de nommages des méthodes et un autre basé sur des annotations.

5.2.1. Routage automatique

Afin de bien voir les effets de nos modifications, nous allons d'abord afficher les routes existantes avec la commande `php bin/console debug:router`.

II. Développement de l'API REST

```
1 php bin/console debug:router
2 -----
3 Name Method Scheme Host Path
4 -----
5 _wdt ANY ANY ANY /_wdt/{token}
6 _profiler_home ANY ANY ANY /_profiler/
7 ...
8 _twig_error_test ANY ANY ANY
  /_error/{code}.{_format}
9 homepage ANY ANY ANY /
10 tests_list GET ANY ANY /tests
11 places_list GET ANY ANY /places
12 places_one GET ANY ANY
  /places/{place_id}
13 users_list GET ANY ANY /users
14 users_one GET ANY ANY
  /users/{user_id}
15 -----
```

Les routes qui nous intéressent ici sont au nombre de 4 :

- GET /places
- GET /places/{place_id}
- GET /users
- GET /users/{user_id}

FOSRestBundle nous permet d'obtenir le même résultat avec beaucoup moins de code. Nous allons donc commencer par supprimer toutes les annotations dans notre contrôleur ***PlaceController***.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\Response;
9 use AppBundle\Entity\Place;
10
11 class PlaceController extends Controller
12 {
13
14     public function getPlacesAction(Request $request)
```


II. Développement de l'API REST

```
15     {
16         $places = $this->get('doctrine.orm.entity_manager')
17             ->getRepository('AppBundle:Place')
18             ->findAll();
19         /* @var $places Place[] */
20
21         $formatted = [];
22         foreach ($places as $place) {
23             $formatted[] = [
24                 'id' => $place->getId(),
25                 'name' => $place->getName(),
26                 'address' => $place->getAddress(),
27             ];
28         }
29
30         return new JsonResponse($formatted);
31     }
32
33     public function getPlaceAction(Request $request)
34     {
35         $place = $this->get('doctrine.orm.entity_manager')
36             ->getRepository('AppBundle:Place')
37             ->find($request->get('place_id'));
38         /* @var $place Place */
39
40         if (empty($place)) {
41             return new JsonResponse(['message' =>
42                 'Place not found'], Response::HTTP_NOT_FOUND);
43         }
44
45         $formatted = [
46             'id' => $place->getId(),
47             'name' => $place->getName(),
48             'address' => $place->getAddress(),
49         ];
50
51         return new JsonResponse($formatted);
52     }
53 }
```

En relançant la commande `php bin/console debug:router`, nous voyons maintenant qu'il n'existe aucune route pour les lieux. Nous allons donc configurer Symfony pour que *FOSRestBundle* s'occupe du routage. Les routes seront directement déclarées dans `app/config/routing.yml`. *FOSRestBundle* introduit un `RouteLoader` qui supporte les routes de type `rest`. C'est donc la seule nouveauté dans la configuration des routes dans Symfony.

II. Développement de l'API REST

```
1 # app/config/routing.yml
2 app:
3     resource: "@AppBundle/Controller/DefaultController.php"
4     type:     annotation
5
6 places:
7     type:     rest
8     resource: AppBundle\Controller\PlaceController
```



Dans la clé `app`, la déclaration a été changée pour dire à Symfony de ne plus charger nos contrôleurs REST, la clé `app.resource` passe ainsi de `@AppBundle/Controller` à `@AppBundle/Controller/DefaultController.php`.

Nous pouvons constater avec la commande `php bin/console debug:router` que deux routes ont été générées pour les lieux :

- `get_places /places.{_format}`
- `get_place /place.{_format}`

Nous reviendrons plus tard sur la présence de l'attribut `_format` dans la route.

Il suffit de tester les nouvelles routes générées pour nous rendre compte que le fonctionnement de l'application reste entièrement le même.



Mais comment *FOSRestBundle* génère-t-il nos routes ?

Tout le secret réside dans des conventions de nommage. Les noms que nous avons utilisé pour le contrôleur et les actions permettent de générer des routes RESTful sans efforts de notre part.

Ainsi, le nom du contrôleur sans le suffixe *Controller* permet d'identifier le nom de notre ressource. *PlaceController* permet de désigner la ressource **places**. Il faut noter aussi que si le contrôleur s'appelait *PlacesController* (avec un « s »), la ressource serait aussi **places**. Ce nom constitue donc le début de notre URL.

Ensuite, pour le reste de l'URL et surtout le verbe HTTP, *FOSRestBundle* se base sur le nom de la méthode. La méthode *getPlacesAction* peut être vu en deux parties : *get* qui désigne le verbe HTTP à utiliser `GET`, et *Places* au pluriel qui correspond exactement au même nom que notre ressource.

Cette méthode dit donc à *FOSRestBundle* que nous voulons récupérer la collection de lieux de notre application qui le traduit en REST par *GET /places*.



Le paramètre `Request $request` est propre à Symfony et donc est ignoré par *FOSRestBundle*.

II. Développement de l'API REST

De la même façon, la méthode `getPlaceAction` (sans un ---« s » à « Place ») dit à *FOSRestBundle* que nous voulons récupérer un seul lieu.

Mais la différence ici réside dans le fait que nous avons besoin d'un paramètre pour identifier le lieu que nous voulons récupérer. Pour que la route générée soit correcte, il est obligatoire de rajouter un paramètre identifiant la ressource.

La signature de la méthode devient alors :

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 public function getPlaceAction($id, Request $request)
4     {
5         $place = $this->get('doctrine.orm.entity_manager')
6                 ->getRepository('AppBundle:Place')
7                 ->find($id); // L'identifiant est utilisé
8                             directement
9         /* @var $place Place */
10        // ...
11        return new JsonResponse($formatted);
12    }
```

Les nouvelles routes deviennent :

- `get_places GET /places.{_format}` qui permet de récupérer tous les lieux de l'application (`get_places` est le nom de la route générée) ;
- `get_place GET /places/{id}.{_format}` qui permet de récupérer un seul lieu de l'application (`get_place` est le nom de la route générée).

Nous retrouvons deux routes totalement opérationnelles. En suivant cet ensemble de normes, les routes sont alors générées automatiquement avec les bonnes URL, les bons paramètres et les bons verbes HTTP.

5.2.2. Routage manuel

Bien que très pratique, le routage automatique peut rapidement montrer ses limites. D'abord, il nous impose des règles de nommage pour nos méthodes. Si nous voulons nommer autrement nos actions dans le contrôleur, nous faisons face à une limitation vu que les URL et les verbes HTTP peuvent être impactés. Ensuite, pour avoir des routes correctes, il faudra connaître l'ensemble des règles de nommage qu'utilise *FOSRestBundle*, ce qui est loin d'être évident.

Heureusement, nous avons à disposition une méthode manuelle permettant de définir nos routes facilement.

L'avantage du routage manuel réside dans le fait qu'il se rapproche au plus du système de routage natif de Symfony avec *SensioFrameworkExtraBundle* et permet donc de moins se perdre en tant que débutant. En plus, les annotations permettant de déclarer les routes sont plus lisibles.

II. Développement de l'API REST

FOSRestBundle propose donc plusieurs annotations de routage :

- `FOS\RestBundle\Controller\Annotations\Get` ;
- `FOS\RestBundle\Controller\Annotations\Head` ;
- `FOS\RestBundle\Controller\Annotations\Put` ;
- `FOS\RestBundle\Controller\Annotations>Delete` ;
- `FOS\RestBundle\Controller\Annotations\Post` ;
- `FOS\RestBundle\Controller\Annotations\Patch` ;

Chacune de ces annotations désigne une méthode HTTP et prend exactement les mêmes paramètres que l'annotation [Route](#) `&` que nous avons déjà utilisée.

Pour l'appliquer dans le cas du contrôleur *PlaceController*, nous aurons :

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations\Get; // N'oublons pas
    d'inclure Get
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14
15     /**
16      * @Get("/places")
17      */
18     public function getPlacesAction(Request $request)
19     {
20         // ...
21     }
22
23     /**
24      * @Get("/places/{id}")
25      */
26     public function getPlaceAction(Request $request)
27     {
28         $place = $this->get('doctrine.orm.entity_manager')
29             ->getRepository('AppBundle:Place')
30             ->find($request->get('id')); // L'identifiant en
                tant que paramètre n'est plus nécessaire
31
32         // ...
33     }
34 }
```

II. Développement de l'API REST

Les nouvelles routes restent inchangées :

- `get_places` GET `/places.{_format}`
- `get_place` GET `/places/{id}.{_format}`



Si une de ces annotations est utilisée sur une action du contrôleur, le système de routage automatique abordé précédemment n'est plus utilisable sur cette même action.

5.3. Quid de l'attribut `_format` ?

Dans chacune des routes générées, nous avons un attribut `_format` qui apparaît. *FOSRestBundle* introduit automatiquement ce paramètre afin de gérer le format des réponses. Vu que pour notre cas nous forçons toujours une réponse JSON, les URL rest-api.local/places , rest-api.local/places.json , rest-api.local/places.nimportequoi correspondent toutes à la même route et renvoient du JSON.

Pour gérer plusieurs formats de réponse, HTTP propose une solution plus élégante avec l'entête `Accept` que nous aborderons plus tard. Nous allons donc désactiver l'ajout automatique de cet attribut en reconfigurant *FOSRestBundle*.

Il faut rajouter une entrée dans le fichier de configuration :

```
1 # app/config/config.yml
2
3 # ...
4
5 fos_rest:
6     routing_loader:
7         include_format: false
```

Si nous relançons `php bin/console debug:config fos_rest`, le format n'est plus présent dans les routes :

- `get_places` GET `/places`
- `get_place` GET `/places/{id}`

Pratiquons en redéfinissant les routes du contrôleur *UserController* avec les annotations de *FOSRestBundle*.

```
1 # src/AppBunble/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
```

II. Développement de l'API REST

```
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Request;
10 use Symfony\Component\HttpFoundation\Response;
11 use FOS\RestBundle\Controller\Annotations\Get;
12 use AppBundle\Entity\User;
13
14 class UserController extends Controller
15 {
16     /**
17      * @Get("/users")
18      */
19     public function getUsersAction(Request $request)
20     {
21         // ...
22     }
23
24     /**
25      * @Get("/users/{user_id}")
26      */
27     public function getUserAction(Request $request)
28     {
29         // ...
30     }
31 }
```

Et n'oublions pas de déclarer dans notre fichier de routage :

```
1 # app/config/routing.yml
2 app:
3     resource: "@AppBundle/Controller/DefaultController.php"
4     type:     annotation
5
6 places:
7     type:     rest
8     resource: AppBundle\Controller\PlaceController
9
10 users:
11     type:     rest
12     resource: AppBundle\Controller\UserController
```

Voyons maintenant les outils que ce bundle nous propose pour la gestion des vues.

5.4. Gestion des réponses avec FOSRestBundle

5.4.1. Configuration du gestionnaire de vue

Avec *FOSRestBundle*, nous disposons d'un service appelé `fos_rest.view_handler` qui nous permet de gérer nos réponses. Pour l'utiliser, il suffit d'instancier une vue *FOSRestBundle*, la configurer et laisser le gestionnaire de vue (le view handler) s'occuper du reste. Voyez par vous-même :

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations\Get;
10 use FOS\RestBundle\View\ViewHandler;
11 use FOS\RestBundle\View\View; // Utilisation de la vue de
    FOSRestBundle
12 use AppBundle\Entity\Place;
13
14 class PlaceController extends Controller
15 {
16
17     /**
18      * @Get("/places")
19      */
20     public function getPlacesAction(Request $request)
21     {
22         $places = $this->get('doctrine.orm.entity_manager')
23             ->getRepository('AppBundle:Place')
24             ->findAll();
25         /* @var $places Place[] */
26
27         $formatted = [];
28         foreach ($places as $place) {
29             $formatted[] = [
30                 'id' => $place->getId(),
31                 'name' => $place->getName(),
32                 'address' => $place->getAddress(),
33             ];
34         }
35
36         // Récupération du view handler
37         $viewHandler = $this->get('fos_rest.view_handler');
```

II. Développement de l'API REST

```
39     // Création d'une vue FOSRestBundle
40     $view = View::create($formatted);
41     $view->setFormat('json');
42
43     // Gestion de la réponse
44     return $viewHandler->handle($view);
45 }
46 }
```

L'intérêt d'utiliser un bundle réside aussi dans le fait de réduire les lignes de codes que nous avons à écrire (et par la même occasion, les sources de bogues). N'hésitez pas à retester notre appel afin de vérifier que la réponse est toujours la même.

FOSRestBundle introduit aussi un listener (ViewResponseListener) qui nous permet, à l'instar de Symfony via l'annotation `Template` du [SensioFrameworkExtraBundle](#), de renvoyer juste une instance de View et laisser le bundle appellait le gestionnaire de vue lui-même.



Pour utiliser l'annotation `View`, il faut que le `SensioFrameworkExtraBundle` soit activé. Mais si vous avez utilisé l'installateur de Symfony pour créer ce projet, c'est déjà le cas.

Nous allons donc activer le listener en modifiant notre configuration :

```
1 # app/config/config.yml
2 fos_rest:
3     routing_loader:
4         include_format: false
5     view:
6         view_response_listener: true
```

Ensuite, il ne reste plus qu'à adapter le code (toutes les annotations de FOSRestBundle seront aliasées par `Rest`) :

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
10 use FOS\RestBundle\View\ViewHandler;
```


II. Développement de l'API REST

```
11 use FOS\RestBundle\View\View; // Utilisation de la vue de
    FOSRestBundle
12 use AppBundle\Entity\Place;
13
14 class PlaceController extends Controller
15 {
16
17     /**
18      * @Rest\View()
19      * @Rest\Get("/places")
20      */
21     public function getPlacesAction(Request $request)
22     {
23         $places = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:Place')
25             ->findAll();
26         /* @var $places Place[] */
27
28         $formatted = [];
29         foreach ($places as $place) {
30             $formatted[] = [
31                 'id' => $place->getId(),
32                 'name' => $place->getName(),
33                 'address' => $place->getAddress(),
34             ];
35         }
36
37         // Création d'une vue FOSRestBundle
38         $view = View::create($formatted);
39         $view->setFormat('json');
40
41         return $view;
42     }
43 }
```

La simplicité qu'apporte ce bundle ne s'arrête pas là. Les données assignées à la vue sont sérialisées au bon format en utilisant le sérialiseur que nous avons configuré au début. Ce sérialiseur supporte aussi bien les tableaux que les objets. Si vous voulez approfondir le sujet, il est préférable de consulter [la documentation complète](#) .

Ce qu'il faut retenir dans notre cas, c'est qu'avec nos objets actuels (accesseurs en visibilité public), le sérialiseur de Symfony peut les transformer pour nous. Au lieu de passer un tableau formaté par nos soins, nous allons passer directement une liste d'objets au view handler. Notre code peut être réduit à :

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
```

II. Développement de l'API REST

```
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
10 use FOS\RestBundle\View\ViewHandler;
11 use FOS\RestBundle\View\View; // Utilisation de la vue de
    FOSRestBundle
12 use AppBundle\Entity\Place;
13
14 class PlaceController extends Controller
15 {
16
17     /**
18      * @Rest\View()
19      * @Rest\Get("/places")
20      */
21     public function getPlacesAction(Request $request)
22     {
23         $places = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:Place')
25             ->findAll();
26         /* @var $places Place[] */
27
28         // Création d'une vue FOSRestBundle
29         $view = View::create($places);
30         $view->setFormat('json');
31
32         return $view;
33     }
34 }
```

Et là, nous voyons vraiment l'intérêt d'utiliser les composants que nous propose le framework. L'objectif est d'être le plus concis et productif possible.

5.4.2. La cerise sur le gâteau : Format automatique et réponse sans l'objet View

Pour l'instant, notre API ne supporte qu'un seul format : le JSON. Donc au lieu de le mettre dans tous les contrôleurs, *FOSRestBundle* propose un mécanisme permettant de gérer les formats et la [négociation de contenu](#) [↗](#) : le [format listener](#) [↗](#).

Il y aura un chapitre dédié à la gestion de plusieurs formats et la négociation de contenu.

Pour l'instant, nous allons juste configurer le format listener de *FOSRestBundle* pour que toutes les URL renvoient du JSON.

II. Développement de l'API REST

```
1 # src/app/config/config.yml
2 fos_rest:
3   routing_loader:
4     include_format: false
5   view:
6     view_response_listener: true
7   format_listener:
8     rules:
9     - { path: '^/', priorities: ['json'], fallback_format:
        'json' }
```

La seule règle déclarée dit que pour toutes les URL (`path: ^/`), le format prioritaire est le JSON (`priorities: ['json']`) et si aucun format n'est demandé par le client, il faudra utiliser le JSON quand même (`fallback_format: 'json'`).

Vu que maintenant nous n'avons plus à définir le format dans les actions de nos contrôleurs, nous avons même la possibilité de renvoyer directement nos objets sans utiliser l'objet *View* de *FOSRestBundle*.

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14
15     /**
16      * @Rest\View()
17      * @Rest\Get("/places")
18      */
19     public function getPlacesAction(Request $request)
20     {
21         $places = $this->get('doctrine.orm.entity_manager')
22             ->getRepository('AppBundle:Place')
23             ->findAll();
24         /* @var $places Place[] */
25
26         return $places;
27     }
28 }
```

```
28 }
```

Un dernier test juste pour la forme :



http://zestedesavoir.com/media/galleries/3183/

FIGURE 5.1. – Récupération des lieux avec Postman

5.5. Pratiquons avec notre code

Maintenant que nous pouvons produire plus en écrivant moins de lignes de code, nous allons transformer toutes nos actions à l'image de *getPlacesAction*.

```
1 # src/AppBunble/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14
15     /**
16      * @Rest\View()
17      * @Rest\Get("/places")
18      */
19     public function getPlacesAction(Request $request)
20     {
21         $places = $this->get('doctrine.orm.entity_manager')
22             ->getRepository('AppBundle:Place')
23             ->findAll();
24         /* @var $places Place[] */
25
26         return $places;
27     }
28 }
```

II. Développement de l'API REST

```
29     /**
30     * @Rest\View()
31     * @Rest\Get("/places/{id}")
32     */
33     public function getPlaceAction(Request $request)
34     {
35         $place = $this->get('doctrine.orm.entity_manager')
36                 ->getRepository('AppBundle:Place')
37                 ->find($request->get('id')); // L'identifiant en
38         /* @var $place Place */
39
40         if (empty($place)) {
41             return new JsonResponse(['message' =>
42                                     'Place not found'], Response::HTTP_NOT_FOUND);
43         }
44
45         return $place;
46     }
}
```

```
1 # src/AppBunble/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\HttpFoundation\Request;
11 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
12     toutes les annotations
13 use AppBundle\Entity\User;
14
15 class UserController extends Controller
16 {
17     /**
18     * @Rest\View()
19     * @Rest\Get("/users")
20     */
21     public function getUsersAction(Request $request)
22     {
23         $users = $this->get('doctrine.orm.entity_manager')
24                 ->getRepository('AppBundle:User')
25                 ->findAll();
26         /* @var $users User[] */
}
```

II. Développement de l'API REST

```
27     return $users;
28 }
29
30 /**
31  * @Rest\View()
32  * @Rest\Get("/users/{user_id}")
33  */
34 public function getUserAction(Request $request)
35 {
36     $user = $this->get('doctrine.orm.entity_manager')
37         ->getRepository('AppBundle:User')
38         ->find($request->get('user_id'));
39     /* @var $user User */
40
41     if (empty($user)) {
42         return new JsonResponse(['message' =>
43             'User not found'], Response::HTTP_NOT_FOUND);
44     }
45
46     return $user;
47 }
```

FOSRestBundle est l'un des bundles les plus connus pour faire une API REST avec Symfony. Bien qu'ayant abordé pas mal de points dans cette partie du cours, il reste encore beaucoup de fonctionnalités à découvrir et durant ce cours une bonne partie sera présentée. Mais la référence reste la [documentation officielle](#) [↗](#) qui vous sera d'une grande aide dans vos futurs développements.

Pour le reste du cours, nous utiliserons ce bundle pour faciliter le travail et ne pas réinventer la roue. Le routage et la gestion des réponses seront calqués sur les cas que nous venons de voir.

6. Créer et supprimer des ressources

Notre API ne permet pour l'instant que la lecture de données. Une API en lecture seule étant loin d'être courante (ni amusante à développer), nous allons voir comment créer et supprimer une ressource en suivant les principes REST.

6.1. Création d'une ressource

Le schéma que nous allons adopter doit maintenant être familier. Plus tôt dans ce cours, nous avons :

Pour concevoir une bonne API RESTful, il faut donc toujours se poser ces questions :

- Sur quelle ressource mon opération doit s'effectuer ?
- Quel verbe HTTP décrit le mieux cette opération ?
- Quelle URL permet d'identifier la ressource ?
- et quel code de statut doit décrire la réponse ?

Nous allons donc suivre ce conseil, et rajouter une action permettant de créer un lieu dans notre application.

6.1.1. Quelle est la ressource cible ?



La première question que nous devons nous poser est sur quelle ressource pouvons-nous faire un appel de création ?

De point de vue sémantique, nous pouvons considérer qu'une entité dans une application est accessible en utilisant la collection (**places**) ou en utilisant directement la ressource à travers son identifiant (**places/1**). Mais comme vous vous en doutez, une ressource que nous n'avons pas encore créé ne peut pas avoir d'identifiant.

Il faut donc voir la *création* d'une ressource comme étant *l'ajout* de celle-ci dans une *collection*.

Créer un lieu revient donc à rajouter un lieu à notre liste déjà existante. Pour créer une ressource, il faudra donc utiliser la collection associée.

6.1.2. Quel verbe HTTP ?

Pour identifier notre collection, nous utiliserons l'URL `rest-api.local/places`. Mais quel appel doit-on faire ? Les verbes HTTP ont chacun une signification et une utilisation bien définie. Pour la création, la méthode `POST` est bien appropriée. Pour s'en convaincre, il suffit de consulter la [RFC 7231](#) qui dit :

For example, POST is used for the following functions (among others

- Providing a block of data, such as the fields entered into an HTML form, to a data-handling process ;
- Posting a message to a bulletin board, newsgroup, mailing list, blog, or similar group of articles ;
- **Creating a new resource that has yet to be identified by the origin server ;**

POST est utilisé pour les fonctions suivantes (entre autres) :

- ...
- ...
- **Création d'une nouvelle ressource qui n'a pas encore été identifiée par le serveur d'origine ;**

6.1.3. Le corps de notre requête

Maintenant que nous savons qu'il faudra une requête du type `POST rest-api.local/places`, nous allons nous intéresser au corps de notre requête : **le payload** (dans le jargon API).

Lorsque nous soumettons un formulaire sur une page web avec la méthode `POST`, le contenu est encodé en utilisant les encodages `application/x-www-form-urlencoded` ou encore `multipart/form-data` que vous avez sûrement déjà rencontrés.

Pour le cas d'une API, nous pouvons utiliser le format que nous voulons dans le corps de nos requêtes tant que le serveur supporte ce format. Nous allons donc choisir le JSON comme format.

i

Ce choix n'est en aucun cas lié au format de sortie de nos réponses. Le JSON reste un format textuel largement utilisé et supporté et représente souvent le minimum à supporter par une API REST. Ceci étant dit, supporter le format JSON n'est pas une contrainte REST.

6.1.4. Quel code de statut HTTP ?

Pour rappels, les codes de statut HTTP peuvent être regroupés par *famille*. Le premier chiffre permet d'identifier la famille de chaque code. Ainsi les codes de la famille `2XX` (200, 201, 204, etc.) décrivent une requête qui s'est effectué avec succès, la famille `4XX` (400, 404, etc.) pour une erreur côté client et enfin la famille `5XX` (500, etc.) pour une erreur serveur. La liste complète des codes de statut et leur signification est disponible dans la [section 6 de la RFC 7231](#). Mais

II. Développement de l'API REST

pour notre cas, une seule nous intéresse : `201 Created`. Le message associé à ce code parle de lui-même, si une ressource a été créée avec succès, nous utiliserons donc le code `201`.



FIGURE 6.1. – Cinématique de création d'un lieu

6.1.5. Créer un nouveau lieu

Mettons en pratique tout cela en donnant la possibilité aux utilisateurs de notre API de créer un lieu. Un utilisateur devra faire une requête `POST` sur l'URL `rest-api.local/places` avec comme payload :

```
1 {
2     "name": "ici un nom",
3     "address": "ici une adresse"
4 }
```



Le corps de la requête ne contient pas l'identifiant vu que nous allons le créer côté serveur.

Pour des soucis de clarté, les méthodes déjà existantes dans le contrôleur *PlaceController* ne seront pas visibles dans les extraits de code. Commençons donc par créer une route et en configurant le routage comme il faut :

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14     /**
15     * @Rest\View()
```

II. Développement de l'API REST

```
16     * @Rest\Post("/places")
17     */
18     public function postPlacesAction(Request $request)
19     {
20
21     }
22 }
```

Pour tester la méthode, nous allons tout d'abord simplement renvoyer les informations qui seront dans le payload.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14     /**
15     * @Rest\View()
16     * @Rest\Post("/places")
17     */
18     public function postPlacesAction(Request $request)
19     {
20         return [
21             'payload' => [
22                 $request->get('name'),
23                 $request->get('address')
24             ]
25         ];
26     }
27 }
```

Pour tester cette méthode, nous allons utiliser Postman.

<http://zestedesavoir.com/media/galleries/3183/>

II. Développement de l'API REST

FIGURE 6.2. – Payload pour la création d'un lieu

Il faut choisir comme contenu JSON, Postman rajoutera automatiquement l'entête `Content-Type` qu'il faut à la requête. Nous explorerons plus en détails ces entêtes plus tard dans ce cours.

A screenshot of a URL, `http://zestedesavoir.com/media/galleries/3183/`, enclosed in a rectangular box.

FIGURE 6.3. – Entête rajoutée par Postman

La réponse obtenue est :

A screenshot of a URL, `http://zestedesavoir.com/media/galleries/3183/`, enclosed in a rectangular box.

FIGURE 6.4. – Réponse temporaire pour la création d'un lieu

Nous avons maintenant un système opérationnel pour récupérer les informations pour créer notre lieu. Mais avant de continuer, un petit aparté sur *FOSRestBundle* s'impose.

6.1.5.1. Le body listener de FOSRestBundle

Il faut savoir que de base, Symfony ne peut pas peupler les paramètres de l'objet `Request` avec le payload `JSON`. Dans une application n'utilisant pas *FOSRestBundle*, il faudrait parser manuellement le contenu en faisant `json_decode($request->getContent(), true)` pour accéder au nom et à l'adresse du lieu.

Pour s'en convaincre, nous allons désactiver le body listener qui est activé par défaut.

```
1 # app/config/config.yml
2 fos_rest:
3     routing_loader:
4         include_format: false
5     view:
6         view_response_listener: true
7     format_listener:
8         rules:
9             - { path: '^/', priorities: ['json'], fallback_format:
10                 'json', prefer_extension: false }
```

II. Développement de l'API REST

```
11 body_listener:  
12     enabled: false
```

La réponse que nous obtenons est tout autre :



http://zestedesavoir.com/media/galleries/3183/

FIGURE 6.5. – Réponse sans le body listener de FOSRestBundle

En remplaçant, le code actuel par :

```
1 <?php  
2 return [  
3     'payload' => json_decode($request->getContent(), true)  
4 ];
```

Nous retrouvons la première réponse.

Là aussi *FOSRestBundle* nous facilite le travail et tout paraît transparent pour nous. Il faut juste garder en tête que ce listener existe et fait la transformation nécessaire pour nous.

Avant de continuer, nous allons le réactiver :

```
1 body_listener:  
2     enabled: true
```

6.1.5.2. Sauvegarde en base

Maintenant que nous avons les informations nécessaires pour créer un lieu, nous allons juste l'insérer en base avec Doctrine. Pour définir le bon code de statut, il suffit de mettre un paramètre `statusCode=Response::HTTP_CREATED` dans l'annotation `View`.

```
1 # src/AppBundle/Controller/PlaceController.php  
2 <?php  
3 namespace AppBundle\Controller;  
4  
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;  
6 use Symfony\Component\HttpFoundation\Request;  
7 use Symfony\Component\HttpFoundation\JsonResponse;
```

II. Développement de l'API REST

```
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
  toutes les annotations
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14     /**
15      * @Rest\View(statusCode=Response::HTTP_CREATED)
16      * @Rest\Post("/places")
17      */
18     public function postPlacesAction(Request $request)
19     {
20         $place = new Place();
21         $place->setName($request->get('name'))
22             ->setAddress($request->get('address'));
23
24         $em = $this->get('doctrine.orm.entity_manager');
25         $em->persist($place);
26         $em->flush();
27
28         return $place;
29     }
30 }
```

Ici, en renvoyant la ressource qui vient d'être créée, nous suivons la [RFC 7231](#) .

▮ The 201 response payload typically describes and links to the resource(s) created.
Pour les tester notre implémentation, nous allons utiliser :

```
1 {
2     "name": "Disneyland Paris",
3     "address": "77777 Marne-la-Vallée"
4 }
```

La réponse renvoyée avec le bon code de statut.



FIGURE 6.6. – Code de statut de la réponse



FIGURE 6.7. – Réponse de la création d'un lieu dans Postman

6.1.5.3. Validation des données

Bien que nous puissions créer avec succès un lieu, nous n'effectuons aucune validation. Dans cette partie, nous allons voir comment valider les informations en utilisant les formulaires de Symfony.

Nous allons commencer par créer un formulaire pour les lieux :

```
1 # src/AppBundle/Form/Type/PlaceType.php
2 <?php
3 namespace AppBundle\Form\Type;
4
5 use Symfony\Component\Form\AbstractType;
6 use Symfony\Component\Form\FormBuilderInterface;
7 use Symfony\Component\OptionsResolver\OptionsResolver;
8
9 class PlaceType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array
12         $options)
13     {
14         $builder->add('name');
15         $builder->add('address');
16     }
17
18     public function configureOptions(OptionsResolver $resolver)
19     {
20         $resolver->setDefaults([
21             'data_class' => 'AppBundle\Entity\Place',
22             'csrf_protection' => false
23         ]);
24     }
25 }
```

Dans une API, il faut obligatoirement désactiver la protection CSRF (Cross-Site Request Forgery). Nous n'utilisons pas de session et l'utilisateur de l'API peut appeler cette méthode sans se soucier de l'état de l'application : l'API doit rester sans état : stateless.

Nous allons maintenant rajouter des contraintes simples pour notre lieu. Le nom et l'adresse ne doivent pas être nulles et en plus, le nom doit être unique. Nous utiliserons le format YAML pour les règles de validations.

II. Développement de l'API REST

```
1 # src/AppBundle/Resources/config/validation.yml
2 AppBundle\Entity\Place:
3     constraints:
4         -
5             Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity:
6                 name
7     properties:
8         name:
9             - NotBlank: ~
10            - Type: string
11        address:
12            - NotBlank: ~
13            - Type: string
```

Jusque-là rien de nouveau avec les formulaires Symfony. Si ce code ne vous paraît pas assez clair. Il est préférable de consulter [la documentation officielle](#) avant de continuer ce cours.



Vu que nous avons une contrainte d'unicité sur le champ `name`. Il est plus logique de rajouter cela dans les annotations Doctrine.

```
1 # src/AppBundle/Entity/Place.php
2 <?php
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity()
9  * @ORM\Table(name="places",
10 *         uniqueConstraints={@ORM\UniqueConstraint(name="places_name_unique", columns={"name"})}
11 * )
12 */
13 class Place
14 {
15     // ...
16 }
```

Il ne reste plus qu'à exploiter le formulaire dans notre contrôleur.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
```

II. Développement de l'API REST

```
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15     /**
16     * @Rest\View(statusCode=Response::HTTP_CREATED)
17     * @Rest\Post("/places")
18     */
19     public function postPlacesAction(Request $request)
20     {
21         $place = new Place();
22         $form = $this->createForm(PlaceType::class, $place);
23
24         $form->submit($request->request->all()); // Validation des
            données
25
26         if ($form->isValid()) {
27             $em = $this->get('doctrine.orm.entity_manager');
28             $em->persist($place);
29             $em->flush();
30             return $place;
31         } else {
32             return $form;
33         }
34     }
35 }
```

Le format des données attendu lorsqu'on utilise la méthode `handleRequest` des formulaires Symfony est un peu différent de celui que nous utilisons pour créer un lieu. Avec `handleRequest`, nous aurions dû utiliser :

```
1 {
2     "place":
3     {
4         "name": "ici un nom",
5         "address": "ici une adresse"
6     }
7 }
```

Où l'attribut `place` est le nom de notre formulaire Symfony.

II. Développement de l'API REST

Donc pour mieux répondre aux contraintes REST, au lieu d'utiliser la méthode `handleRequest` pour soumettre le formulaire, nous avons opté pour [la soumission manuelle](#) avec `submit`. Nous adaptons Symfony à REST et pas l'inverse.

6.1.5.4. Gestion des erreurs avec FOSRestBundle

Lorsque le formulaire n'est pas valide, nous nous contentons juste de renvoyer le formulaire. Le `ViewHandler` de `FOSRestBundle` est conçu pour gérer nativement les formulaires invalides.

Non seulement, il est en mesure de formater les erreurs dans le formulaire mais en plus, il renvoie le bon code de statut lorsque les données soumises sont invalide : `400`. Le code de statut `400` permet de signaler au client de l'API que sa requête est invalide.

Pour s'en assurer, essayons de recréer un autre lieu avec les mêmes informations que le précédent.



FIGURE 6.8. – Code de statut pour un formulaire invalide

```
1 {
2   "code": 400,
3   "message": "Validation Failed",
4   "errors": {
5     "children": {
6       "name": {
7         "errors": [
8           "This value is already used."
9         ]
10      },
11     "address": []
12   }
13 }
14 }
```

Si la clé `errors` d'un attribut existe, alors il y a des erreurs de validation sur cet attribut.

6.1.6. Pratiquons avec les utilisateurs

Comme pour les lieux, nous allons créer une action permettant de rajouter un utilisateur à notre application. Nous aurons comme contraintes :

- le prénom, le nom et l'adresse mail de l'utilisateur ne doivent pas être nuls ;

II. Développement de l'API REST

— et l'adresse mail doit être unique.

Pour créer un utilisateur, un client de l'API devra envoyer une requête au format :

```
1 {
2     "firstname": "",
3     "lastname": "",
4     "email": ""
5 }
```

Comme pour les lieux, pour créer un utilisateur il faudra une requête **POST** sur l'URL `rest-api.local/users` qui désigne notre collection d'utilisateurs.

Allons-y!

Configuration du formulaire et des contraintes de validation :

```
1 # src/AppBundle/Form/Type/UserType.php
2 <?php
3 namespace AppBundle\Form\Type;
4
5 use Symfony\Component\Form\AbstractType;
6 use Symfony\Component\Form\FormBuilderInterface;
7 use Symfony\Component\OptionsResolver\OptionsResolver;
8 use Symfony\Component\Form\Extension\Core\Type\EmailType;
9
10 class UserType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array
13         $options)
14     {
15         $builder->add('firstname');
16         $builder->add('lastname');
17         $builder->add('email', EmailType::class);
18     }
19
20     public function configureOptions(OptionsResolver $resolver)
21     {
22         $resolver->setDefaults([
23             'data_class' => 'AppBundle\Entity\User',
24             'csrf_protection' => false
25         ]);
26     }
27 }
```

II. Développement de l'API REST

```
1 # src/AppBundle/Resources/config/validation.yml
2
3 # ...
4
5 AppBundle\Entity\User:
6   constraints:
7     -
8       Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity:
9         email
10
11   properties:
12     firstname:
13       - NotBlank: ~
14       - Type: string
15     lastname:
16       - NotBlank: ~
17       - Type: string
18     email:
19       - NotBlank: ~
20       - Email: ~
```

Rajout d'une contrainte d'unicité dans Doctrine :

```
1 # src/AppBundle/Entity/User.php
2 <?php
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity()
9  * @ORM\Table(name="users",
10 *         uniqueConstraints={@ORM\UniqueConstraint(name="users_email_unique", columns={"email"})}
11 * )
12 */
13 class User
14 {
15     // ...
16 }
```

Utilisation de notre formulaire dans le contrôleur :

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
```

```
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\HttpFoundation\Request;
11 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
12 use AppBundle\Form\Type\UserType;
13 use AppBundle\Entity\User;
14
15 class UserController extends Controller
16 {
17     // ...
18
19     /**
20      * @Rest\View(statusCode=Response::HTTP_CREATED)
21      * @Rest\Post("/users")
22      */
23     public function postUsersAction(Request $request)
24     {
25         $user = new User();
26         $form = $this->createForm(UserType::class, $user);
27
28         $form->submit($request->request->all());
29
30         if ($form->isValid()) {
31             $em = $this->get('doctrine.orm.entity_manager');
32             $em->persist($user);
33             $em->flush();
34             return $user;
35         } else {
36             return $form;
37         }
38     }
39 }
```

Nous pouvons maintenant créer des utilisateurs grâce à l'API.

6.2. Suppression d'une ressource

La suppression d'une ressource reste une action très facile à appréhender. Le protocole HTTP dispose d'une méthode appelée **DELETE** qui comme son nom l'indique permet de supprimer une ressource. Quant à l'URL de la ressource, il suffit de se poser une seule question :



Que voulons-nous supprimer ?

II. Développement de l'API REST

La méthode `DELETE` s'appliquera sur la ressource à supprimer. Si par exemple nous voulons supprimer le lieu avec comme identifiant 3, il suffira de faire une requête sur l'URL `rest-api.local/places/3`.

Une fois n'est pas de coutume, nous allons consulter la [RFC 7312](#) ↗

If a `DELETE` method is successfully applied, the origin server SHOULD send a 202 (Accepted) status code if the action will likely succeed but has not yet been enacted, a 204 (No Content) status code if the action has been enacted and no further information is to be supplied, or a 200 (OK) status code if the action has been enacted and the response message includes a representation describing the status.

Cette citation est bien longue mais ce qui nous intéresse ici se limite à `a 204 (No Content) status code if the action has been enacted and no further information is to be supplied`. Pour notre cas, lorsque la ressource sera supprimée, nous allons renvoyer aucune information. Le code de statut à utiliser est donc : 204.



FIGURE 6.9. – Cinématique de suppression d'une ressource

6.2.1. Suppression d'un lieu

Nous allons, sans plus attendre, créer une méthode pour supprimer un lieu de notre application.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15     // ...
16
17     /**
18      * @Rest\View(statusCode=Response::HTTP_NO_CONTENT)
```

II. Développement de l'API REST

```
19  * @Rest\Delete("/places/{id}")
20  */
21  public function removePlaceAction(Request $request)
22  {
23      $em = $this->get('doctrine.orm.entity_manager');
24      $place = $em->getRepository('AppBundle:Place')
25              ->find($request->get('id'));
26      /* @var $place Place */
27
28      $em->remove($place);
29      $em->flush();
30  }
31 }
```

Un petit test rapide avec Postman nous donne :



FIGURE 6.10. – Suppression d'un lieu avec Postman

Le code de statut est aussi correct :



FIGURE 6.11. – Code de statut pour la suppression d'une ressource



Que faire si nous essayons de supprimer une ressource qui n'existe pas ou plus ?

Si nous essayons de supprimer à nouveau la même ressource, nous obtenons une erreur interne. Mais, il se trouve que dans [les spécifications](#) de la méthode `DELETE`, il est dit que cette méthode doit être idempotente.



Une action idempotente est une action qui produit le même résultat et ce, peu importe le nombre de fois qu'elle est exécutée.

Pour suivre ces spécifications HTTP, nous allons modifier notre code pour gérer le cas où le lieu à supprimer n'existe pas ou plus. En plus, l'objectif d'un client qui fait un appel de suppression

II. Développement de l'API REST

est de supprimer une ressource, donc si elle l'est déjà, nous pouvons considérer que tout c'est bien passé.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15     /**
16      * @Rest\View(statusCode=Response::HTTP_NO_CONTENT)
17      * @Rest>Delete("/places/{id}")
18      */
19     public function removePlaceAction(Request $request)
20     {
21         $em = $this->get('doctrine.orm.entity_manager');
22         $place = $em->getRepository('AppBundle:Place')
23             ->find($request->get('id'));
24         /* @var $place Place */
25
26         if ($place) {
27             $em->remove($place);
28             $em->flush();
29         }
30     }
31 }
```

6.2.2. Pratiquons avec les utilisateurs

Rajoutons une méthode pour supprimer un utilisateur.

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
```

II. Développement de l'API REST

```
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\HttpFoundation\Request;
11 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
12 use AppBundle\Form\Type\UserType;
13 use AppBundle\Entity\User;
14
15 class UserController extends Controller
16 {
17     // ...
18
19     /**
20      * @Rest\View(statusCode=Response::HTTP_NO_CONTENT)
21      * @Rest\Delete("/users/{id}")
22      */
23     public function removeUserAction(Request $request)
24     {
25         $em = $this->get('doctrine.orm.entity_manager');
26         $user = $em->getRepository('AppBundle:User')
27             ->find($request->get('id'));
28         /* @var $user User */
29
30         if ($user) {
31             $em->remove($user);
32             $em->flush();
33         }
34     }
35 }
```

Pour revenir sur nos tableaux récapitulatifs, voici le mode de fonctionnement simplifié d'une API REST :

| Opération souhaitée | Verbe HTTP |
|---------------------|------------|
| Lecture | GET |
| Création | POST |
| Suppression | DELETE |

| Code statut | Signification |
|-------------|---|
| 200 | Tout s'est bien passé et la réponse a du contenu |
| 204 | Tout s'est bien passé mais la réponse est vide |
| 400 | Les données envoyées par le client sont invalides |

II. Développement de l'API REST

| | |
|-----|---|
| 404 | La ressource demandée n'existe pas |
| 500 | Une erreur interne a eu lieu sur le serveur |

7. Mettre à jour des ressources

Maintenant que nous pouvons lire, écrire et supprimer des ressources, il ne reste plus qu'à apprendre à les modifier et le CRUD¹ (Créer, Lire, Mettre à jour et Supprimer) en REST n'aura plus de secret pour nous.

Dans cette partie, nous aborderons les concepts liés à la mise à jour de ressources REST et nous ferons un petit détour sur la gestion des erreurs avec *FOSRestBundle*.

7.1. Mise à jour complète d'une ressource

Afin de gérer la mise à jour des ressources, nous devons différencier la mise à jour complète (le client de l'API veut changer toute la ressource) de la mise à jour partielle (le client de l'API veut changer juste quelques attributs de la ressource).

Déroulons notre schéma classique pour trouver les mécanismes qu'offre HTTP pour gérer la mise à jour complète d'une ressource.

7.1.1. Quelle est la ressource cible ?

Lorsque nous voulons modifier une ressource, la question ne se pose pas. La cible de notre requête est la ressource à mettre à jour. Donc pour mettre à jour un lieu, nous devons faire une requête sur l'URL de celle-ci (par exemple `rest-api.local/places/1`).

7.1.2. Quel verbe HTTP ?

La différenciation entre la mise à jour complète ou partielle d'une ressource se fait avec le choix du verbe HTTP utilisé. Donc le verbe est ici d'**une importance capitale**.

Notre fameuse [RFC 7231](#) décrit la méthode `PUT` comme :

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload.

La méthode `PUT` permet de créer ou de remplacer une ressource.

Le cas d'utilisation de `PUT` pour créer une ressource est très rare et nous ne l'aborderons pas. Il faut juste retenir que pour que cet verbe soit utilisé pour créer une ressource, il faudrait laisser au client de l'API le choix des URL de nos ressources. Nous l'utiliserons donc juste afin de

1. Create Read Update Delete

II. Développement de l'API REST

remplacer le contenu d'une ressource par le payload de la requête, bref pour la mettre à jour en entier.

7.1.3. Le corps de notre requête

Le corps de la requête sera donc la nouvelle valeur que nous voulons affecter à notre ressource (toujours au format JSON comme pour la création).

7.1.4. Quel code de statut HTTP ?

Dans la description même de la requête PUT, le code de statut à utiliser est explicité : 200.

A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent in a 200 (OK) response.



FIGURE 7.1. – Cinématique de la mise à jour complète

i

Juste pour rappel, comme pour la récupération d'une ressource, si le client essaye de mettre à jour une ressource inexistante, nous aurons un 404.

7.1.5. Mise à jour d'un lieu

Pour une mise à jour complète, un utilisateur devra faire une requête PUT sur l'URL `rest-api.local/places/{id}` où `{id}` représente l'identifiant du lieu avec comme payload, le même qu'à la création :

```
1 {  
2   "name": "ici un nom",  
3   "address": "ici une adresse"  
4 }
```

×

Le corps de la requête ne contient pas l'identifiant de la ressource vu qu'elle sera disponible dans l'URL.

II. Développement de l'API REST

Le routage dans notre contrôleur se rapproche beaucoup de celle pour récupérer un lieu :

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Entity\Place;
11
12 class PlaceController extends Controller
13 {
14     /**
15      * @Rest\View()
16      * @Rest\Put("/places/{id}")
17      */
18     public function putPlaceAction(Request $request)
19     {
20
21     }
22 }
```

Les règles de validation des informations sont exactement les mêmes qu'à la création d'un lieu. Nous allons donc exploiter le même formulaire Symfony. La seule différence ici réside dans le fait que nous devons d'abord récupérer une instance du lieu dans la base de données avant d'appliquer les mises à jour.

```
1 # src/AppController/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15     // ...
```

II. Développement de l'API REST

```
16
17     /**
18     * @Rest\View()
19     * @Rest\Put("/places/{id}")
20     */
21     public function updatePlaceAction(Request $request)
22     {
23         $place = $this->get('doctrine.orm.entity_manager')
24                 ->getRepository('AppBundle:Place')
25                 ->find($request->get('id')); // L'identifiant en
                tant que paramètre n'est plus nécessaire
26         /* @var $place Place */
27
28         if (empty($place)) {
29             return new JsonResponse(['message' =>
30                                     'Place not found'], Response::HTTP_NOT_FOUND);
31         }
32
33         $form = $this->createForm(PlaceType::class, $place);
34
35         $form->submit($request->request->all());
36
37         if ($form->isValid()) {
38             $em = $this->get('doctrine.orm.entity_manager');
39             // l'entité vient de la base, donc le merge n'est pas
40             nécessaire.
41             // il est utilisé juste par soucis de clarté
42             $em->merge($place);
43             $em->flush();
44             return $place;
45         } else {
46             return $form;
47         }
48     }
49 }
```

Pour tester cette méthode, nous allons utiliser Postman.



FIGURE 7.2. – Requête de mise à jour complète avec Postman

La réponse est :

```
1 {
2   "id": 2,
3   "name": "Mont-Saint-Michel",
4   "address": "Autre adresse Le Mont-Saint-Michel"
5 }
```



http://zestedesavoir.com/media/galleries/3183/

FIGURE 7.3. – Réponse à la requête de mise à jour dans Postman

7.1.6. Pratiquons avec les utilisateurs

La mise à jour complète d'un utilisateur suit exactement le même modèle que celle d'un lieu. Les contraintes de validation sont identiques à celles de la création d'un utilisateur.

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\HttpFoundation\Request;
11 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
12 use AppBundle\Form\Type\UserType;
13 use AppBundle\Entity\User;
14
15 class UserController extends Controller
16 {
17     // ...
18
19     /**
20      * @Rest\View()
21      * @Rest\Put("/users/{id}")
22      */
23     public function updateUserAction(Request $request)
24     {
25         $user = $this->get('doctrine.orm.entity_manager')
```

II. Développement de l'API REST

```
26         ->getRepository('AppBundle:User')
27         ->find($request->get('id')); // L'identifiant en
           tant que paramètre n'est plus nécessaire
28     /* @var $user User */
29
30     if (empty($user)) {
31         return new JsonResponse(['message' =>
           'User not found'], Response::HTTP_NOT_FOUND);
32     }
33
34     $form = $this->createForm(UserType::class, $user);
35
36     $form->submit($request->request->all());
37
38     if ($form->isValid()) {
39         $em = $this->get('doctrine.orm.entity_manager');
40         // l'entité vient de la base, donc le merge n'est pas
           nécessaire.
41         // il est utilisé juste par soucis de clarté
42         $em->merge($user);
43         $em->flush();
44         return $user;
45     } else {
46         return $form;
47     }
48 }
49 }
```



Que se passe-t-il si nous faisons une requête en omettant le champ `address` ?

Modifions notre requête en supprimant le champ `address` :

```
1 {
2     "name": "Autre-Mont-Saint-Michel"
3 }
```

La réponse est une belle erreur 400 :

```
1 {
2     "code": 400,
3     "message": "Validation Failed",
4     "errors": {
5         "children": {
6             "name": [],
```

II. Développement de l'API REST

```
7     "address": {
8         "errors": [
9             "This value should not be blank."
10        ]
11    }
12 }
13 }
14 }
```

Cela nous permet de bien valider les données envoyées par le client mais avec cette méthode, il est dans l'obligation de connaître tous les champs afin d'effectuer sa mise à jour.

7.2. Mise à jour partielle d'une ressource



Que faire alors si nous voulons modifier par exemple que le nom d'un lieu ?

Jusqu'à présent, nos appels API pour modifier une ressource se contentent de la remplacer par une nouvelle (en gardant l'identifiant). Mais dans une API plus complète avec des ressources avec beaucoup d'attributs, nous pouvons rapidement sentir le besoin de modifier juste quelques-uns de ces attributs.

Pour cela, la seule chose que nous devons changer dans notre API c'est le verbe HTTP utilisé.

7.2.1. À la rencontre de PATCH

Parmi toutes les méthodes que nous avons déjà pu utiliser, `PATCH` est la seule qui n'est pas spécifiée dans la RFC 7231 mais plutôt dans la [RFC 5789](#) .

Ce standard n'est pas encore validé - `PROPOSED STANDARD` (au moment où ces lignes sont écrites) - mais est déjà largement utilisé.

Cette méthode doit être utilisée pour décrire *un ensemble de changements* à appliquer à la ressource identifiée par son URI.



Comment décrire les changements à appliquer ?

Vu que nous utilisons du JSON dans nos payloads. Il existe une [RFC 6902](#) , elle aussi pas encore adoptée, qui essaie de formaliser le payload d'une requête `PATCH` utilisant du JSON.

Par exemple, dans la [section 4.3](#) , nous pouvons lire la description d'une opération consistant à remplacer un champ d'une ressource :

II. Développement de l'API REST

```
1 { "op": "replace", "path": "/a/b/c", "value": 42 }
```

Pour le cas de notre lieu, si nous voulions un correctif (patch) pour ne changer que l'adresse, il faudrait :

```
1 { "op": "replace", "path": "/address", "value":  
  "Ma nouvelle adresse" }
```

Outre le fait que cette méthode n'est pas beaucoup utilisée, sa mise en œuvre par un client est complexe et son traitement coté serveur l'est autant.

Donc par *pragmatisme*, nous n'allons pas utiliser PATCH de cette façon.

Dans notre API, une requête PATCH aura comme payload le même que celui d'une requête POST à une grande différence près : Si un attribut n'existe pas dans le corps de la requête, nous devons conserver son ancienne valeur.

Notre requête avec comme payload :

```
1 {  
2   "name": "Autre-Mont-Saint-Michel"  
3 }
```

... ne devra pas renvoyer une erreur mais juste modifier le nom de notre lieu.



FIGURE 7.4. – Cinématique de la mise à jour partielle d'une ressource

7.2.2. Mise à jour partielle d'un lieu

L'implémentation de la mise à jour partielle avec Symfony est très proche de la mise à jour complète. Il suffit de rajouter un paramètre dans la méthode `submit` (`clearMissing = false`) et le tour est joué. Comme son nom l'indique, avec `clearMissing` à `false`, Symfony conservera tous les attributs de l'entité `Place` qui ne sont pas présents dans le payload de la requête.

```
1 # src/AppBundle/Controller/PlaceController.php  
2 <?php  
3 namespace AppBundle\Controller;
```

II. Développement de l'API REST

```
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15     // ...
16
17     /**
18      * @Rest\View()
19      * @Rest\Patch("/places/{id}")
20      */
21     public function patchPlaceAction(Request $request)
22     {
23         $place = $this->get('doctrine.orm.entity_manager')
24                 ->getRepository('AppBundle:Place')
25                 ->find($request->get('id')); // L'identifiant en
                tant que paramètre n'est plus nécessaire
26         /* @var $place Place */
27
28         if (empty($place)) {
29             return new JsonResponse(['message' =>
30                                     'Place not found'], Response::HTTP_NOT_FOUND);
31         }
32
33         $form = $this->createForm(PlaceType::class, $place);
34
35         // Le paramètre false dit à Symfony de garder les valeurs
36         // dans notre
37         // entité si l'utilisateur n'en fournit pas une dans sa
38         // requête
39         $form->submit($request->request->all(), false);
40
41         if ($form->isValid()) {
42             $em = $this->get('doctrine.orm.entity_manager');
43             // l'entité vient de la base, donc le merge n'est pas
44             // nécessaire.
45             // il est utilisé juste par soucis de clarté
46             $em->merge($place);
47             $em->flush();
48             return $place;
49         } else {
50             return $form;
51         }
52     }
53 }
```

II. Développement de l'API REST

```
48     }
49 }
```



Nous avons ici un gros copier-coller de la méthode `updatePlace`, un peu de refactoring ne sera pas de mal.

```
1  # src/AppBundle/Controller/PlaceController.php
2  <?php
3  namespace AppBundle\Controller;
4
5  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6  use Symfony\Component\HttpFoundation\Request;
7  use Symfony\Component\HttpFoundation\JsonResponse;
8  use Symfony\Component\HttpFoundation\Response;
9  use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15     // ...
16
17     /**
18      * @Rest\View()
19      * @Rest\Put("/places/{id}")
20      */
21     public function updatePlaceAction(Request $request)
22     {
23         return $this->updatePlace($request, true);
24     }
25
26     /**
27      * @Rest\View()
28      * @Rest\Patch("/places/{id}")
29      */
30     public function patchPlaceAction(Request $request)
31     {
32         return $this->updatePlace($request, false);
33     }
34
35     private function updatePlace(Request $request, $clearMissing)
36     {
37         $place = $this->get('doctrine.orm.entity_manager')
38             ->getRepository('AppBundle:Place')
```

II. Développement de l'API REST

```
39         ->find($request->get('id')); // L'identifiant en
40         tant que paramètre n'est plus nécessaire
41     /* @var $place Place */
42     if (empty($place)) {
43         return new JsonResponse(['message' =>
44             'Place not found'], Response::HTTP_NOT_FOUND);
45     }
46     $form = $this->createForm(PlaceType::class, $place);
47
48     // Le paramètre false dit à Symfony de garder les valeurs
49     // entité si l'utilisateur n'en fournit pas une dans sa
50     // requête
51     $form->submit($request->request->all(), $clearMissing);
52
53     if ($form->isValid()) {
54         $em = $this->get('doctrine.orm.entity_manager');
55         $em->persist($place);
56         $em->flush();
57         return $place;
58     } else {
59         return $form;
60     }
61 }
```

En relançant notre requête, la réponse est bien celle attendue :



FIGURE 7.5. – Requête de mise à jour partielle avec Postman



FIGURE 7.6. – Réponse de la mise à jour dans Postman

7.2.3. Pratiquons avec les utilisateurs

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\HttpFoundation\Request;
11 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
12 use AppBundle\Form\Type\UserType;
13 use AppBundle\Entity\User;
14
15 class UserController extends Controller
16 {
17     // ...
18
19     /**
20      * @Rest\View()
21      * @Rest\Patch("/users/{id}")
22      */
23     public function patchUserAction(Request $request)
24     {
25         return $this->updateUser($request, false);
26     }
27
28     private function updateUser(Request $request, $clearMissing)
29     {
30         $user = $this->get('doctrine.orm.entity_manager')
31             ->getRepository('AppBundle:User')
32             ->find($request->get('id')); // L'identifiant en
    tant que paramètre n'est plus nécessaire
33         /* @var $user User */
34
35         if (empty($user)) {
36             return new JsonResponse(['message' =>
    'User not found'], Response::HTTP_NOT_FOUND);
37         }
38
39         $form = $this->createForm(UserType::class, $user);
40
41         $form->submit($request->request->all(), $clearMissing);
42
43         if ($form->isValid()) {
44             $em = $this->get('doctrine.orm.entity_manager');
```

II. Développement de l'API REST

```
45         $em->persist($user);
46         $em->flush();
47         return $user;
48     } else {
49         return $form;
50     }
51 }
52 }
```

Au fur et à mesure de nos développements, nous nous rendons compte que notre API est très uniforme, donc facile à utiliser pour un client. Mais aussi l'implémentation serveur l'est autant. Cette uniformité facilite grandement le développement d'une API RESTful et notre productivité est décuplée !

7.2.4. Gestion des erreurs avec FOSRestBundle

Jusque-là, nous utilisons un objet `JsonResponse` lorsque la ressource recherchée n'existe pas. Cela fonctionne bien mais nous n'utilisons pas `FOSRestBundle` de manière appropriée. Au lieu de renvoyer une réponse JSON, nous allons juste renvoyer une vue `FOSRestBundle` et laisser le view handler le formater en JSON. En procédant ainsi, nous pourrons plus tard exploiter toutes les fonctionnalités de ce bundle comme par exemple changer le format des réponses (par exemple renvoyer du XML) sans modifier notre code. Pour ce faire, il suffit de remplacer toutes les lignes :

```
1 return new JsonResponse(['message' => 'Place not found'],
    Response::HTTP_NOT_FOUND);
```

Par

```
1 return \FOS\RestBundle\View\View::create(['message' => 'Place not
    found'], Response::HTTP_NOT_FOUND);
```

Il faudra aussi faire de même avec le contrôleur *UserController*.

7.3. Notre application vu selon le modèle de Richardson

Pour rappel, le modèle de Richardson est un modèle qui permet d'évaluer son application selon les principes REST.



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 7.7. – Le niveau de maturité de notre application

Nous pouvons facilement affirmer que notre application est au niveau 2 vu que nous exploitons les différents verbes que propose le protocole HTTP pour interagir avec des ressources identifiées par des URIs. Nous verrons plus tard comment s'approcher du niveau 3 en exploitant d'autres bundles de Symfony à notre disposition.

Nos tableaux récapitulatifs s'étoffent encore plus et nous pouvons rajouter les opérations de mise à jour.

| Opération souhaitée | Verbe HTTP |
|--------------------------------------|------------|
| Lecture | GET |
| Création | POST |
| Suppression | DELETE |
| Modification complète (remplacement) | PUT |
| Modification partielle | PATCH |

| Code statut | Signification |
|-------------|---|
| 200 | Tout s'est bien passé et la réponse a du contenu |
| 204 | Tout s'est bien passé mais la réponse est vide |
| 400 | Les données envoyées par le client sont invalides |
| 404 | La ressource demandée n'existe pas |
| 500 | Une erreur interne a eu lieu sur le serveur |

8. Relations entre ressources

Maintenant que nous pouvons effectuer toutes les opérations CRUD (Créer, Lire, Mettre à jour et Supprimer) sur nos ressources, qu'est ce qui pourrait rester pour avoir une API pleinement fonctionnelle ?

Actuellement, nous avons une liste d'utilisateurs d'un côté et une liste de lieux d'un autre. Mais l'objectif de notre application est de pouvoir proposer à chaque utilisateur, selon ses centres d'intérêts, une idée de sortie en utilisant les différents lieux référencés.

Nous pouvons imaginer qu'un utilisateur passionné d'histoire ou d'architecture irait plutôt visiter un musée, un château, etc. Ou encore, selon le budget dont nous disposons, le tarif pour accéder à ces différents lieux pourrait être un élément important.

En résumé, nos ressources doivent avoir des relations entre elles et c'est ce que nous allons aborder dans cette partie du cours.

8.1. Hiérarchie entre ressources : la notion de sous-ressources

8.1.1. Un peu de conception

Supposons qu'un lieu ait un ou plusieurs tarifs par exemple moins de 12 ans et tout public. En termes de conception de la base de données, une relation **oneToMany** permet de gérer facilement cette situation et donc d'interagir avec les tarifs d'un lieu donné.



Comment matérialiser une telle relation avec une API qui suit les contraintes REST ?

Si nous créons une URI nommée `rest-api.local/prices`, nous pouvons effectivement accéder à nos prix comme pour les lieux ou les utilisateurs. Mais nous aurons accès à **l'ensemble des tarifs** appliqués pour **tous les lieux** de notre application.

Pour accéder aux prix d'un lieu **1**, il serait tentant de rajouter un paramètre du style `rest-api.local/prices?place_id=1` mais, la répétition étant pédagogique, nous allons regarder à nouveau le deuxième chapitre "Premières interactions avec les ressources" :

la [RFC 3986](#) spécifie clairement les query strings comme étant des composants qui contiennent des données *non-hiérarchiques*.

Nous avons une notion d'hiérarchie entre un lieu et ses tarifs et donc cette relation doit apparaître dans notre URI.

II. Développement de l'API REST

`rest-api.local/prices/1` ferait-il l'affaire ? Sûrement pas, cette URL désigne le tarif ayant comme identifiant 1.

Pour trouver la bonne URL, nous devons commencer par le possesseur dans la relation ici c'est un lieu qui a des tarifs, donc `rest-api.local/places/{id}` doit être le début de notre URL. Ensuite, il suffit de rajouter l'identifiant de la collection de prix que nous appellerons `prices`.

En définitif, `rest-api.local/places/{id}/prices` permet de désigner clairement les tarifs pour le lieu ayant comme identifiant `{id}`.



FIGURE 8.1. – Hiérarchie entre les ressources

Une fois que nous avons identifié notre ressource, tous les principes déjà abordés pour interagir avec une ressource s'appliquent.

| Opération souhaitée | Verbe HTTP | URI |
|---|------------|---|
| Récupérer tous les prix d'un lieu | GET | <code>/places{id}/prices</code> |
| Récupérer un seul prix d'un lieu | GET | <code>/places/{id}/prices/{price_id}</code> |
| Créer un nouveau prix pour un lieu | POST | <code>/places{id}/prices</code> |
| Suppression d'un prix pour un lieu | DELETE | <code>/places/{id}/prices/{price_id}</code> |
| Mise à jour complète d'un prix d'un lieu | PUT | <code>/places/{id}/prices/{price_id}</code> |
| Mise à jour partielle d'un prix d'un lieu | PATCH | <code>/places/{id}/prices/{price_id}</code> |

8.1.2. Pratiquons avec les lieux

Pour mettre en pratique toutes ces informations, nous allons ajouter deux nouveaux appels à notre API :

- un pour créer un nouveau prix ;
- un pour lister tous les prix d'un lieu.

Nous considérons qu'un prix a deux caractéristiques :

- un type (tout public, moins de 12 ans, etc.) ;
- une valeur (10, 15.5, 22.75, 29.99, etc.) qui désigne le tarif en euros.

Pour l'instant, seul deux types de prix sont supportés :

II. Développement de l'API REST

- *less_than_12* pour moins de 12 ans ;
- *for_all* pour tout public.

Commençons par la base de données en créant une nouvelle entité **Price**, nous rajouterons une contrainte d'unicité sur le type et le lieu afin de nous assurer qu'un lieu ne pourra pas avoir deux prix du même type :

```
1 # src/AppBundle/Entity/Price.php
2 <?php
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity()
9  * @ORM\Table(name="prices",
10 *         uniqueConstraints={@ORM\UniqueConstraint(name="prices_type_place_unique
11 * )
12 */
13 class Price
14 {
15     /**
16     * @ORM\Id
17     * @ORM\Column(type="integer")
18     * @ORM\GeneratedValue
19     */
20     protected $id;
21
22     /**
23     * @ORM\Column(type="string")
24     */
25     protected $type;
26
27     /**
28     * @ORM\Column(type="float")
29     */
30     protected $value;
31
32     /**
33     * @ORM\ManyToOne(targetEntity="Place", inversedBy="prices")
34     * @var Place
35     */
36     protected $place;
37
38     // tous les getters et setters
39 }
```

Nous utilisons une relation bidirectionnelle car nous voulons afficher les prix d'un lieu en plus des informations de base lorsqu'un client de l'API consulte les informations de ce lieu.

II. Développement de l'API REST

```
1 # src/AppBundle/Entity/Place.php
2 <?php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7 use Doctrine\Common\Collections\ArrayCollection;
8
9 /**
10  * @ORM\Entity()
11  * @ORM\Table(name="places",
12  *           uniqueConstraints={@ORM\UniqueConstraint(name="places_name_unique",colu
13  * )
14  */
15 class Place
16 {
17     /**
18      * @ORM\Id
19      * @ORM\Column(type="integer")
20      * @ORM\GeneratedValue
21      */
22     protected $id;
23
24     /**
25      * @ORM\Column(type="string")
26      */
27     protected $name;
28
29     /**
30      * @ORM\Column(type="string")
31      */
32     protected $address;
33
34     /**
35      * @ORM\OneToMany(targetEntity="Price", mappedBy="place")
36      * @var Price[]
37      */
38     protected $prices;
39
40     public function __construct()
41     {
42         $this->prices = new ArrayCollection();
43     }
44     // tous les getters et setters
45 }
```

N'oublions pas de mettre à jour la base de données avec :

II. Développement de l'API REST

```
1 php bin/console doctrine:schema:update --dump-sql --force
```

La création d'un prix nécessite quelques règles de validation que nous devons implémenter.

```
1 # src/AppBundle/Resources/config/validation.yml
2
3 # ...
4
5 AppBundle\Entity\Price:
6   properties:
7     type:
8       - NotNull: ~
9       - Choice:
10         choices: [less_than_12, for_all]
11   value:
12     - NotNull: ~
13     - Type: numeric
14     - GreaterThanOrEqual:
15       value: 0
```

Il ne reste plus qu'à créer le formulaire associé :

```
1 # src/AppBundle/Form/Type/PriceType.php
2 <?php
3 namespace AppBundle\Form\Type;
4
5 use Symfony\Component\Form\AbstractType;
6 use Symfony\Component\Form\FormBuilderInterface;
7 use Symfony\Component\OptionsResolver\OptionsResolver;
8
9 class PriceType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array
12         $options)
13     {
14         // Pas besoin de rajouter les options avec ChoiceType vu
15         // que nous allons l'utiliser via API.
16         // Le formulaire ne sera jamais affiché
17         $builder->add('type');
18         $builder->add('value');
19     }
20
21     public function configureOptions(OptionsResolver $resolver)
22     {
23         $resolver->setDefaults([
```

II. Développement de l'API REST

```
22         'data_class' => 'AppBundle\Entity\Price',
23         'csrf_protection' => false
24     ]);
25 }
26 }
```

Les deux appels seront mis en place dans un nouveau contrôleur pour des raisons de clarté. Mais il est parfaitement possible de le mettre dans le contrôleur déjà existant. Nous aurons un nouveau dossier nommé `src/AppBundle/Controller/Place` qui contiendra un contrôleur `PriceController`.

Avec ce découpage des fichiers, nous mettons en évidence la relation hiérarchique entre *Place* et *Price*.

```
1 # src/AppBundle/Controller/Place/PriceController.php
2 <?php
3 namespace AppBundle\Controller\Place;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10
11 class PriceController extends Controller
12 {
13
14     /**
15      * @Rest\View()
16      * @Rest\Get("/places/{id}/prices")
17      */
18     public function getPricesAction(Request $request)
19     {
20
21     }
22
23     /**
24      * @Rest\View(statusCode=Response::HTTP_CREATED)
25      * @Rest\Post("/places/{id}/prices")
26      */
27     public function postPricesAction(Request $request)
28     {
29
30     }
31 }
```

II. Développement de l'API REST

```
1 # app/config/routing.yml
2 app:
3     resource: "@AppBundle/Controller/DefaultController.php"
4     type:     annotation
5
6 places:
7     type:     rest
8     resource: AppBundle\Controller\PlaceController
9
10 prices:
11     type:     rest
12     resource: AppBundle\Controller\Place\PriceController
13
14 users:
15     type:     rest
16     resource: AppBundle\Controller\UserController
```

Au niveau des URL utilisées dans le routage, il suffit de se référer au tableau plus haut. Finissons notre implémentation en ajoutant de la logique aux actions du contrôleur :

```
1 # src/AppBundle/Controller/Place/PriceController.php
2 <?php
3 namespace AppBundle\Controller\Place;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PriceType;
11 use AppBundle\Entity\Price;
12
13 class PriceController extends Controller
14 {
15
16     /**
17      * @Rest\View()
18      * @Rest\Get("/places/{id}/prices")
19      */
20     public function getPricesAction(Request $request)
21     {
22         $place = $this->get('doctrine.orm.entity_manager')
23             ->getRepository('AppBundle:Place')
24             ->find($request->get('id')); // L'identifiant en
   tant que paramètre n'est plus nécessaire
25         /* @var $place Place */
```

II. Développement de l'API REST

```
26
27     if (empty($place)) {
28         return $this->placeNotFound();
29     }
30
31     return $place->getPrices();
32 }
33
34
35 /**
36  * @Rest\View(statusCode=Response::HTTP_CREATED)
37  * @Rest\Post("/places/{id}/prices")
38  */
39 public function postPricesAction(Request $request)
40 {
41     $place = $this->get('doctrine.orm.entity_manager')
42         ->getRepository('AppBundle:Place')
43         ->find($request->get('id'));
44     /* @var $place Place */
45
46     if (empty($place)) {
47         return $this->placeNotFound();
48     }
49
50     $price = new Price();
51     $price->setPlace($place); // Ici, le lieu est associé au
52     $form = $this->createForm(PriceType::class, $price);
53
54     // Le paramètre false dit à Symfony de garder les valeurs
55     // entité si l'utilisateur n'en fournit pas une dans sa
56     // requête
57     $form->submit($request->request->all());
58
59     if ($form->isValid()) {
60         $em = $this->get('doctrine.orm.entity_manager');
61         $em->persist($price);
62         $em->flush();
63         return $price;
64     } else {
65         return $form;
66     }
67 }
68
69 private function placeNotFound()
70 {
71     return \FOS\RestBundle\View\View::create(['message' =>
72         'Place not found'], Response::HTTP_NOT_FOUND);
73 }
```

II. Développement de l'API REST

```
72 }
```

Le principe reste le même qu'avec les différentes actions que nous avons déjà implémentées. Il faut juste noter que lorsque nous créons un prix, nous pouvons lui associer un lieu en récupérant l'identifiant du lieu qui est dans l'URL de la requête.

Pour tester nos nouveaux appels, nous allons créer un nouveau prix pour le lieu. Voici le payload JSON utilisé :

```
1 {
2     "type": "less_than_12",
3     "value": 5.75
4 }
```

Requête :



```
http://zestedesavoir.com/media/galleries/3183/
```

FIGURE 8.2. – Corps de la requête de création d'un lieu avec Postman

Réponse :

```
1 {
2     "error": {
3         "code": 500,
4         "message": "Internal Server Error",
5         "exception": [
6             {
7                 "message":
8                     "A circular reference has been detected (configured limit: 1).",
9                 "class":
10                    "Symfony\\Component\\Serializer\\Exception\\CircularReferenceExcepti
11            ]
12        }
13    }
```



Nous obtenons une belle erreur interne ! Pourquoi une exception avec comme message `A circular reference has been detected (configured limit: 1).` ?

8.2. Les groupes avec le sérialiseur de Symfony

Nous venons de faire face à une erreur assez commune lorsque nous travaillons avec un sérialiseur sur des entités avec des relations.

Le problème que nous avons ici est simple à expliquer. Lorsque le sérialiseur affiche un prix, il doit sérialiser le type, la valeur mais aussi le lieu associé.

Nous aurons donc :

```
1 {
2     "id": 1,
3     "type": "less_than_12",
4     "value": 5.75,
5     "place": {
6         "...": "..."
7     }
8 }
```

Les choses se gâtent lorsque le sérialiseur va essayer de transformer le lieu contenu dans notre objet. Ce lieu contient lui-même l'objet prix qui devra être sérialisé à nouveau. Et la boucle se répète à l'infini.



FIGURE 8.3. – Référence circulaire

Pour prévenir ce genre de problème, le sérialiseur de Symfony utilise [la notion de groupe](#) [↗](#). L'objectif des groupes est de définir les attributs qui seront sérialisés selon la vue que nous voulons afficher.

Reprenons le cas de notre prix pour mieux comprendre. Lorsque nous affichons les informations sur un prix, ce qui nous intéresse c'est :

- son identifiant ;
- son type ;
- sa valeur ;
- et son lieu associé.

Jusque-là notre problème reste entier. Mais lorsque nous allons afficher ce fameux lieu, nous devons limiter les informations affichées. Ainsi, nous pouvons décider que le lieu embarqué dans la réponse ne doit contenir que :

- son identifiant ;
- son nom ;

II. Développement de l'API REST

— et son adresse.

Le champ `prices` doit être ignoré.

Tous ces attributs peuvent représenter un groupe : `price`. À chaque fois que le sérialiseur est utilisé en spécifiant le groupe `price` alors seul ces attributs seront sérialisés.

De la même façon, lorsque nous voudrons afficher un lieu, tous les attributs seront affichés en excluant un seul attribut : le champ `place` de l'objet `Price`.

La configuration Symfony pour obtenir un tel comportement est assez simple :

```
1 # src/AppBundle/Resources/config/serialization.yml
2 AppBundle\Entity\Place:
3   attributes:
4     id:
5       groups: ['place', 'price']
6     name:
7       groups: ['place', 'price']
8     address:
9       groups: ['place', 'price']
10    prices:
11      groups: ['place']
12
13
14 AppBundle\Entity\Price:
15   attributes:
16     id:
17       groups: ['place', 'price']
18     type:
19       groups: ['place', 'price']
20     value:
21       groups: ['place', 'price']
22     place:
23       groups: ['price']
```

Ce fichier de configuration définit deux choses essentielles :

- Si nous utilisons le groupe `price` avec le sérialiseur, seuls les attributs dans ce groupe seront affichés ;
- et de la même façon, seuls les attributs dans le groupe `place` seront affichés si celui-ci est utilisé avec notre sérialiseur.

i

Il est aussi possible de déclarer les règles de sérialisations avec des annotations sur nos entités. Pour en savoir plus, il est préférable de consulter la [documentation officielle](#) [↗](#). Les fichiers de configuration peuvent aussi être placés dans un dossier `src/AppBundle/Resources/config/serialization/` afin de mieux les isoler.

II. Développement de l'API REST

Pour l'utiliser dans notre contrôleur avec *FOSRestBundle*, la modification à faire est très simple. Il suffit d'utiliser l'attribut `serializerGroups` de l'annotation `View`.

```
1 # src/AppBundle/Controller/Place/PriceController.php
2 <?php
3 namespace AppBundle\Controller\Place;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PriceType;
11 use AppBundle\Entity\Price;
12
13 class PriceController extends Controller
14 {
15
16     /**
17      * @Rest\View(serializerGroups={"price"})
18      * @Rest\Get("/places/{id}/prices")
19      */
20     public function getPricesAction(Request $request)
21     {
22         // ...
23     }
24
25
26     /**
27      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"price"})
28      * @Rest\Post("/places/{id}/prices")
29      */
30     public function postPricesAction(Request $request)
31     {
32         // ...
33     }
34
35     private function placeNotFound()
36     {
37         return \FOS\RestBundle\View\View::create(['message' =>
38             'Place not found'], Response::HTTP_NOT_FOUND);
39     }
39 }
```

Pour tester cette configuration, récupérons la liste des prix du lieu **1**.

II. Développement de l'API REST



http://zestedesavoir.com/media/galleries/3183/

FIGURE 8.4. – Requête Postman pour récupérer les prix d'un lieu

La réponse ne contient que les attributs que nous avons affectés au groupe `price`.

```
1 [
2   {
3     "id": 1,
4     "type": "less_than_12",
5     "value": 5.75,
6     "place": {
7       "id": 1,
8       "name": "Tour Eiffel",
9       "address": "5 Avenue Anatole France, 75007 Paris"
10    }
11  }
12 ]
```

De la même façon, nous devons modifier le contrôleur des lieux pour définir le ou les groupes à utiliser pour la sérialisation des réponses.

```
1 # src/AppBundle/Controller/PlaceController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\Request;
7 use Symfony\Component\HttpFoundation\JsonResponse;
8 use Symfony\Component\HttpFoundation\Response;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\PlaceType;
11 use AppBundle\Entity\Place;
12
13 class PlaceController extends Controller
14 {
15
16     /**
17      * @Rest\View(serializerGroups={"place"})
18      * @Rest\Get("/places")
19      */
20     public function getPlacesAction(Request $request)
```

II. Développement de l'API REST

```
21     {
22         // ...
23     }
24
25     /**
26     * @Rest\View(serializerGroups={"place"})
27     * @Rest\Get("/places/{id}")
28     */
29     public function getPlaceAction(Request $request)
30     {
31         // ...
32     }
33
34     /**
35     * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"place"})
36     * @Rest\Post("/places")
37     */
38     public function postPlacesAction(Request $request)
39     {
40         // ...
41     }
42
43     /**
44     * @Rest\View(statusCode=Response::HTTP_NO_CONTENT, serializerGroups={"place"})
45     * @Rest\Delete("/places/{id}")
46     */
47     public function removePlaceAction(Request $request)
48     {
49         // ...
50     }
51
52     /**
53     * @Rest\View(serializerGroups={"place"})
54     * @Rest\Put("/places/{id}")
55     */
56     public function updatePlaceAction(Request $request)
57     {
58         // ...
59     }
60
61     /**
62     * @Rest\View(serializerGroups={"place"})
63     * @Rest\Patch("/places/{id}")
64     */
65     public function patchPlaceAction(Request $request)
66     {
67         // ...
68     }
69
70     // ...
```

II. Développement de l'API REST

```
71 }
```

Et récupérons le lieu 1 pour voir la réponse :



http://zestedesavoir.com/media/galleries/3183/

FIGURE 8.5. – Requête Postman pour récupérer un lieu

```
1 {
2   "id": 1,
3   "name": "Tour Eiffel",
4   "address": "5 Avenue Anatole France, 75007 Paris",
5   "prices": [
6     {
7       "id": 1,
8       "type": "less_than_12",
9       "value": 5.75
10    }
11  ]
12 }
```

Grâce aux groupes, les références circulaires ne sont plus qu'un mauvais souvenir.



Les groupes du sérialiseur de Symfony ne sont supportés que depuis la version 2.0 de *FOSRestBundle*. Dans le cas où vous utilisez une version de *FOSRestBundle* inférieure à la 2.0, il faudra alors utiliser le [JMSSerializerBundle](#) à la place du sérialiseur de base de Symfony.

8.3. Mise à jour de la suppression d'une ressource

Avec la gestion des relations entre ressources, la méthode de suppression des lieux doit être revue. En effet, vu qu'un lieu peut avoir des prix, nous devons nous assurer qu'à sa suppression tous les prix qui lui sont associés le seront aussi. Sans cela, la clé étrangère empêcherait toute suppression d'un lieu ayant des prix.

La modification à effectuer reste cependant assez minime.

II. Développement de l'API REST

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5
6 // ...
7
8 class PlaceController extends Controller
9 {
10 // ...
11
12 /**
13  * @Rest\View(statusCode=Response::HTTP_NO_CONTENT, serializerGroups={"place"})
14  * @Rest\Delete("/places/{id}")
15  */
16 public function removePlaceAction(Request $request)
17 {
18     $em = $this->get('doctrine.orm.entity_manager');
19     $place = $em->getRepository('AppBundle:Place')
20         ->find($request->get('id'));
21     /* @var $place Place */
22
23     if (!$place) {
24         return;
25     }
26
27     foreach ($place->getPrices() as $price) {
28         $em->remove($price);
29     }
30     $em->remove($place);
31     $em->flush();
32 }
33 // ...
34 }
```

Avec ce chapitre, nous venons de faire un tour complet des concepts de base pour développer une API RESTful. Les possibilités d'évolution de notre API sont nombreuses et ne dépendent que de notre imagination.

Maintenant que les sous ressources n'ont plus de secrets pour nous, nous allons implémenter la fonctionnalité de base de notre API : Proposer une idée de sortie à un utilisateur.

9. TP : Le clou du spectacle - Proposer des suggestions aux utilisateurs

Nous allons dans cette partie finaliser notre API en rajoutant un système de suggestion pour les utilisateurs. Tous les concepts de base du style d'architecture qu'est REST ont déjà été abordés. L'objectif est donc de mettre en pratique les connaissances acquises.

9.1. Énoncé

Afin de gérer les suggestions, nous partons sur un design simple. Dans l'application, nous aurons une notion de préférences et de thèmes. Chaque utilisateur pourra choisir un ou plusieurs préférences avec une note sur 10. Et de la même façon, un lieu sera lié à un ou plusieurs thèmes avec une note sur 10.

Un lieu sera suggéré à un utilisateur si au moins une préférence de l'utilisateur correspond à un des thèmes du lieu et que le niveau de correspondance est supérieur ou égale à 25.

i

Le niveau de correspondance est une valeur calculée qui nous permettra de quantifier à quel point **un lieu pourrait intéresser un utilisateur**. La méthode de calcul est détaillée ci-dessous.

Pour un utilisateur donné, il faut d'abord prendre toutes ses préférences. Ensuite **pour chaque lieu** enregistré dans l'application, si une des préférences de l'utilisateur correspond au thème du lieu, il faudra calculer le produit : **valeur de la préférence de l'utilisateur * valeur du thème du lieu**.

La somme de tous ces produits représente le *niveau de correspondance* pour ce lieu .

Un exemple vaut mieux que mille discours : Un utilisateur a comme préférences (art, 5), (history, 8) et (architecture, 2). Un lieu 1 a comme thèmes (architecture, 3), (sport, 2), (history, 3). et un lieu 2 a comme thèmes (art, 3), (science-fiction, 2).

Pour le lieu 1, nous avons 2 thèmes qui correspondent à ses préférences : history et architecture.

| | history | architecture |
|-------------|---------|--------------|
| utilisateur | 8 | 2 |
| lieu 1 | 4 | 3 |

II. Développement de l'API REST

La valeur de correspondance est donc :

$$8 * 4 + 2 * 3 = 32 + 6 = 38$$

38 est supérieur à 25 donc c'est une suggestion valide.

Pour le lieu 2, nous avons un seul thème qui correspond : art.

| | |
|-------------|------------|
| | art |
| utilisateur | 5 |
| lieu 2 | 3 |

La valeur de correspondance est donc :

$$5 * 3 = 15$$

15 étant inférieur à 25 donc ce n'est pas une suggestion valide.

9.2. Détails de l'implémentation

Comme pour la liste des types de tarifs, nous disposons d'une liste de préférences et de thèmes prédéfinis :

- Art (art) ;
- Architecture (architecture) ;
- Histoire (history) ;
- Sport (sport) ;
- Science-fiction (science-fiction).

Une préférence associée à un utilisateur doit avoir 3 attributs :

- id : représente l'identifiant unique de la préférence utilisateur (auto-incrémenté) ;
- name : une des valeurs parmi la liste des préférences prédéfinies ;
- value : un entier désignant le niveau de préférence sur 10.

Un thème lié à un lieu doit avoir 3 attributs :

- id : représente l'identifiant unique du thème (auto-incrémenté) ;
- name : une des valeurs parmi la liste des thèmes prédéfinies ;
- value : un entier désignant le niveau du thème sur 10.

II. Développement de l'API REST

Une préférence associée à un utilisateur doit avoir une relation bidirectionnelle avec cet utilisateur et idem pour les lieux.

Une même préférence ne peut pas être associée deux fois à un même utilisateur ou un même lieu. (ex : un utilisateur ne peut pas avoir 2 fois la préférence art) et idem pour les lieux.

Il faudra 2 tables (donc 2 entités distinctes) :

- preferences (entité **Preference**) pour stocker les préférences utilisateurs ;
- themes (entité **Theme**) pour stocker les thèmes sur les lieux.

Il faudra 3 appels API :

- un permettant d'ajouter une préférence pour un utilisateur avec sa valeur ;
- un permettant d'ajouter un thème à un lieu avec sa valeur ;
- un pour récupérer les suggestions d'un utilisateur.

i

Une ressource REST n'est pas forcément une entité brute de notre modèle de données. Nous pouvons utiliser un appel GET sur l'URL `rest-api.local/users/1/suggestions` pour récupérer la liste des suggestions pour l'utilisateur 1.

Une fois que les préférences et les thèmes seront rajoutés, les appels de listing des utilisateurs et des lieux doivent remonter respectivement les informations sur les préférences et les informations sur les thèmes. Il faudra donc penser à gérer les références circulaires.

À vous de jouer !

9.3. Travail préparatoire

9.3.1. Gestion des thèmes pour les lieux

Nous allons commencer notre implémentation en mettant en place la gestion des thèmes.

L'entité contiendra les champs cités plus haut avec en plus une contrainte d'unicité sur le nom d'un thème et l'identifiant du lieu.

```
1 <?php
2 # src/AppBundle/Entity/Theme.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity()
10 * @ORM\Table(name="themes",
11 *           uniqueConstraints={@ORM\UniqueConstraint(name="themes_name_place_unique
12 * )
```

II. Développement de l'API REST

```
13  */
14  class Theme
15  {
16      /**
17       * @ORM\Id
18       * @ORM\Column(type="integer")
19       * @ORM\GeneratedValue
20       */
21      protected $id;
22
23      /**
24       * @ORM\Column(type="string")
25       */
26      protected $name;
27
28      /**
29       * @ORM\Column(type="integer")
30       */
31      protected $value;
32
33      /**
34       * @ORM\ManyToOne(targetEntity="Place", mappedBy="themes")
35       * @var Place
36       */
37      protected $place;
38
39      public function getId()
40      {
41          return $this->id;
42      }
43
44      public function setId($id)
45      {
46          $this->id = $id;
47      }
48
49      public function getName()
50      {
51          return $this->name;
52      }
53
54      public function setName($name)
55      {
56          $this->name = $name;
57      }
58
59      public function getValue()
60      {
61          return $this->value;
62      }
```

II. Développement de l'API REST

```
63
64     public function setValue($value)
65     {
66         $this->value = $value;
67     }
68
69     public function getPlace()
70     {
71         return $this->place;
72     }
73
74     public function setPlace(Place $place)
75     {
76         $this->place = $place;
77     }
78 }
```

L'entité `Place` doit aussi être modifiée pour avoir une relation bidirectionnelle.

```
1 <?php
2 # src/AppBundle/Entity/Place.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7 use Doctrine\Common\Collections\ArrayCollection;
8
9 /**
10  * @ORM\Entity()
11  * @ORM\Table(name="places",
12  *           uniqueConstraints={@ORM\UniqueConstraint(name="places_name_unique",colu
13  * )
14  */
15 class Place
16 {
17     // ...
18
19     /**
20      * @ORM\OneToMany(targetEntity="Theme", mappedBy="place")
21      * @var Theme[]
22      */
23     protected $themes;
24
25     public function __construct()
26     {
27         $this->prices = new ArrayCollection();
28         $this->themes = new ArrayCollection();
29     }
}
```

II. Développement de l'API REST

```
30     // ...
31
32     public function getThemes()
33     {
34         return $this->themes;
35     }
36
37     public function setThemes($themes)
38     {
39         $this->themes = $themes;
40     }
41 }
```

Pour supporter la création de thèmes pour les lieux, nous allons créer un formulaire Symfony et les règles de validation associées.

```
1 <?php
2 # src/AppBundle/Form/Type/ThemeType.php
3
4 namespace AppBundle\Form\Type;
5
6 use Symfony\Component\Form\AbstractType;
7 use Symfony\Component\Form\FormBuilderInterface;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9
10 class ThemeType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array
13         $options)
14     {
15         $builder->add('name');
16         $builder->add('value');
17     }
18
19     public function configureOptions(OptionsResolver $resolver)
20     {
21         $resolver->setDefaults([
22             'data_class' => 'AppBundle\Entity\Theme',
23             'csrf_protection' => false
24         ]);
25     }
26 }
```



La liste des thèmes prédéfinis est utilisée pour valider le formulaire Symfony.

II. Développement de l'API REST

```
1 # src/AppBundle/Resources/config/validation.yml
2
3 AppBundle\Entity\Theme:
4   properties:
5     name:
6       - NotNull: ~
7       - Choice:
8         choices: [art, architecture, history,
9                 science-fiction, sport]
10
11     value:
12       - NotNull: ~
13       - Type: numeric
14       - GreaterThan:
15         value: 0
16       - LessThanOrEqual:
17         value: 10
```

Pour ajouter un thème, nous allons créer un nouveau contrôleur qui ressemble à quelques lignes près à ce que nous avons déjà fait jusqu'ici. Nous allons en profiter pour ajouter une méthode pour lister les thèmes d'un lieu donné.

```
1 <?php
2 # src/AppBundle/Controller/Place/ThemeController.php
3
4 namespace AppBundle\Controller\Place;
5
6 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
11   toutes les annotations
12 use AppBundle\Form\Type\ThemeType;
13 use AppBundle\Entity\Theme;
14
15 class ThemeController extends Controller
16 {
17     /**
18      * @Rest\View(serializerGroups={"theme"})
19      * @Rest\Get("/places/{id}/themes")
20      */
21     public function getThemesAction(Request $request)
22     {
23         $place = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:Place')
25             ->find($request->get('id'));
```

II. Développement de l'API REST

```
26     /* @var $place Place */
27
28     if (empty($place)) {
29         return $this->placeNotFound();
30     }
31
32     return $place->getThemes();
33 }
34
35
36 /**
37  * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"theme"})
38  * @Rest\Post("/places/{id}/themes")
39  */
40 public function postThemesAction(Request $request)
41 {
42     $place = $this->get('doctrine.orm.entity_manager')
43         ->getRepository('AppBundle:Place')
44         ->find($request->get('id'));
45     /* @var $place Place */
46
47     if (empty($place)) {
48         return $this->placeNotFound();
49     }
50
51     $theme = new Theme();
52     $theme->setPlace($place);
53     $form = $this->createForm(ThemeType::class, $theme);
54
55     $form->submit($request->request->all());
56
57     if ($form->isValid()) {
58         $em = $this->get('doctrine.orm.entity_manager');
59         $em->persist($theme);
60         $em->flush();
61         return $theme;
62     } else {
63         return $form;
64     }
65 }
66
67 private function placeNotFound()
68 {
69     return \FOS\RestBundle\View\View::create(['message' =>
70         'Place not found'], Response::HTTP_NOT_FOUND);
71 }
```

Le fichier de routage de l'application doit être modifié en conséquence pour charger ce nouveau contrôleur.

II. Développement de l'API REST

```
1 # app/config/routing.yml
2
3 # ...
4
5 themes:
6     type:      rest
7     resource: AppBundle\Controller\Place\ThemeController
8
9 # ...
```

Il ne faut pas oublier de rajouter un nouveau groupe de sérialisation pour la gestion de ces thèmes.

```
1 # src/AppBundle/Resources/config/serialization.yml
2 AppBundle\Entity\Place:
3     attributes:
4         id:
5             groups: ['place', 'price', 'theme']
6         name:
7             groups: ['place', 'price', 'theme']
8         address:
9             groups: ['place', 'price', 'theme']
10        prices:
11            groups: ['place']
12        themes:
13            groups: ['place']
14
15 # ...
16
17 AppBundle\Entity\Theme:
18     attributes:
19         id:
20             groups: ['place', 'theme']
21         name:
22             groups: ['place', 'theme']
23         value:
24             groups: ['place', 'theme']
25         place:
26             groups: ['theme']
```



Le nouveau groupe est aussi utilisé pour configurer la sérialisation de l'entité `Place` afin d'éviter les références circulaires.

9.3.2. Gestions des préférences

Pour la gestion des utilisateurs, nous allons suivre exactement le même schéma d'implémentation. Les extraits de code fournis se passeront donc de commentaires.

Commençons par l'entité pour la gestion des préférences et le formulaire permettant de le gérer.

```
1 <?php
2 # src/AppBundle/Entity/Preference.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity()
10  * @ORM\Table(name="preferences",
11  *           uniqueConstraints={@ORM\UniqueConstraint(name="preferences_name_user_u
12  * )
13  */
14 class Preference
15 {
16     /**
17      * @ORM\Id
18      * @ORM\Column(type="integer")
19      * @ORM\GeneratedValue
20      */
21     protected $id;
22
23     /**
24      * @ORM\Column(type="string")
25      */
26     protected $name;
27
28     /**
29      * @ORM\Column(type="integer")
30      */
31     protected $value;
32
33     /**
34      * @ORM\ManyToOne(targetEntity="User", inversedBy="preferences")
35      * @var User
36      */
37     protected $user;
38
39     public function getId()
40     {
41         return $this->id;
```

II. Développement de l'API REST

```
42     }
43
44     public function setId($id)
45     {
46         $this->id = $id;
47     }
48
49     public function getName()
50     {
51         return $this->name;
52     }
53
54     public function setName($name)
55     {
56         $this->name = $name;
57     }
58
59     public function getValue()
60     {
61         return $this->value;
62     }
63
64     public function setValue($value)
65     {
66         $this->value = $value;
67     }
68
69     public function getUser()
70     {
71         return $this->user;
72     }
73
74     public function setUser(User $user)
75     {
76         $this->user = $user;
77     }
78 }
```

```
1 <?php
2 # src/AppBundle/Entity/User.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7 use Doctrine\Common\Collections\ArrayCollection;
8
9 /**
10 * @ORM\Entity()
```

II. Développement de l'API REST

```
11 * @ORM\Table(name="users",
12 *     uniqueConstraints={@ORM\UniqueConstraint(name="users_email_unique", column="email")}
13 * )
14 */
15 class User
16 {
17     // ...
18
19     /**
20     * @ORM\OneToMany(targetEntity="Preference", mappedBy="user")
21     * @var Preference[]
22     */
23     protected $preferences;
24
25     public function __construct()
26     {
27         $this->preferences = new ArrayCollection();
28     }
29
30     // ...
31
32     public function getPreferences()
33     {
34         return $this->preferences;
35     }
36
37     public function setPreferences($preferences)
38     {
39         $this->preferences = $preferences;
40     }
41 }
```

Le formulaire associé et les règles de validation sont proches de celui des thèmes.

```
1 <?php
2 # src/AppBundle/Form/Type/PreferenceType.php
3
4 namespace AppBundle\Form\Type;
5
6 use Symfony\Component\Form\AbstractType;
7 use Symfony\Component\Form\FormBuilderInterface;
8 use Symfony\Component\OptionsResolver\OptionsResolver;
9
10 class PreferenceType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array
13         $options)
14     {
```

II. Développement de l'API REST

```
14     $builder->add('name');
15     $builder->add('value');
16 }
17
18 public function configureOptions(OptionsResolver $resolver)
19 {
20     $resolver->setDefaults([
21         'data_class' => 'AppBundle\Entity\Preference',
22         'csrf_protection' => false
23     ]);
24 }
25 }
```

```
1 # src/AppBundle/Resources/config/validation.yml
2
3 # ...
4
5 AppBundle\Entity\Preference:
6     properties:
7         name:
8             - NotNull: ~
9             - Choice:
10                 choices: [art, architecture, history,
11                           science-fiction, sport]
12         value:
13             - NotNull: ~
14             - Type: numeric
15             - GreaterThan:
16                 value: 0
17             - LessThanOrEqual:
18                 value: 10
```

Un nouveau contrôleur sera aussi créé pour assurer la gestion des préférences utilisateurs via notre API.

```
1 <?php
2 # src/AppBundle/Controller/User/PreferenceController.php
3
4 namespace AppBundle\Controller\User;
5
6 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
```

II. Développement de l'API REST

```
11 use AppBundle\Form\Type\PreferenceType;
12 use AppBundle\Entity\Preference;
13
14 class PreferenceController extends Controller
15 {
16
17     /**
18      * @Rest\View(serializerGroups={"preference"})
19      * @Rest\Get("/users/{id}/preferences")
20      */
21     public function getPreferencesAction(Request $request)
22     {
23         $user = $this->get('doctrine.orm.entity_manager')
24             ->getRepository('AppBundle:User')
25             ->find($request->get('id'));
26         /* @var $user User */
27
28         if (empty($user)) {
29             return $this->userNotFound();
30         }
31
32         return $user->getPreferences();
33     }
34
35
36     /**
37      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"preference"})
38      * @Rest\Post("/users/{id}/preferences")
39      */
40     public function postPreferencesAction(Request $request)
41     {
42         $user = $this->get('doctrine.orm.entity_manager')
43             ->getRepository('AppBundle:User')
44             ->find($request->get('id'));
45         /* @var $user User */
46
47         if (empty($user)) {
48             return $this->userNotFound();
49         }
50
51         $preference = new Preference();
52         $preference->setUser($user);
53         $form = $this->createForm(PreferenceType::class,
54             $preference);
55
56         $form->submit($request->request->all());
57
58         if ($form->isValid()) {
59             $em = $this->get('doctrine.orm.entity_manager');
60             $em->persist($preference);
61         }
62     }
63 }
```

II. Développement de l'API REST

```
60         $em->flush();
61         return $preference;
62     } else {
63         return $form;
64     }
65 }
66
67 private function userNotFound()
68 {
69     return \FOS\RestBundle\View\View::create(['message' =>
70         'User not found'], Response::HTTP_NOT_FOUND);
71 }
```

```
1 # app/config/routing.yml
2
3 # ...
4
5 preferences:
6     type:     rest
7     resource: AppBundle\Controller\User\PreferenceController
```

Les groupes de sérialisation doivent aussi être mis à jour afin d'éviter les fameuses références circulaires.

Avec ces modifications que nous venons d'apporter, nous pouvons maintenant associer des thèmes et des préférences respectivement aux lieux et aux utilisateurs. Nous allons donc finaliser ce chapitre en rajoutant enfin les suggestions.

9.4. Proposer des suggestions aux utilisateurs

9.4.1. Calcul du niveau de correspondance

i

La technique utilisée pour trouver les suggestions n'est pas optimale. L'objectif ici est juste de présenter une méthode fonctionnelle et avoir une API complète.

L'algorithme pour calculer le niveau de correspondance va être implémenté dans l'entité `User`. À partir des thèmes d'un lieu, nous allons créer une méthode permettant de déterminer le niveau de correspondance (défini plus haut).

II. Développement de l'API REST

```
1 <?php
2 # src/AppBundle/Entity/User.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7 use Doctrine\Common\Collections\ArrayCollection;
8
9 /**
10 * @ORM\Entity()
11 * @ORM\Table(name="users",
12 *           uniqueConstraints={@ORM\UniqueConstraint(name="users_email_unique",column="email")})
13 */
14
15 class User
16 {
17     const MATCH_VALUE_THRESHOLD = 25;
18
19     // ...
20
21     public function preferencesMatch($themes)
22     {
23         $matchValue = 0;
24         foreach ($this->preferences as $preference) {
25             foreach ($themes as $theme) {
26                 if ($preference->match($theme)) {
27                     $matchValue += $preference->getValue() *
28                                 $theme->getValue();
29             }
30         }
31
32         return $matchValue >= self::MATCH_VALUE_THRESHOLD;
33     }
34 }
```

La méthode `match` de l'objet `Preference` permet juste de vérifier si le nom du thème est le même que celui de la préférence de l'utilisateur.

```
1 <?php
2 # src/AppBundle/Entity/Preference.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
```

```
9  * @ORM\Entity()
10 * @ORM\Table(name="preferences",
11 *     uniqueConstraints={@ORM\UniqueConstraint(name="preferences_name_user_u
12 * )
13 */
14 class Preference
15 {
16     // ...
17
18     public function match(Theme $theme)
19     {
20         return $this->name === $theme->getName();
21     }
22 }
```

9.4.2. Appel API pour récupérer les suggestions

Pour récupérer les suggestions, il nous suffit maintenant de créer un appel dans le contrôleur *UserController*.

```
1  <?php
2  # src/AppBundle/Controller/UserController.php
3
4  namespace AppBundle\Controller;
5
6  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
7  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
8  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
9  use Symfony\Component\HttpFoundation\JsonResponse;
10 use Symfony\Component\HttpFoundation\Response;
11 use Symfony\Component\HttpFoundation\Request;
12 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
13 use AppBundle\Form\Type\UserType;
14 use AppBundle\Entity\User;
15
16 class UserController extends Controller
17 {
18     // ...
19
20     /**
21      * @Rest\View(serializerGroups={"place"})
22      * @Rest\Get("/users/{id}/suggestions")
23      */
24     public function getUserSuggestionsAction(Request $request)
25     {
```


II. Développement de l'API REST

```
26     $user = $this->get('doctrine.orm.entity_manager')
27         ->getRepository('AppBundle:User')
28         ->find($request->get('id'));
29     /* @var $user User */
30
31     if (empty($user)) {
32         return $this->userNotFound();
33     }
34
35     $suggestions = [];
36
37     $places = $this->get('doctrine.orm.entity_manager')
38         ->getRepository('AppBundle:Place')
39         ->findAll();
40
41     foreach ($places as $place) {
42         if ($user->preferencesMatch($place->getThemes())) {
43             $suggestions[] = $place;
44         }
45     }
46
47     return $suggestions;
48 }
49
50 // ...
51
52 private function userNotFound()
53 {
54     return \FOS\RestBundle\View\View::create(['message' =>
55         'User not found'], Response::HTTP_NOT_FOUND);
56 }
```



Un fait important à relever ici est que la méthode, bien qu'étant dans le contrôleur des utilisateurs, renvoie des lieux. Le groupe de sérialisation utilisé est donc **place**.

Pour tester, nous avons un utilisateur défini comme suit :

```
1 {
2     "id": 1,
3     "firstname": "My",
4     "lastname": "Bis",
5     "email": "my.last@test.local",
6     "preferences": [
7         {
8             "id": 1,
```

II. Développement de l'API REST

```
9     "name": "history",
10    "value": 4
11  },
12  {
13    "id": 2,
14    "name": "art",
15    "value": 4
16  },
17  {
18    "id": 6,
19    "name": "sport",
20    "value": 3
21  }
22 ]
23 }
```

Et la liste de lieux dans l'application est la suivante :

```
1  [
2  {
3    "id": 1,
4    "name": "Tour Eiffel",
5    "address": "5 Avenue Anatole France, 75007 Paris",
6    "prices": [
7      {
8        "id": 1,
9        "type": "less_than_12",
10       "value": 5.75
11      }
12     ],
13    "themes": [
14      {
15        "id": 1,
16        "name": "architecture",
17        "value": 7
18      },
19      {
20        "id": 2,
21        "name": "history",
22        "value": 6
23      }
24     ]
25   },
26   {
27     "id": 2,
28     "name": "Mont-Saint-Michel",
29     "address": "50170 Le Mont-Saint-Michel",
30     "prices": [],
```

II. Développement de l'API REST

```
31     "themes": [  
32       {  
33         "id": 3,  
34         "name": "history",  
35         "value": 3  
36       },  
37       {  
38         "id": 4,  
39         "name": "art",  
40         "value": 7  
41       }  
42     ]  
43   }  
44 ]
```

Quand nous récupérons les suggestions pour notre utilisateur, nous obtenons :



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 9.1. – Récupération des suggestions pour l'utilisateur avec Postman

```
1  [  
2  {  
3    "id": 2,  
4    "name": "Mont-Saint-Michel",  
5    "address": "50170 Le Mont-Saint-Michel",  
6    "prices": [],  
7    "themes": [  
8      {  
9        "id": 3,  
10       "name": "history",  
11       "value": 3  
12     },  
13     {  
14       "id": 4,  
15       "name": "art",  
16       "value": 7  
17     }  
18   ]  
19 }  
20 ]
```

II. Développement de l'API REST

Nous avons donc un lieu dans notre application qui correspondrait aux goûts de notre utilisateur.

Les fonctionnalités que nous voulons pour notre application peuvent être implémentées assez facilement sans se soucier des contraintes imposées par le style d'architecture REST. REST n'intervient que pour définir l'API à utiliser pour accéder à ces fonctionnalités et nous laisse donc la responsabilité des choix techniques et de conceptions.

Vous pouvez vous entraîner et améliorer l'API en rajoutant encore plus de fonctionnalités. Nous pouvons par exemple imaginer que chaque utilisateur à un budget et que les tarifs des lieux sont pris en compte pour améliorer les suggestions.

10. REST à son paroxysme

Il reste un point sur les contraintes REST que nous n'avons toujours pas abordé : l'Hypermédia. En plus, notre API supporte un seul format le JSON. Toutes les requêtes et toutes les réponses sont en JSON. Nous imposons donc une contrainte aux futurs clients de notre API.

Pour remédier à cela, nous allons voir comment supporter facilement d'autre format de réponse en utilisant *FOSRestBundle* et le sérialiseur de Symfony. Et pour finir, nous verrons comment mettre en place de l'hypermédia dans une API REST, son utilité et comment l'exploiter (*si cela est possible*) ?

10.1. Supporter plusieurs formats de requêtes et de réponses

10.1.1. Cas des requêtes

Depuis que nous avons installé *FOSRestBundle*, notre API supporte déjà trois formats : le JSON, le format *x-www-form-urlencoded* (utilisé par les formulaires) et le XML.

Le body listener que nous avons activé utilise déjà par défaut ces trois formats. Pour déclarer le format utilisé dans la requête, il suffit d'utiliser l'entête HTTP `Content-Type` qui permet de décrire le type du contenu de la requête (et même de la réponse).

Avec Postman, nous pouvons tester la création d'un utilisateur en exploitant cette fonctionnalité. Au lieu d'avoir du JSON, nous devons juste formater la requête en XML. Le corps de la requête doit être :

```
1 <user>
2   <firstname>test</firstname>
3   <lastname>XML</lastname>
4   <email>test@xml.fr</email>
5 </user>
```

Chaque format a un type **MIME** qui permet de le décrire avec l'entête `Content-Type` :

- JSON : `Application/json`
- XML : `application/xml`



C'est au client de définir dans sa requête le format utilisé pour que le serveur puisse la traiter correctement.

II. Développement de l'API REST

Avec Postman, il y a un onglet **Headers** qui permet de rajouter des entêtes HTTP. Pour faciliter le travail, nous pouvons aussi choisir dans l'onglet **Body**, le contenu de la requête. Postman rajoutera automatiquement le bon type **MIME** de la requête à notre place.



FIGURE 10.1. – Choix du type de contenu avec Postman



FIGURE 10.2. – Entête rajoutée par Postman

En envoyant la requête, l'utilisateur est créé et nous obtenons une réponse en ... JSON! Nous allons donc voir dans la partie suivante comment autoriser plusieurs formats de réponses comme nous l'avons déjà pour les requêtes.



Il est possible de supporter d'autres formats en plus de celle par défaut. Pour en savoir plus, vous pouvez consulter [la documentation officielle](#) ↗.

10.1.2. Cas des réponses

L'utilisation de l'annotation **View** de *FOSRestBundle* permet de créer des réponses qui peuvent être affichées dans différents formats. Dans tous nos contrôleurs, nous nous contentons de renvoyer un objet ou un tableau et ces données sont envoyées au client dans le bon format.

Pour supporter plusieurs formats, les données renvoyées par les contrôleurs ne changent pas. Nous devons juste configurer *FOSRestBundle* correctement. Ce bundle supporte deux types de réponses :

- celles ne nécessitant pas de template pour être affichées : celles au format JSON, au format XML, etc. Il suffit d'avoir les données pour les encoder et le sérialiseur fait le reste du travail.
- celles qui nécessitent un template : le html, etc. Pour ce genre de réponse, nous devons avoir des informations en plus permettant de *décorer* la réponse (mise en page, CSS, etc.) et le moteur de rendu (ici Twig) s'occupe du reste.

II. Développement de l'API REST

Dans le cadre du cours, nous allons juste aborder le premier type de réponse. [La documentation](#) couvre bien l'ensemble du sujet si cela vous intéresse.

Pour activer ces fonctionnalités, nous devons configurer deux sections. La première nous permettra de déclarer les formats de réponses supportés et la seconde nous permettra de configurer la priorité entre ces formats, le comportement du serveur si aucun format n'est choisi par le client, etc.

Nous allons supporter les formats JSON et XML pour les réponses. La configuration devient maintenant (la clé `formats` a été rajoutée) :

```
1 # app/config/config.yml
2 # ...
3
4 fos_rest:
5   routing_loader:
6     include_format: false
7   view:
8     view_response_listener: true
9     formats:
10      json: true
11      xml: true
12   format_listener:
13     rules:
14       - { path: '^/', priorities: ['json'], fallback_format:
15           'json', prefer_extension: false }
16   body_listener:
17     enabled: true
```

En réalité, ces deux formats sont déjà activés par défaut mais par soucis de clarté nous allons les laisser visibles dans le fichier de configuration.

Le reste de la configuration se fait avec la clé `rules`. C'est au niveau des priorités (clé `priorities`) que les formats supportés sont définis. Pour notre configuration, nous avons une seule règle. Mais il est tout à fait possible de définir plusieurs règles différentes selon les URL utilisées. Nous pouvons imaginer par exemple une règle par version de l'api, ou bien encore une règle par ressources.

Il suffit de rajouter le format XML aux priorités et notre API pourra répondre aussi bien en XML qu'en JSON.

```
1 # app/config/config.yml
2 # ...
3
4 fos_rest:
5   routing_loader:
6     include_format: false
7   view:
```

II. Développement de l'API REST

```
8     view_response_listener: true
9     formats:
10         json: true
11         xml: true
12     format_listener:
13         rules:
14             - { path: '^/', priorities: ['json', 'xml'],
15               fallback_format: 'json', prefer_extension: false }
15     body_listener:
16         enabled: true
```

i

C'est maintenant au client d'informer le serveur sur le ou les formats qu'il préfère.

×

L'ordre de déclaration est très important ici. Si une requête ne spécifie aucun format alors le serveur choisira du JSON.

10.1.3. La négociation de contenu

La négociation de contenu est un mécanisme [du protocole HTTP](#) qui permet de proposer plusieurs formats pour une même ressource. Pour sa mise en œuvre, le client doit envoyer un entête HTTP de la famille **Accept**. Nous avons entre autres :

| Entête | Utilisation |
|-----------------|--|
| Accept | Pour choisir un média type (text, json, html etc). |
| Accept-Charset | Pour choisir le jeu de caractères (iso-8859-1, utf8, etc.) |
| Accept-Language | Pour choisir le langage (français, anglais, etc.) |

L'entête qui nous intéresse ici est **Accept**. Comme pour l'entête **Content-Type**, la valeur de cet entête doit contenir un type **MIME**.

Mais en plus, avec cet entête, nous pouvons déclarer plusieurs formats à la fois en prenant le soin de définir un ordre de préférence en utilisant un facteur de qualité.

i

Le facteur de qualité (**q**) est un nombre compris entre 0 et 1 qui permet de définir l'ordre de préférence. Plus **q** est élevé, plus le type **MIME** associé est prioritaire.

II. Développement de l'API REST

Une requête avec comme entête `Accept: application/json;q=0.7, application/xml;q=1`, veut dire que le client préfère du XML et en cas d'indisponibilité du XML alors du JSON.

Une requête avec comme entête `Accept: application/xml` veut dire que le client préfère du XML. Si le facteur de qualité n'est pas spécifié, sa valeur est à 1.

Pour tester, nous allons ajouter cet entête à une requête pour lister tous les lieux de notre API.



FIGURE 10.3. – Récupération des lieux en XML avec Postman

La réponse est bien en XML et nous pouvons tester avec n'importe quelle méthode de notre API.

```
1 <?xml version="1.0"?>
2 <response>
3   <item key="0">
4     <id>1</id>
5     <name>Tour Eiffel</name>
6     <address>5 Avenue Anatole France, 75007 Paris</address>
7     <prices>
8       <id>1</id>
9       <type>less_than_12</type>
10      <value>5.75</value>
11    </prices>
12    <themes>
13      <id>1</id>
14      <name>architecture</name>
15      <value>7</value>
16    </themes>
17    <themes>
18      <id>2</id>
19      <name>history</name>
20      <value>6</value>
21    </themes>
22  </item>
23  <item key="1">
24    <id>2</id>
25    <name>Mont-Saint-Michel</name>
26    <address>50170 Le Mont-Saint-Michel</address>
27    <prices/>
28    <themes>
29      <id>3</id>
30      <name>history</name>
```

II. Développement de l'API REST

```
31         <value>3</value>
32     </themes>
33     <themes>
34         <id>4</id>
35         <name>art</name>
36         <value>7</value>
37     </themes>
38 </item>
39 <item key="2">
40     <id>4</id>
41     <name>Disneyland Paris</name>
42     <address>77777 Marne-la-Vallée</address>
43     <prices/>
44     <themes/>
45 </item>
46 <item key="3">
47     <id>5</id>
48     <name>Aquaboulevard</name>
49     <address>4-6 Rue Louis Armand, 75015 Paris</address>
50     <prices/>
51     <themes/>
52 </item>
53 <item key="4">
54     <id>6</id>
55     <name>test</name>
56     <address>test</address>
57     <prices/>
58     <themes/>
59 </item>
60 </response>
```

Le serveur renvoie aussi un entête `Content-Type` pour signaler au client le format de la réponse.



FIGURE 10.4. – Entête renvoyée par le serveur pour le format de la réponse



Attention, certaines API proposent de rajouter un format à une URL pour sélectionner un format de réponse (places.json, places.xml, etc.). Cette technique ne respecte pas les contraintes REST vu que l'URL doit juste servir à identifier une ressource.

10.2. L'Hypermédia

La dernière contrainte du REST que nous n'avons pas encore implémentée est l'hypermédia en tant que moteur de l'état de l'application *HATEOAS*. Pour rappel, le contrôle hypermédia désigne l'état d'une application ou API avec un seul point d'entrée mais qui propose des éléments permettant de l'explorer et d'interagir avec elle.

Avec un humain qui surfe sur le web, il est facile de suivre cette contrainte. En général, nous utilisons tous des sites web en tapant sur notre navigateur l'URL de la page d'accueil. Ensuite, avec les différents liens et formulaires, nous interagissons avec ledit site. Un site web est l'exemple parfait du concept HATEOAS.

Pour une API, nous avons des outils comme [BazingaHateoasBundle](#) qui permettent d'avoir un *semblant* de HATEOAS.

Une fois configuré, voici un exemple de réponse lorsqu'on récupère un utilisateur (exemple issu de [la documentation du bundle](#)).

```
1 {
2   "id": 42,
3   "first_name": "Adrien",
4   "last_name": "Brault",
5   "_links": {
6     "self": {
7       "href": "/api/users/42"
8     },
9     "manager": {
10      "href": "/api/users/23"
11    }
12  },
13  "_embedded": {
14    "manager": {
15      "id": 23,
16      "first_name": "Will",
17      "last_name": "Durand",
18      "_links": {
19        "self": {
20          "href": "/api/users/23"
21        }
22      }
23    }
24  }
25 }
```

Les attributs `_links` et `_embedded` sont issus des spécifications [Hypertext Application Language \(HAL\)](#). Ils permettent de décrire notre ressource en suivant les spécifications HAL encore à l'état de brouillon.

Des initiatives identiques comme [JSON for Linking Data \(json-ld\)](#) tentent de traiter le problème mais se heurtent tous face à un même obstacle.

II. Développement de l'API REST

La contrainte HATEOAS de REST nécessite un client très intelligent qui puisse :

- comprendre les relations déclarées entre ressource ;
- auto-découvrir notre API à partir d'une seule URL.

Malheureusement, il n'existe pas encore de client d'API en mesure de comprendre et d'exploiter une API RESTful niveau 3 (selon le modèle de Richardson).

Nous n'implémenterons donc pas cette contrainte et c'est le cas pour beaucoup d'API REST. Dans les faits, cela ne pose aucun problème et notre API est pleinement fonctionnelle.

Le support de plusieurs formats de requêtes et de réponses se fait en utilisant la négociation de contenu. Les entêtes mis en œuvre pour atteindre un tel comportement sont **Accept** et **Content-Type**. *FOSRestBundle* exploite ensuite les capacités de notre sérialiseur afin de produire des réponses pour différents formats en se basant sur les mêmes données.

Troisième partie

Amélioration de l'API REST

11. Sécurisation de l'API 1/2

Jusque-là, les actions disponibles dans notre API sont accessibles pour n'importe quel client. Nous ne disposons d'aucun moyen pour gérer l'identité de ces derniers.

Pour être bref, n'importe qui peut faire n'importe quoi avec notre API.

i

La sécurité n'est pas un sujet adressé par les concepts REST mais nous pouvons adapter les méthodes d'autorisation et d'authentification classiques aux principes REST.

Il existe beaucoup de techniques et d'outils comme [OAuth](#) ou [JSON Web Tokens](#) permettant de mettre en place un système d'authentification.

Cependant nous ne nous baserons sur aucun de ces outils et nous allons mettre en place un système d'authentification totalement personnalisé.

11.1. Connexion et déconnexion avec une API

Qui dit système d'authentification dit des opérations comme *se connecter* et *se déconnecter*.

?

Comment mettre en place un tel système en se basant sur des concepts REST ?

Pour bien adapter ses opérations, il faut d'abord bien les comprendre.

En général, lorsque nous nous connectons à un site web, nous fournissons un login et un mot de passe via un formulaire de connexion. Si les informations fournies sont valides, le serveur crée une *cookie* qui permettra d'assurer la gestion de la session. Une fois que nous avons fini de naviguer sur le site, il suffit de nous déconnecter pour que la *cookie* de session soit supprimée.



FIGURE 11.1. – Cycle d'authentification

Nous avons donc 3 éléments essentiels pour un tel fonctionnement :

- une méthode pour se connecter ;

III. Amélioration de l'API REST

- une méthode pour se déconnecter ;
- et une entité pour suivre l'utilisateur pendant sa navigation (le cookie).

En REST toutes nos opérations doivent se faire sur des *ressources*.

Pour rappel,

Du moment où vous devez interagir avec une entité de votre application, créer une entité, la modifier, la consulter ou que vous devez l'identifier de manière unique alors vous avez pour la plupart des cas une ressource.

Les opérations se font sur le cookie, nous pouvons donc dire qu'il représente notre ressource. Pour le cas d'un site web, l'utilisation d'un cookie est pratique vue que les navigateurs le gèrent nativement (envoi à chaque requête, limitation à un seul domaine pour la sécurité, durée de validité, etc.).

Pour le cas d'une API, il est certes possible d'utiliser un cookie mais il existe une solution équivalente mais plus simple et plus courante : les tokens.

i

Donc *se connecter* ou encore *se déconnecter* se traduisent respectivement par créer un token d'authentification et supprimer son token d'authentification.

Pour chaque requête, le token ainsi créé est rajouté en utilisant une entête HTTP comme pour les cookies.

Commençons d'abord par gérer la création des tokens.

11.2. Login et mot de passe pour les utilisateurs

Avant de créer un token, nous devons mettre à jour notre modèle de données. Un utilisateur doit maintenant avoir un mot de passe et son *adresse mail* sera son *login*. Pour la gestion de ce mot de passe, nous utiliserons [les outils](#) que nous propose Symfony.

Le nouveau modèle utilisateur :

```
1 # src/AppBundle/Entity/User.php
2 <?php
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6 use Doctrine\Common\Collections\ArrayCollection;
7
8 /**
9  * @ORM\Entity()
10  * @ORM\Table(name="users",
11  *           uniqueConstraints={@ORM\UniqueConstraint(name="users_email_unique", column="email")})
12  * )
13 */
```

III. Amélioration de l'API REST

```
14 class User
15 {
16     //...
17
18     /**
19      * @ORM\Column(type="string")
20      */
21     protected $password;
22
23     protected $plainPassword;
24
25     // ... tous les getters et setters
26 }
```

L'attribut `plainPassword` ne sera pas sauvegardé en base. Il nous permettra de conserver le mot de passe de l'utilisateur en clair à sa création ou modification.

Comme toujours, n'oubliez pas de mettre à jour la base de données :

```
1 php bin/console doctrine:schema:update --dump-sql --force
2
3 ALTER TABLE users ADD password VARCHAR(255) NOT NULL;
4
5 Updating database schema...
6 Database schema updated successfully! "1" query was executed
```

La création d'un utilisateur nécessite maintenant un léger travail supplémentaire. À la création, il faudra fournir un mot de passe en claire que nous hasherons avant de le sauvegarder en base. Rajoutons donc les configurations de sécurité de Symfony :

```
1 # To get started with security, check out the documentation:
2 # http://symfony.com/doc/current/book/security.html
3 security:
4
5     # Ajout d'un encodeur pour notre entité User
6     encoders:
7         AppBundle\Entity\User:
8             algorithm: bcrypt
9             cost: 12
```

Notre entité utilisateur doit implémenter l'interface `UserInterface` :

```
1 # src/AppBundle/Entity/User.php
2 <?php
```


III. Amélioration de l'API REST

```
3 namespace AppBundle\Entity;
4
5 use Symfony\Component\Security\Core\User\UserInterface;
6 use Doctrine\ORM\Mapping as ORM;
7 use Doctrine\Common\Collections\ArrayCollection;
8
9 /**
10 * @ORM\Entity()
11 * @ORM\Table(name="users",
12 *     uniqueConstraints={@ORM\UniqueConstraint(name="users_email_unique", column="email")}
13 * )
14 */
15 class User implements UserInterface
16 {
17     // ...
18
19     public function getPassword()
20     {
21         return $this->password;
22     }
23
24     public function setPassword($password)
25     {
26         $this->password = $password;
27     }
28
29
30     public function getRoles()
31     {
32         return [];
33     }
34
35     public function getSalt()
36     {
37         return null;
38     }
39
40     public function getUsername()
41     {
42         return $this->email;
43     }
44
45     public function eraseCredentials()
46     {
47         // Suppression des données sensibles
48         $this->plainPassword = null;
49     }
50 }
```

Le formulaire de création d'utilisateur et l'action associée dans notre contrôleur vont être adaptés

III. Amélioration de l'API REST

en conséquence :

```
1 # src/AppBundle/Form/Type/UserType.php
2 <?php
3 namespace AppBundle\Form\Type;
4
5 use Symfony\Component\Form\AbstractType;
6 use Symfony\Component\Form\FormBuilderInterface;
7 use Symfony\Component\OptionsResolver\OptionsResolver;
8 use Symfony\Component\Form\Extension\Core\Type\EmailType;
9
10 class UserType extends AbstractType
11 {
12     public function buildForm(FormBuilderInterface $builder, array
13         $options)
14     {
15         $builder->add('firstname');
16         $builder->add('lastname');
17         $builder->add('plainPassword'); // Rajout du mot de passe
18         $builder->add('email', EmailType::class);
19     }
20     // ...
21 }
```

Pour le mot de passe, nous aurons juste quelques règles de validation basiques :

```
1 # src/AppBundle/Resources/config/validation.yml
2
3 # ...
4
5 AppBundle\Entity\User:
6     constraints:
7         -
8             Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity:
9                 email
10     properties:
11         firstname:
12             - NotBlank: ~
13             - Type: string
14         lastname:
15             - NotBlank: ~
16             - Type: string
17         email:
18             - NotBlank: ~
19             - Email: ~
20         plainPassword:
```

III. Amélioration de l'API REST

```
19         - NotBlank: { groups: [New, FullUpdate] }
20         - Type: string
21         - Length:
22             min: 4
23             max: 50
24 # ...
```

Le champ `plainPassword` est un champ un peu spécial. Les groupes nous permettront d'activer sa contrainte `NotBlank` lorsque le client voudra créer ou mettre à jour tous les champs de l'utilisateur. Mais lors d'une mise à jour partielle (PATCH), si le champ est nul, il sera tout simplement ignoré.



Le mot de passe ne doit en aucun cas être sérialisé. Il ne doit pas être associé à un groupe de sérialisation.

La création et la modification d'un utilisateur nécessite maintenant un hashage du mot de passe en clair, le service `password_encoder` de Symfony fait ce travail pour nous en utilisant toutes les configurations que nous venons de mettre en place.

```
1 # src/AppBundle/Controller/UserController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
7 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use Symfony\Component\HttpFoundation\Request;
11 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
12 use AppBundle\Form\Type\UserType;
13 use AppBundle\Entity\User;
14
15 class UserController extends Controller
16 {
17     /**
18      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"user"})
19      * @Rest\Post("/users")
20      */
21     public function postUsersAction(Request $request)
22     {
23         $user = new User();
24         $form = $this->createForm(UserType::class, $user,
25             ['validation_groups'=>['Default', 'New']]);
```

III. Amélioration de l'API REST

```
26     $form->submit($request->request->all());
27
28     if ($form->isValid()) {
29         $encoder = $this->get('security.password_encoder');
30         // le mot de passe en clair est encodé avant la
           sauvegarde
31         $encoded = $encoder->encodePassword($user,
           $user->getPlainPassword());
32         $user->setPassword($encoded);
33
34         $em = $this->get('doctrine.orm.entity_manager');
35         $em->persist($user);
36         $em->flush();
37         return $user;
38     } else {
39         return $form;
40     }
41 }
42
43 /**
44  * @Rest\View(serializerGroups={"user"})
45  * @Rest\Put("/users/{id}")
46  */
47 public function updateUserAction(Request $request)
48 {
49     return $this->updateUser($request, true);
50 }
51
52 /**
53  * @Rest\View(serializerGroups={"user"})
54  * @Rest\Patch("/users/{id}")
55  */
56 public function patchUserAction(Request $request)
57 {
58     return $this->updateUser($request, false);
59 }
60
61 private function updateUser(Request $request, $clearMissing)
62 {
63     $user = $this->get('doctrine.orm.entity_manager')
64         ->getRepository('AppBundle:User')
65         ->find($request->get('id')); // L'identifiant en
           tant que paramètre n'est plus nécessaire
66     /* @var $user User */
67
68     if (empty($user)) {
69         return $this->userNotFound();
70     }
71 }
```

III. Amélioration de l'API REST

```
72     if ($clearMissing) { // Si une mise à jour complète, le mot
73         de passe doit être validé
74         $options = ['validation_groups'=>['Default',
75             'FullUpdate']];
76     } else {
77         $options = []; // Le groupe de validation par défaut de
78         Symfony est Default
79     }
80
81     $form = $this->createForm(UserType::class, $user,
82         $options);
83
84     $form->submit($request->request->all(), $clearMissing);
85
86     if ($form->isValid()) {
87         // Si l'utilisateur veut changer son mot de passe
88         if (!empty($user->getPlainPassword())) {
89             $encoder = $this->get('security.password_encoder');
90             $encoded = $encoder->encodePassword($user,
91                 $user->getPlainPassword());
92             $user->setPassword($encoded);
93         }
94         $em = $this->get('doctrine.orm.entity_manager');
95         $em->merge($user);
96         $em->flush();
97         return $user;
98     } else {
99         return $form;
100    }
101 }
102 }
```

Le groupe de validation `Default` regroupe toutes les contraintes de validation qui ne sont dans aucun groupe. Il est créé automatiquement par Symfony. N'hésitez surtout pas à consulter [la documentation](#) pour des informations plus détaillées avant de continuer.

Nous pouvons maintenant tester la création d'un utilisateur en fournissant un mot de passe.

<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 11.2. – Requête de création d'un utilisateur avec mot de passe

L'utilisateur est créé et la réponse ne contient aucun mot de passe :

```
1 {
2   "id": 5,
3   "firstname": "test",
4   "lastname": "Pass",
5   "email": "test@pass.fr",
6   // ...
7 }
```



Toutes les modifications effectuées ici sont propres à Symfony. Si vous avez du mal à suivre, il est vivement (grandement) conseillé de consulter [la documentation officielle](#) du framework.

11.3. Création d'un token

Revenons maintenant à notre système d'authentification avec des tokens. Un token aura les caractéristiques suivantes :

- une valeur : une suite de chaînes de caractères générées aléatoirement et *unique* ;
- une date de création : la date à laquelle le token a été créé. Cette date nous permettra plus tard de vérifier l'âge du token et donc sa validité du token ;
- un utilisateur : une référence vers l'utilisateur qui a demandé la création de ce token. Comme pour toute ressource, nous avons besoin d'une URL pour l'identifier. Nous utiliserons `rest-api.local/auth-tokens`, `auth-tokens` étant juste le diminutif de *authentication tokens* i.e les tokens d'authentification.

Contrairement aux autres ressources, la requête de création d'un token est légèrement différente. Le payload contiendra le login et le mot de passe de l'utilisateur et les informations qui décrivent le token seront générées par le serveur.

La réponse contiendra donc les informations ainsi créées.



FIGURE 11.3. – Cinématique de création de token

L'implémentation va donc ressembler à tout ce que nous avons déjà fait.

Commençons par l'entité `AuthToken` :

III. Amélioration de l'API REST

```
1 # src/AppBundle/Entity/AuthToken.php
2 <?php
3 namespace AppBundle\Entity;
4
5 use Doctrine\ORM\Mapping as ORM;
6
7 /**
8  * @ORM\Entity()
9  * @ORM\Table(name="auth_tokens",
10 *     uniqueConstraints={@ORM\UniqueConstraint(name="auth_tokens_value_unique
11 * )
12 */
13 class AuthToken
14 {
15     /**
16     * @ORM\Id
17     * @ORM\Column(type="integer")
18     * @ORM\GeneratedValue
19     */
20     protected $id;
21
22     /**
23     * @ORM\Column(type="string")
24     */
25     protected $value;
26
27     /**
28     * @ORM\Column(type="datetime")
29     * @var \DateTime
30     */
31     protected $createdAt;
32
33     /**
34     * @ORM\ManyToOne(targetEntity="User")
35     * @var User
36     */
37     protected $user;
38
39
40     public function getId()
41     {
42         return $this->id;
43     }
44
45     public function setId($id)
46     {
47         $this->id = $id;
48     }
49
```

III. Amélioration de l'API REST

```
50     public function getValue()  
51     {  
52         return $this->value;  
53     }  
54  
55     public function setValue($value)  
56     {  
57         $this->value = $value;  
58     }  
59  
60     public function getCreatedAt()  
61     {  
62         return $this->createdAt;  
63     }  
64  
65     public function setCreatedAt(\DateTime $createdAt)  
66     {  
67         $this->createdAt = $createdAt;  
68     }  
69  
70     public function getUser()  
71     {  
72         return $this->user;  
73     }  
74  
75     public function setUser(User $user)  
76     {  
77         $this->user = $user;  
78     }  
79 }
```

La mise à jour de la base de données avec Doctrine :

```
1 php bin/console doctrine:schema:update --dump-sql --force  
2 #> CREATE TABLE auth_tokens (id INT AUTO_INCREMENT NOT NULL,  
   user_id INT DEFAULT NULL, value VARCHAR(255) NOT NULL,  
   created_at DATETIME NOT NULL, INDEX IDX_8AF9B66CA76ED395  
   (user_id), UNIQUE INDEX auth_tokens_value_unique (value),  
   PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE  
   utf8_unicode_ci ENGINE = InnoDB;  
3 #> ALTER TABLE auth_tokens ADD CONSTRAINT FK_8AF9B66CA76ED395  
   FOREIGN KEY (user_id) REFERENCES users (id);  
4  
5 #> Updating database schema...  
6 #> Database schema updated successfully! "2" queries were executed
```

Pour la gestion du login et du mot de passe de l'utilisateur, nous allons créer :

III. Amélioration de l'API REST

- une entité nommée **Credentials** avec deux attributs : `login` et `password`. Cette entité n'aura aucune annotation Doctrine, elle permettra juste de transporter ces informations;
- un formulaire nommé **CredentialsType** pour valider que les champs de l'entité **Credentials** ne sont pas vides (`NotBlank`).

L'entité ressemble donc à :

```
1 # src/AppBundle/Entity/Credentials.php
2 <?php
3 namespace AppBundle\Entity;
4
5 class Credentials
6 {
7     protected $login;
8
9     protected $password;
10
11     public function getLogin()
12     {
13         return $this->login;
14     }
15
16     public function setLogin($login)
17     {
18         $this->login = $login;
19     }
20
21     public function getPassword()
22     {
23         return $this->password;
24     }
25
26     public function setPassword($password)
27     {
28         $this->password = $password;
29     }
30 }
```

Le formulaire et les règles de validation associées :

```
1 # src/AppBundle/Form/Type/CredentialsType.php
2 <?php
3 namespace AppBundle\Form\Type;
4
5 use Symfony\Component\Form\AbstractType;
6 use Symfony\Component\Form\FormBuilderInterface;
7 use Symfony\Component\OptionsResolver\OptionsResolver;
8
```

III. Amélioration de l'API REST

```
 9 class CredentialsType extends AbstractType
10 {
11     public function buildForm(FormBuilderInterface $builder, array
12         $options)
13     {
14         $builder->add('login');
15         $builder->add('password');
16     }
17     public function configureOptions(OptionsResolver $resolver)
18     {
19         $resolver->setDefaults([
20             'data_class' => 'AppBundle\Entity\Credentials',
21             'csrf_protection' => false
22         ]);
23     }
24 }
```

```
 1 # src/AppBundle/Resources/config/validation.yml
 2
 3 # ...
 4
 5 AppBundle\Entity\Credentials:
 6     properties:
 7         login:
 8             - NotBlank: ~
 9             - Type: string
10         password:
11             - NotBlank: ~
12             - Type: string
```

Il ne faut pas oublier de configurer le sérialiseur pour afficher le token en utilisant un groupe prédéfini.

```
 1 # src/AppBundle/Resources/config/serialization.yml
 2 # ...
 3
 4 AppBundle\Entity\User:
 5     attributes:
 6         id:
 7             groups: ['user', 'preference', 'auth-token']
 8         firstname:
 9             groups: ['user', 'preference', 'auth-token']
10         lastname:
11             groups: ['user', 'preference', 'auth-token']
12         email:
```

III. Amélioration de l'API REST

```
13         groups: ['user', 'preference', 'auth-token']
14     preferences:
15         groups: ['user']
16
17 AppBundle\Entity\AuthToken:
18     attributes:
19         id:
20             groups: ['auth-token']
21         value:
22             groups: ['auth-token']
23         createdAt:
24             groups: ['auth-token']
25         user:
26             groups: ['auth-token']
```

Maintenant, il ne reste plus qu'à créer le contrôleur qui assure la gestion des tokens d'authentification.

```
1 # src/AppBundle/Controller/AuthTokenController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Response;
8 use Symfony\Component\HttpFoundation\Request;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
   toutes les annotations
10 use AppBundle\Form\Type\CredentialsType;
11 use AppBundle\Entity\AuthToken;
12 use AppBundle\Entity\Credentials;
13
14 class AuthTokenController extends Controller
15 {
16     /**
17      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"auth-t
18      * @Rest\Post("/auth-tokens")
19      */
20     public function postAuthTokensAction(Request $request)
21     {
22         $credentials = new Credentials();
23         $form = $this->createForm(CredentialsType::class,
24             $credentials);
25
26         $form->submit($request->request->all());
27
28         if (!$form->isValid()) {
29             return $form;
```

III. Amélioration de l'API REST

```
29     }
30
31     $em = $this->get('doctrine.orm.entity_manager');
32
33     $user = $em->getRepository('AppBundle:User')
34         ->findOneByEmail($credentials->getLogin());
35
36     if (!$user) { // L'utilisateur n'existe pas
37         return $this->invalidCredentials();
38     }
39
40     $encoder = $this->get('security.password_encoder');
41     $isPasswordValid = $encoder->isPasswordValid($user,
42         $credentials->getPassword());
43
44     if (!$isPasswordValid) { // Le mot de passe n'est pas
45         correct
46         return $this->invalidCredentials();
47     }
48
49     $authToken = new AuthToken();
50     $authToken->setValue(base64_encode(random_bytes(50)));
51     $authToken->setCreatedAt(new \DateTime('now'));
52     $authToken->setUser($user);
53
54     $em->persist($authToken);
55     $em->flush();
56
57     return $authToken;
58 }
59
60 private function invalidCredentials()
61 {
62     return \FOS\RestBundle\View\View::create(['message' =>
63         'Invalid credentials'], Response::HTTP_BAD_REQUEST);
64 }
65 }
```

N'oublions pas de déclarer le nouveau contrôleur.

```
1 # app/config/routing.yml
2 # ...
3
4 auth-tokens:
5     type:     rest
6     resource: AppBundle\Controller\AuthTokenController
```



Pour des raisons de sécurité, nous évitons de donner des détails sur les comptes existants, un même message - `Invalid Credentials` - est renvoyé lorsque le login n'existe pas ou lorsque le mot de passe n'est pas correct.

Nous pouvons maintenant créer un token en utilisant le compte `test@pass.fr` créé plus tôt.

```
1 {
2   "login": "test@pass.fr",
3   "password": "test"
4 }
```



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 11.4. – Requête de création d'un token d'authentification avec Postman

La réponse contient un token que nous pourrions exploiter plus tard pour décliner notre identité.

```
1 {
2   "id": 3,
3   "value":
4     "MVgq3dT8QyWv3t+s7DLyvsquVbu+m0SPMdYX7VUQOEQcJGwaGD8ETa+zi9ReHPWYFKI=",
5   "createdAt": "2016-04-08T17:49:00+00:00",
6   "user": {
7     "id": 5,
8     "firstname": "test",
9     "lastname": "Pass",
10    "email": "test@pass.fr"
11  }
```

Nous disposons maintenant d'un système fonctionnel pour générer des tokens pour les utilisateurs. Ces tokens nous permettront par la suite de vérifier l'identité des clients de l'API afin de la sécuriser.

12. Sécurisation de l'API 2/2

Un client utilisant notre API est maintenant en mesure de créer des tokens d'authentification. Nous allons donc rajouter un système de sécurité afin d'imposer l'utilisation de ce token pour accéder à notre API REST. Au lieu d'envoyer le login et le mot de passe dans chaque requête, nous utiliserons le token associé au client.

12.1. Exploisons le token grâce à Symfony

Vu que nous allons imposer l'utilisation du token dans toutes les requêtes, nous devons vérifier sa validité afin de nous assurer que le client de l'API est bien authentifié.

Pour nous assurer de ce bon fonctionnement, chaque requête doit contenir une entête `X-Auth-Token` qui contiendra notre token fraîchement créée.

i

Le nom de notre entête ne vient pas du néant. De manière conventionnelle, lorsqu'une requête contient une entête n'appartenant pas aux spécifications HTTP, le nom débute par **X-**. Ensuite, le reste du nom reflète le contenu de l'entête, **Auth-Token** pour **Authentication Token**.

Nous avons beaucoup d'exemples dans notre API actuelle qui suivent ce modèle de nommage. Lorsque nous consultons les entêtes d'une réponse quelconque de notre API, nous pouvons voir **X-Debug-Token** (créé par Symfony en mode config dev) ou encore **X-Powered-By** (créé par PHP).



FIGURE 12.1. – Entêtes personnalisées renvoyées par notre API

Symfony dispose d'un mécanisme spécifique permettant de gérer les clés d'API. Il existe [un cookbook](#) décrivant de manière très succincte les mécanismes en jeu pour le mettre en place.

Pour résumer, à chaque requête de l'utilisateur, un listener est appelé afin de vérifier que la requête contient une entête nommée `X-Auth-Token`. Et si tel est le cas, son existence dans notre base de données et sa validité sont vérifiées.



Pour une requête permettant de créer un token d'authentification, ce listener ne fait aucune action afin d'autoriser la requête.



FIGURE 12.2. – Cinématique d'authentification en succès

Pour simplifier notre implémentation, nous considérons qu'un token d'authentification est invalide si son ancienneté est supérieur à 12 heures. Vous pouvez cependant modifier ce comportement et définir les règles de validité que vous voulez.



FIGURE 12.3. – Cinématique d'authentification en erreur

Comme pour tous les systèmes d'authentification de Symfony, nous avons besoin d'un fournisseur d'utilisateurs (`UserProvider`). Pour notre cas, il faut que notre fournisseur puisse charger un token en utilisant la valeur dans notre entête `X-Auth-Token`.

```
1 <?php
2 # src/AppBundle/Security/AuthTokenUserProvider.php
3
4 namespace AppBundle\Security;
5
6 use Symfony\Component\Security\Core\User\UserProviderInterface;
7 use Symfony\Component\Security\Core\User\User;
8 use Symfony\Component\Security\Core\User\UserInterface;
9 use
    Symfony\Component\Security\Core\Exception\UnsupportedUserException;
10 use Doctrine\ORM\EntityRepository;
11
12 class AuthTokenUserProvider implements UserProviderInterface
13 {
14     protected $authTokenRepository;
15     protected $userRepository;
16
```

III. Amélioration de l'API REST

```
17     public function __construct(EntityRepository
18         $authTokenRepository, EntityRepository $userRepository)
19     {
20         $this->authTokenRepository = $authTokenRepository;
21         $this->userRepository = $userRepository;
22     }
23     public function getAuthToken($authTokenHeader)
24     {
25         return $this->authTokenRepository->findOneByValue($authTokenHeader);
26     }
27
28     public function loadUserByUsername($email)
29     {
30         return $this->userRepository->findByEmail($email);
31     }
32
33     public function refreshUser(UserInterface $user)
34     {
35         // Le système d'authentification est stateless, on ne doit
36         // donc jamais appeler la méthode refreshUser
37         throw new UnsupportedUserException();
38     }
39     public function supportsClass($class)
40     {
41         return 'AppBundle\Entity\User' === $class;
42     }
43 }
```

Cette classe permettra de récupérer les utilisateurs en se basant sur le token d'authentification fourni.

Pour piloter le mécanisme d'authentification, nous devons créer une classe implémentant l'interface `SimplePreAuthenticatorInterface` de Symfony. C'est cette classe qui gère la cinématique d'authentification que nous avons décrite plus haut.

```
1 # src/AppBundle/Security/AuthTokenAuthenticator.php
2 <?php
3 namespace AppBundle\Security;
4
5 use Symfony\Component\HttpFoundation\Request;
6 use
7     Symfony\Component\Security\Core\Authentication\Token\PreAuthenticatedToken
8 use
9     Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
```


III. Amélioration de l'API REST

```
8 use
  Symfony\Component\Security\Core\Exception\AuthenticationException;
9 use
  Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
10 use
  Symfony\Component\Security\Core\Exception\BadCredentialsException;
11 use
  Symfony\Component\Security\Core\User\UserProviderInterface;
12 use
  Symfony\Component\Security\Http\Authentication\AuthenticationFailureHandlerInterface;
13 use
  Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;
14 use
  Symfony\Component\Security\Http\HttpUtils;
15
16 class AuthTokenAuthenticator implements
  SimplePreAuthenticatorInterface,
  AuthenticationFailureHandlerInterface
17 {
18     /**
19     * Durée de validité du token en secondes, 12 heures
20     */
21     const TOKEN_VALIDITY_DURATION = 12 * 3600;
22
23     protected $httpUtils;
24
25     public function __construct(HttpUtils $httpUtils)
26     {
27         $this->httpUtils = $httpUtils;
28     }
29
30     public function createToken(Request $request, $providerKey)
31     {
32
33         $targetUrl = '/auth-tokens';
34         // Si la requête est une création de token, aucune
35         // vérification n'est effectuée
36         if ($request->getMethod() === "POST" &&
37             $this->httpUtils->checkRequestPath($request,
38             $targetUrl)) {
39             return;
40         }
41
42         $authTokenHeader = $request->headers->get('X-Auth-Token');
43
44         if (!$authTokenHeader) {
45             throw new
46                 BadCredentialsException('X-Auth-Token header is required');
47         }
48
49         return new PreAuthenticatedToken(
50             'anon.',
```

III. Amélioration de l'API REST

```
47         $authTokenHeader,  
48         $providerKey  
49     );  
50 }  
51  
52 public function authenticateToken(TokenInterface $token,  
53     UserProviderInterface $userProvider, $providerKey)  
54 {  
55     if (!$userProvider instanceof AuthTokenUserProvider) {  
56         throw new \InvalidArgumentException(  
57             sprintf(  
58                 'The user provider must be an instance of AuthTokenUser  
59                 get_class($userProvider)  
60             )  
61         );  
62     }  
63     $authTokenHeader = $token->getCredentials();  
64     $authToken = $userProvider->getAuthToken($authTokenHeader);  
65  
66     if (!$authToken || !$this->isTokenValid($authToken)) {  
67         throw new  
68         BadCredentialsException('Invalid authentication token');  
69     }  
70     $user = $authToken->getUser();  
71     $pre = new PreAuthenticatedToken(  
72         $user,  
73         $authTokenHeader,  
74         $providerKey,  
75         $user->getRoles()  
76     );  
77  
78     // Nos utilisateurs n'ont pas de role particulier, on doit  
79     // donc forcer l'authentification du token  
80     $pre->setAuthenticated(true);  
81     return $pre;  
82 }  
83  
84 public function supportsToken(TokenInterface $token,  
85     $providerKey)  
86 {  
87     return $token instanceof PreAuthenticatedToken &&  
88         $token->getProviderKey() === $providerKey;  
89 }  
90 /**  
91 * Vérifie la validité du token
```

III. Amélioration de l'API REST

```
91  */
92  private function isValid($authToken)
93  {
94      return (time() -
95              $authToken->getCreatedAt()->getTimestamp()) <
96              self::TOKEN_VALIDITY_DURATION;
97  }
98
99  public function onAuthenticationFailure(Request $request,
100 AuthenticationException $exception)
101 {
102     // Si les données d'identification ne sont pas correctes,
103     // une exception est levée
104     throw $exception;
105 }
```

La configuration du service est classique :

```
1 # app/config/services.yml
2
3 # Learn more about services, parameters and containers at
4 # http://symfony.com/doc/current/book/service_container.html
5 parameters:
6 #     parameter_name: value
7
8 services:
9     auth_token_user_provider:
10         class: AppBundle\Security\AuthTokenUserProvider
11         arguments: ["@auth_token_repository", "@user_repository"]
12         public: false
13
14     auth_token_repository:
15         class: Doctrine\ORM\EntityManager
16         factory: ["@doctrine.orm.entity_manager", "getRepository"]
17         arguments: ["AppBundle:AuthToken"]
18
19     user_repository:
20         class: Doctrine\ORM\EntityManager
21         factory: ["@doctrine.orm.entity_manager", "getRepository"]
22         arguments: ["AppBundle:User"]
23
24     auth_token_authenticator:
25         class: AppBundle\Security\AuthTokenAuthenticator
26         arguments: ["@security.http_utils"]
27         public: false
```

Nous devons maintenant activer le pare-feu (firewall) de Symfony et le configurer avec notre

III. Amélioration de l'API REST

fournisseur d'utilisateurs et le listener que nous venons de créer.

```
1 # app/config/security.yml
2
3 # To get started with security, check out the documentation:
4 # http://symfony.com/doc/current/book/security.html
5 security:
6
7     #
8     http://symfony.com/doc/current/book/security.html#where-do-users-come-
9     providers:
10     auth_token_user_provider:
11         id: auth_token_user_provider
12
13     firewalls:
14     # disables authentication for assets and the profiler,
15     # adapt it according to your needs
16     dev:
17         pattern: ^/(_(profiler|wdt)|css|images|js)/
18         security: false
19
20     main:
21         pattern: ^/
22         stateless: true
23         simple_preauth:
24             authenticator: auth_token_authenticator
25             provider: auth_token_user_provider
26             anonymous: ~
27
28     encoders:
29     AppBundle\Entity\User:
30         algorithm: bcrypt
31         cost: 12
```



Vous pouvez remarquer que le pare-feu (firewall) est configuré en mode `stateless`. À chaque requête, l'identité de l'utilisateur est revérifiée. La session n'est jamais utilisée.

Maintenant, lorsque nous essayons de lister les lieux sans mettre l'entête d'authentification, une exception est levée :



FIGURE 12.4. – Requête Postman pour lister les lieux sans token d'authentification

III. Amélioration de l'API REST

```
1 {
2   "error": {
3     "code": 500,
4     "message": "Internal Server Error",
5     "exception": [
6       {
7         "message": "X-Auth-Token header is required",
8         "class":
9           "Symfony\\Component\\Security\\Core\\Exception\\BadCredentialsException",
10        "trace": [
11          "...",
12        ]
13      }
14    ]
15  }
```

Spoil! Les codes de statut et les messages renvoyés pour ce cas de figure ne sont pas conformes aux principes REST. Nous verrons dans ce chapitre comment corriger le tir.

Pour le moment, l'exception `BadCredentialsException`, avec le message `X-Auth-Token header is required`, confirme bien que la vérification du token est effectuée.

En rajoutant le token que nous avons généré plus tôt, la réponse contient bien la liste des lieux de notre application.

Avec Postman, il faut accéder à l'onglet `Headers` en dessous de l'URL pour ajouter des entêtes à notre requête.



FIGURE 12.5. – Requête Postman pour lister les lieux avec un token d'authentification

```
1 [
2   {
3     "id": 1,
4     "name": "Tour Eiffel",
5     "address": "5 Avenue Anatole France, 75007 Paris",
6     "prices": [
7       {
8         "id": 1,
9         "type": "less_than_12",
10        "value": 5.75
11      }
12    ]
13  }
```

```
11     }
12   ],
13   "themes": [
14     {
15       "id": 1,
16       "name": "architecture",
17       "value": 7
18     },
19     {
20       "id": 2,
21       "name": "history",
22       "value": 6
23     }
24   ]
25 },
26 // ...
27 ]
```

Notre API est maintenant sécurisée!

Par contre, la gestion des exceptions n'est pas encore très élaborée. En plus, vous l'avez peut-être déjà remarqué mais le format des messages d'erreur n'est pas uniforme. Lorsque le formulaire est invalide ou une exception est levée, les réponses renvoyées ne sont pas identiques, un client de l'API aura donc du mal à gérer les réponses en erreur.

12.2. Gestion des erreurs avec FOSRestBundle

Le bundle *FOSRestBundle* met à notre disposition un ensemble de composants pour gérer différents aspects d'une API. Et vous vous en doutez donc, il existe un listener pour gérer de manière simple et efficace les exceptions.

À l'état actuel, les exceptions sont gérées par le listener du bundle *Twig*. La première configuration à effectuer est de le remplacer par l'exception listener de *FOSRestBundle*. Pour cela, il suffit de rajouter une ligne dans la configuration du bundle.

```
1 # app/config/config.yml
2
3 # ...
4 fos_rest:
5   routing_loader:
6     include_format: false
7   # ...
8   exception:
9     enabled: true
```

III. Amélioration de l'API REST

En activant ce composant, la gestion des exceptions avec *Twig* est automatiquement désactivée. Rien qu'avec cette configuration, nous pouvons voir un changement dans la réponse lorsque l'entête `X-Auth-Token` n'est pas renseignée.

```
1 {
2   "code": 500,
3   "message": "X-Auth-Token header is required"
4 }
```



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 12.6. – Code de statut de l'erreur interne

Lorsque le token renseigné n'est pas valide, nous obtenons comme réponse :

```
1 {
2   "code": 500,
3   "message": "Invalid authentication token"
4 }
```

Les messages correspondent à ceux que nous avons défini dans les exceptions parce que l'application est en mode développement. En production, ces messages sont remplacés par `Internal Server Error`. Pour s'en rendre compte, il suffit de lancer la même requête avec comme URL `rest-api.local/app.php/places` pour forcer la configuration en production.

```
1 {
2   "code": 500,
3   "message": "Internal Server Error"
4 }
```

Il peut arriver que nous voulions conserver les messages des exceptions même en production. Pour ce faire, il suffit de rajouter dans la configuration un système d'autorisation des exceptions concernées.

```
1 # app/config/config.yml
2
3 # ...
4 fos_rest:
```

III. Amélioration de l'API REST

```
5     routing_loader:
6         include_format: false
7     # ...
8     exception:
9         enabled: true
10        messages:
11
12                'Symfony\Component\Security\Core\Exception\BadCredentialsException'
13                true
```

Le tableau message contient comme clés les noms des exceptions à autoriser et la valeur vaut **true**.

En re-testant, la requête sur l'URL `rest-api.local/app.php/places` (n'oubliez pas de vider le cache avant de tester), le message est bien affiché :

```
1 {
2     "code": 500,
3     "message": "Invalid authentication token"
4 }
```

Nous venons de franchir un premier pas.

Mais comme nous l'avons déjà vu, le code **500** ne doit être utilisé que pour les erreurs internes du serveur. Pour le cas d'une authentification qui a échoué, le protocole HTTP propose un code bien spécifique - **401 Unauthorized** - qui dit qu'une authentification est nécessaire pour accéder à notre ressource.

Encore une fois, *FOSRestBundle* propose un système très simple. À l'instar du tableau `messages`, nous pouvons rajouter un tableau `codes` avec comme clés les exceptions et comme valeur les codes de statut associés.

Nous aurons donc comme configuration finale :

```
1 # app/config/config.yml
2
3 # ...
4 fos_rest:
5     routing_loader:
6         include_format: false
7     # ...
8     exception:
9         enabled: true
10        messages:
11
12                'Symfony\Component\Security\Core\Exception\BadCredentialsException'
13                true
```


III. Amélioration de l'API REST

```
12     codes :
13         'Symfony\Component\Security\Core\Exception\BadCredentialsException'
14         401
```

Encore une fois ce bundle, nous facilite grandement le travail et réduit considérablement le temps de développement.

Lorsque nous re-exécutons notre requête :



FIGURE 12.7. – Requête Postman sans token d'autorisation

La réponse contient le bon message et le code de statut est aussi correct :



FIGURE 12.8. – Code de statut de la réponse non autorisée

```
1 {
2   "code": 401,
3   "message": "X-Auth-Token header is required"
4 }
```



L'attribut `code` dans la réponse est créé par *FOSRestBundle* par soucis de clarté. La contrainte REST, elle, exige juste que le code HTTP de la réponse soit conforme.

Vu que le bundle est conçu pour interagir avec Symfony, toutes les exceptions du framework qui implémentent l'interface `Symfony\Component\HttpKernel\Exception\HttpExceptionInterface` peuvent être traitées automatiquement.

Si par exemple, nous utilisons l'exception `NotFoundHttpException`, le code de statut devient automatiquement `404`. En général, il est aussi utile d'autoriser tous les messages des exceptions de type `HttpException` pour faciliter la gestion des cas d'erreurs.

La configuration du bundle devient maintenant :

III. Amélioration de l'API REST

```
1 # app/config/config.yml
2 # ...
3
4 fos_rest:
5     routing_loader:
6         include_format: false
7     # ...
8     exception:
9         enabled: true
10        messages:
11            'Symfony\Component\HttpKernel\Exception\HttpException'
12                : true
13
14                'Symfony\Component\Security\Core\Exception\BadCredentialsException'
15                    : true
16
17        codes:
18            'Symfony\Component\Security\Core\Exception\BadCredentialsException'
19                : 401
```

Toutes les occurrences de `return \FOS\RestBundle\View\View::create(['message' => 'Place not found'], Response::HTTP_NOT_FOUND);` peuvent être remplacées par `throw new \Symfony\Component\HttpKernel\Exception\NotFoundHttpException('Place not found');`.

Et de même `return \FOS\RestBundle\View\View::create(['message' => 'User not found'], Response::HTTP_NOT_FOUND);` peut être remplacé par `throw new \Symfony\Component\HttpKernel\Exception\NotFoundHttpException('User not found');`.

Les réponses restent identiques mais les efforts fournis pour les produire sont réduits.



FIGURE 12.9. – Récupération d'un utilisateur inexistant avec Postman



FIGURE 12.10. – Code de statut de la réponse

12.3. 401 ou 403, quand faut-il utiliser ces codes de statut ?

Les codes de statuts `401` et `403` sont souvent source de confusion. Ils sont tous les deux utilisés pour gérer les informations liées à la sécurité.

Le code `401` est utilisé pour signaler que **la requête nécessite une authentification**. Avec notre système de sécurité actuel, nous exigeons que toutes les requêtes - à part celle de création de token - aient une entête `X-Auth-Token` valide. Donc si une requête ne remplit pas ces conditions, elle est considérée comme non autorisée d'où le code de statut `401`. En général, c'est depuis [le pare-feu de Symfony](#) que ce code de statut doit être renvoyé.

Par contre, le code `403` est utilisé pour signaler qu'**une requête est interdite**. La différence réside dans le fait que pour qualifier une requête comme étant interdite, nous devons d'abord identifier le client de l'API à l'origine de celle-ci. Le code `403` doit donc être utilisé si le client de l'API est bien identifié via l'entête `X-Auth-Token` mais ne dispose pas des privilèges nécessaires pour effectuer l'opération qu'il souhaite.

Si par exemple, nous disposons d'un appel API réservé uniquement aux administrateurs, si un client simple essaye d'effectuer cette requête, nous devons renvoyer un code de statut `403`. Ce code de statut peut être utilisé au niveau des ACLs (Access Control List) ou [des voteurs de Symfony](#).

i

En résumé, le code de statut `401` permet de signaler au client qu'il doit décliner son identité et le code de statut `403` permet de notifier à un client déjà identifié qu'il ne dispose pas de droits suffisants.

12.4. Suppression d'un token ou la déconnexion

Pour en finir avec la partie sécurisation, il ne reste plus qu'à rajouter une méthode pour se déconnecter de notre API. La déconnexion consiste juste à la suppression du token d'authentification. Par contre, une petite précaution va s'imposer. Pour traiter la suppression d'un token, il faudra juste nous assurer que l'utilisateur veut supprimer son propre token et pas celui d'un autre. À part cette modification, tous les autres mécanismes déjà vus rentrent en jeu.

Pour supprimer la ressource, il faudra donc une requête `DELETE` sur la ressource `rest-api.local/auth-tokens/{id}`. La réponse en cas de succès doit être vide avec comme code de statut : `204 No Content`.

```
1 # src/AppBundle/Controller/AuthTokenController.php
2 <?php
3 namespace AppBundle\Controller;
4
5 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
6 use Symfony\Component\HttpFoundation\JsonResponse;
7 use Symfony\Component\HttpFoundation\Response;
```

```

8 use Symfony\Component\HttpFoundation\Request;
9 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
  toutes les annotations
10 use AppBundle\Form\Type\CredentialsType;
11 use AppBundle\Entity\AuthToken;
12 use AppBundle\Entity\Credentials;
13
14 class AuthTokenController extends Controller
15 {
16     //...
17
18     /**
19      * @Rest\View(statusCode=Response::HTTP_NO_CONTENT)
20      * @Rest>Delete("/auth-tokens/{id}")
21      */
22     public function removeAuthTokenAction(Request $request)
23     {
24         $em = $this->get('doctrine.orm.entity_manager');
25         $authToken = $em->getRepository('AppBundle:AuthToken')
26             ->find($request->get('id'));
27         /* @var $authToken AuthToken */
28
29         $connectedUser = $this-
30             >get('security.token_storage')->getToken()->getUser();
31
32         if ($authToken && $authToken->getUser()->getId() ===
33             $connectedUser->getId()) {
34             $em->remove($authToken);
35             $em->flush();
36         } else {
37             throw new
38                 \Symfony\Component\HttpKernel\Exception\BadRequestHttpException
39
40     }
41 }
42 // ...
43 }

```



<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 12.11. – Requête de suppression d'un token avec Postman

Si tout se passe bien, la réponse lors d'une suppression est vide avec comme statut 204. En cas d'erreur une réponse 400 est renvoyée au client.

III. Amélioration de l'API REST

```
1 {  
2   "code": 400,  
3   "message": "Bad Request"  
4 }
```

Notre fameux tableau récapitulatif s'enrichit d'un nouveau code de statut et listing des entêtes HTTP utilisables :

| Opération souhaitée | Verbe HTTP |
|--------------------------------------|------------|
| Lecture | GET |
| Création | POST |
| Suppression | DELETE |
| Modification complète (remplacement) | PUT |
| Modification partielle | PATCH |

| Code statut | Signification |
|-------------|---|
| 200 | Tout s'est bien passé et la réponse a du contenu |
| 204 | Tout s'est bien passé mais la réponse est vide |
| 400 | Les données envoyées par le client sont invalides |
| 401 | Une authentification est nécessaire pour accéder à la ressource |
| 403 | Le client authentifié ne dispose pas des droits nécessaires |
| 404 | La ressource demandée n'existe pas |
| 500 | Une erreur interne a eu lieu sur le serveur |

| Entête HTTP | Contenu |
|--------------|--|
| Accept | Un ou plusieurs média type souhaités par le client |
| Content-Type | Le média type de la réponse ou de la requête |
| X-Auth-Token | Token d'authentification |

Un client de l'API peut maintenant se connecter et se déconnecter et toutes ses requêtes nécessitent une authentification. Notre API vient d'être sécurisée ! La durée de validité du token et les critères de validation de celui-ci sont purement arbitraires. Vous pouvez donc les changer à votre guise.

Pour les besoins de ce cours les requêtes se font via HTTP mais il faudra garder en tête que la

III. Amélioration de l'API REST

meilleure des sécurités ne vaut rien si le protocole utilisé n'est pas sécurisé. Donc dans une API en production, il faut **systématiquement** utiliser le **HTTPS**.

13. Créer une ressource avec des relations

Revenons un peu sur les relations entre les ressources.

À la création d'un lieu, nous ne pouvons pas renseigner les tarifs. Nous sommes donc obligés de créer d'abord un lieu avant de rajouter ses tarifs.

Même si ce fonctionnement pourrait convenir dans certains cas, il peut aussi s'avérer utile de créer un lieu et de lui associer des tarifs avec un seul appel API. On pourra ainsi optimiser l'API et réduire les interactions entre le client et le serveur.

Nous allons donc voir dans cette partie comment arriver à un tel résultat avec Symfony.

13.1. Rappel de l'existant

Jusqu'à présent pour créer un lieu, il fallait juste renseigner le nom et l'adresse. Le payload pour la création d'un lieu ressemblait donc à :

```
1 {
2     "name": "Disneyland Paris",
3     "address": "77777 Marne-la-Vallée"
4 }
```

En réalité, pour supporter la création d'un lieu avec ses tarifs, les contraintes de REST ne rentrent pas en jeu. Nous pouvons adapter librement le payload afin de rajouter toutes les informations nécessaires pour supporter la création d'un lieu avec ses tarifs avec un seul appel.

13.2. Création d'un lieu avec des tarifs

13.2.1. Un peu de conception

Vu que nous avons déjà une méthode pour créer des tarifs, nous allons utiliser le même payload pour la création d'un lieu pour garder une API cohérente. Le payload existant doit être maintenant :

```
1 {
2     "name": "Disneyland Paris",
3     "address": "77777 Marne-la-Vallée",
```

```
4     "prices": [  
5         {  
6             "type": "for_all",  
7             "value": 10.0  
8         },  
9         {  
10            "type": "less_than_12",  
11            "value": 5.75  
12        }  
13    ]  
14 }
```

L'attribut `prices` est un tableau qui contiendra la liste des prix que nous voulons rajouter à la création du lieu.

Les tarifs resteront optionnels ce qui nous permettra de créer des lieux avec ou sans. Nous allons sans plus attendre appliquer ces modifications à l'appel existant.

13.2.2. Implémentation

13.2.2.1. Mise à jour du formulaire

La méthode pour créer un lieu reste inchangée. Nous devons juste changer le formulaire des lieux et le traitement associé.

```
1 <?php  
2 # src/AppBundle/Form/Type/PlaceType.php  
3  
4 namespace AppBundle\Form\Type;  
5  
6 use Symfony\Component\Form\AbstractType;  
7 use Symfony\Component\Form\Extension\Core\Type\CollectionType;  
8 use Symfony\Component\Form\FormBuilderInterface;  
9 use Symfony\Component\OptionsResolver\OptionsResolver;  
10  
11 class PlaceType extends AbstractType  
12 {  
13     public function buildForm(FormBuilderInterface $builder, array  
14         $options)  
15     {  
16         $builder->add('name');  
17         $builder->add('address');  
18         $builder->add('prices', CollectionType::class, [  
19             'entry_type' => PriceType::class,  
20             'allow_add' => true,  
21             'error_bubbling' => false,  
22         ]);  
23     }  
24 }
```



```
22     }
23
24     public function configureOptions(OptionsResolver $resolver)
25     {
26         $resolver->setDefaults([
27             'data_class' => 'AppBundle\Entity\Place',
28             'csrf_protection' => false
29         ]);
30     }
31 }
```

La configuration du formulaire est typique des formulaires Symfony avec une collection. [La documentation officielle](#) aborde le sujet d'une manière plus complète.

Les règles de validation pour les thèmes existent déjà. Pour les utiliser, nous devons modifier la validation de l'entité *Place* en rajoutant la règle *Valid*. Avec cette annotation, nous disons à Symfony de valider l'attribut *prices* en utilisant les contraintes de validation de l'entité *Price*.

```
1 # src/AppBundle/Resources/config/validation.yml
2 AppBundle\Entity\Place:
3     constraints:
4         -
5             Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity:
6                 name
7     properties:
8         name:
9             - NotBlank: ~
10            - Type: string
11        address:
12            - NotBlank: ~
13            - Type: string
14        prices:
15            - Valid: ~
```

Notez qu'il n'y a pas d'assertions de type *NotBlank* puisque l'attribut *prices* est optionnel.

13.2.2.2. Traitement du formulaire

Avec les modifications que nous venons d'apporter, nous pouvons déjà tester la création d'un lieu avec des prix. Mais avant de le faire, nous allons rapidement adapter le contrôleur pour gérer la sauvegarde des prix.

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
```

```

3
4 namespace AppBundle\Controller;
5
6 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
7 use Symfony\Component\HttpFoundation\Request;
8 use Symfony\Component\HttpFoundation\JsonResponse;
9 use Symfony\Component\HttpFoundation\Response;
10 use FOS\RestBundle\Controller\Annotations as Rest; // alias pour
    toutes les annotations
11 use AppBundle\Form\Type\PlaceType;
12 use AppBundle\Entity\Place;
13
14 class PlaceController extends Controller
15 {
16
17     // ...
18
19     /**
20      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"place"})
21      * @Rest\Post("/places")
22      */
23     public function postPlacesAction(Request $request)
24     {
25         $place = new Place();
26         $form = $this->createForm(PlaceType::class, $place);
27
28         $form->submit($request->request->all());
29
30         if ($form->isValid()) {
31             $em = $this->get('doctrine.orm.entity_manager');
32             foreach ($place->getPrices() as $price) {
33                 $price->setPlace($place);
34                 $em->persist($price);
35             }
36             $em->persist($place);
37             $em->flush();
38             return $place;
39         } else {
40             return $form;
41         }
42     }
43
44     // ...

```



Comme vous l'avez sûrement remarqué, toute la logique de notre API est regroupée dans les contrôleurs. Ceci n'est pas une bonne pratique et l'utilisation d'un service dédié est vivement conseillée pour une application destinée à une mise en production.

III. Amélioration de l'API REST

L'entité *Place* a été légèrement modifiée. L'attribut `prices` est maintenant initialisé avec une collection vide.

```
1 <?php
2 # src/AppBundle/Entity/Place.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity()
10  * @ORM\Table(name="places",
11  *           uniqueConstraints={@ORM\UniqueConstraint(name="places_name_unique", col
12  * )
13  */
14 class Place
15 {
16     // ...
17
18     /**
19      * @ORM\OneToMany(targetEntity="Price", mappedBy="place")
20      * @var Price[]
21      */
22     protected $prices;
23
24     public function __construct()
25     {
26         $this->prices = new ArrayCollection();
27         // ...
28     }
29
30     // ...
```

En testant une création d'un lieu avec des prix :

```
1 {
2   "name": "Musée du Louvre",
3   "address": "799, rue de Rivoli, 75001 Paris",
4   "prices": [
5     {
6       "type": "less_than_12",
7       "value": 6
8     },
9     {
10      "type": "for_all",
11      "value": 15
```

III. Amélioration de l'API REST

```
12   }
13 ]
14 }
```

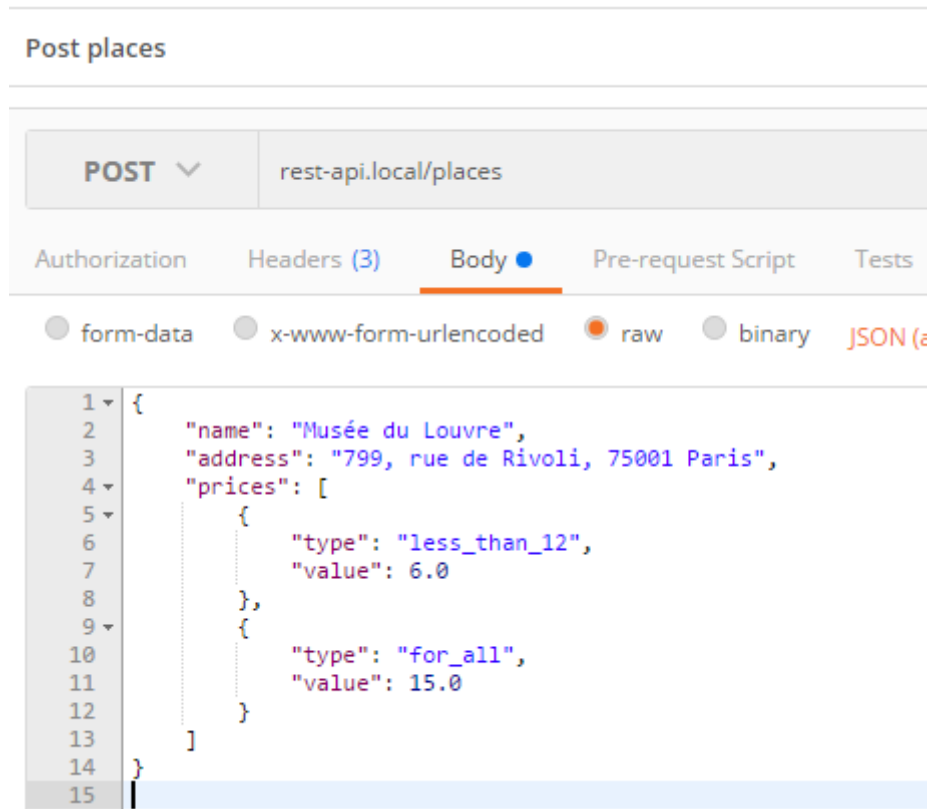


FIGURE 13.1. – Requête de création d'un lieu avec des prix

La réponse est identique à ce que nous avons déjà eu mais les prix sont enregistrés en même temps.

```
1 {
2   "id": 9,
3   "name": "Musée du Louvre",
4   "address": "799, rue de Rivoli, 75001 Paris",
5   "prices": [
6     {
7       "id": 6,
8       "type": "less_than_12",
9       "value": 6
10    },
11    {
12      "id": 7,
13      "type": "for_all",
14      "value": 15
15    }
16  ],
```

```
17     "themes": []
18 }
```

Nous pouvons maintenant créer un lieu tout en rajoutant des prix et le principe peut même être élargi pour les thèmes des lieux et les préférences des utilisateurs.

13.3. Bonus : Une validation plus stricte

13.3.1. Création d'un lieu avec deux prix de même type

Si nous essayons de créer un lieu avec des prix du même type, nous obtenons une erreur interne car il y a une contrainte d'unicité sur l'identifiant du lieu et le type du produit.

```
1 <?php
2 # src/AppBundle/Entity/Price.php
3
4 namespace AppBundle\Entity;
5
6 use Doctrine\ORM\Mapping as ORM;
7
8 /**
9  * @ORM\Entity()
10  * @ORM\Table(name="prices",
11  *           uniqueConstraints={@ORM\UniqueConstraint(name="prices_type_place_unique
12  * )
13  */
14 class Price
15 {
16     // ...
17 }
```

Pour s'en convaincre, il suffit d'essayer de créer un nouveau lieu avec comme payload :

```
1 {
2     "name": "Arc de Triomphe",
3     "address": " Place Charles de Gaulle, 75008 Paris",
4     "prices": [
5         {
6             "type": "less_than_12",
7             "value": 0.0
8         },
9         {
10            "type": "less_than_12",
```

III. Amélioration de l'API REST

```
11         "value": 0.0
12     }
13 ]
14 }
```

La réponse est sans appel :

```
1 {
2     "code": 500,
3     "message":
4         "An exception occurred while executing 'INSERT INTO prices (type, value,
```

Pour corriger le problème, nous allons créer une règle de validation personnalisée.

13.3.2. Validation personnalisée avec Symfony

13.3.2.1. Création de la contrainte

La contrainte est la partie la plus simple à implémenter. Il suffit d'une classe pour la nommer et d'un message en cas d'erreur.

```
1 <?php
2 # src/AppBundle/Form/Validator/Constraint/PriceTypeUnique.php
3
4 namespace AppBundle\Form\Validator\Constraint;
5
6 use Symfony\Component\Validator\Constraint;
7
8 /**
9  * @Annotation
10  */
11 class PriceTypeUnique extends Constraint
12 {
13     public $message =
14         'A place cannot contain prices with same type';
15 }
```

13.3.2.2. Création du validateur

Une fois que nous avons une nouvelle contrainte, il reste à créer un validateur pour gérer cette contrainte.

```
1 <?php
2 #
3     src/AppBundle/Form/Validator/Constraint/PriceTypeUniqueValidator.php
4 namespace AppBundle\Form\Validator\Constraint;
5
6 use Symfony\Component\Validator\Constraint;
7 use Symfony\Component\Validator\ConstraintValidator;
8
9 class PriceTypeUniqueValidator extends ConstraintValidator
10 {
11     public function validate($prices, Constraint $constraint)
12     {
13         if (!$prices instanceof
14             \Doctrine\Common\Collections\ArrayCollection) {
15             return;
16         }
17
18         $pricesType = [];
19
20         foreach ($prices as $price) {
21             if (in_array($price->getType(), $pricesType)) {
22                 $this-
23                     >context->buildViolation($constraint->message)
24                     ->addViolation();
25                 return; // Si il y a un doublon, on arrête la
26                     recherche
27             } else {
28                 // Sauvegarde des types de prix déjà présents
29                 $pricesType[] = $price->getType();
30             }
31         }
32     }
33 }
```

Le nom choisi n'est pas un hasard. Vu que la contrainte s'appelle *PriceTypeUnique*, le validateur a été nommé *PriceTypeUniqueValidator* afin d'utiliser les conventions de nommage de Symfony. Ainsi notre contrainte est validée en utilisant le validateur que nous venons de créer.



Ce comportement par défaut peut être modifié en étendant la méthode `validatedBy` de la contrainte. La [documentation officielle](#) de Symfony apporte plus d'informations à ce sujet.

Pour utiliser notre nouvelle contrainte, nous allons modifier les règles de validation qui s'applique à un lieu :

III. Amélioration de l'API REST

```
1 # src/AppBundle/Resources/config/validation.yml
2 AppBundle\Entity\Place:
3   constraints:
4     -
5       Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity:
6         name
7   properties:
8     name:
9       - NotBlank: ~
10      - Type: string
11     address:
12       - NotBlank: ~
13       - Type: string
14     prices:
15       - Valid: ~
16       - AppBundle\Form\Validator\Constraint\PriceTypeUnique:
17         ~
```

En testant à nouveau la création d'un lieu avec deux prix du même type, nous obtenons une belle erreur de validation avec un message clair.

```
1 {
2   "code": 400,
3   "message": "Validation Failed",
4   "errors": {
5     "children": {
6       "name": [],
7       "address": [],
8       "prices": {
9         "errors": [
10          "A place cannot contain prices with same type"
11        ],
12        "children": [
13          {
14            "children": {
15              "type": [],
16              "value": []
17            }
18          },
19          {
20            "children": {
21              "type": [],
22              "value": []
23            }
24          }
25        ]
26      }
27    }
28  }
```


III. Amélioration de l'API REST

```
27     }  
28   }  
29 }
```

Comme vous avez pu le remarquer, la création d'une ressource en relation avec d'autres ressources ne relève pas trop de REST mais plutôt de la gestion des formulaires avec Symfony.

Ainsi, les connaissances que vous avez déjà pu acquérir pour la gestion des formulaires dans Symfony peuvent être exploitées pour mettre en place ces fonctionnalités.

Prendre en compte ce genre de détails d'implémentation permet de réduire le nombre d'appels API et donc améliorer les performances des applications qui doivent l'exploiter et l'expérience utilisateur par la même occasion.

14. Quand utiliser les query strings ?

Jusqu'à présent les query strings ou paramètres d'URL ont été recalés dans tous les choix de conception que nous avons déjà faits.

Ces composants à part entière du protocole HTTP peuvent être exploités dans une API REST pour atteindre différents objectifs.

Dans cette partie, nous allons aborder quelques cas pratiques où les query strings peuvent être utilisés.

Tout au long de cette partie, le terme *query strings* sera utilisé pour désigner les paramètres d'URL.

14.1. Pourquoi utiliser les query strings ?

Le premier cas d'usage qui est assez courant lorsque nous utilisons ou nous développons une API est la pagination ou le filtrage des réponses que nous obtenons.

Nous pouvons actuellement lister tous les lieux ou tous les utilisateurs de notre application. Cette réponse peut rapidement poser des problèmes de performance si ces listes grossissent dans le temps.

?

Comment alors récupérer notre liste de lieux tout en réduisant/filtrant cette liste alors que nous avons un seul appel permettant de lister les lieux de notre application : `GET rest-api.local/places?`

La liste de lieux est une ressource avec un identifiant `places`. Pour récupérer cette même liste tout en conservant son identifiant nous ne pouvons pas modifier l'URL.

Par contre, les query string nous permettent de pallier à ce genre de problèmes.

The query component contains non-hierarchical data that, along with data in the path component (Section 3.3), serves **to identify a resource within the scope of the URI's scheme** and naming authority (if any).

Source : [RFC 3986](#) ↗

Donc au sein d'une même URL (ici `rest-api.local/places`), nous pouvons rajouter des query strings afin d'obtenir des réponses différentes mais qui représentent toutes une liste de lieux.

14.2. Gestion des query strings avec FOSRestBundle

Avant d'aborder les cas pratiques, nous allons commencer par voir comment *FOSRestBundle* nous permet de définir les query strings.

Le framework Symfony supporte de base les query strings mais *FOSRestBundle* rajoute beaucoup de fonctionnalités comme :

- définir des règles de validation pour ce query string ;
- définir une valeur par défaut ;
- et beaucoup d'autres fonctionnalités.

14.2.1. L'annotation QueryParam

Pour accéder à toutes ces fonctionnalités, il suffit d'utiliser une annotation `FOS\RestBundle\Controller\Annotations\QueryParam` sur le ou les actions de nos contrôleurs.



Il est aussi possible d'utiliser cette annotation sur un contrôleur mais nous ne parlerons pas de ce cas d'usage.

```
1 <?php
2 /**
3  * @QueryParam(
4  *   name="",
5  *   key=null,
6  *   requirements="",
7  *   incompatibles={},
8  *   default=null,
9  *   description="",
10 *   strict=false,
11 *   array=false,
12 *   nullable=false
13 * )
14 */
```

Nous aborderons les cas d'utilisation des attributs de cette annotation dans la suite.

14.2.2. Le listener

Pour dire à *FOSRestBundle* de traiter cette annotation, nous devons activer un listener dédié appelé le *Param Fetcher Listener*. Pour ce faire, nous allons modifier le fichier de configuration :

```
1 # app/config/config.yml
2
3 # ...
4
5 fos_rest:
6   routing_loader:
7     include_format: false
8   view:
9     view_response_listener: true
10  formats:
11    json: true
12    xml: true
13  format_listener:
14    rules:
15      - { path: '^/', priorities: ['json', 'xml'],
16          fallback_format: 'json', prefer_extension: false }
17  body_listener:
18    enabled: true
19  param_fetcher_listener:
20    enabled: true
21 # ...
```

Maintenant que le listener est activé, nous pouvons passer aux choses sérieuses.

14.3. Paginer et Trier les réponses

14.3.1. Paginer la liste de lieux

Commençons par mettre en place une pagination pour la liste des lieux. Pour obtenir cette pagination, nous allons utiliser un principe simple.

Deux query strings vont permettre de choisir l'index du premier résultat souhaité (`offset`) et le nombre de résultats souhaités (`limit`).

Ces deux paramètres sont facultatifs mais doivent obligatoirement être des entiers positifs.

Pour implémenter ce fonctionnement, il suffit de rajouter deux annotations `QueryParam` dans l'action qui liste les lieux.

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5
6 // ...
7 use FOS\RestBundle\Controller\Annotations\QueryParam;
```

```

8 // ...
9
10 class PlaceController extends Controller
11 {
12
13     /**
14     * @Rest\View(serializerGroups={"place"})
15     * @Rest\Get("/places")
16     * @QueryParam(name="offset", requirements="\d+", default="", description="")
17     * @QueryParam(name="limit", requirements="\d+", default="", description="")
18     */
19     public function getPlacesAction(Request $request)
20     {
21         $places = $this->get('doctrine.orm.entity_manager')
22                 ->getRepository('AppBundle:Place')
23                 ->findAll();
24         /* @var $places Place[] */
25
26         return $places;
27     }
28 // ...
29 }

```

Avec l'attribut `requirements`, nous utilisons une expression régulière pour valider les paramètres. Si les données ne sont pas valides alors le paramètre vaudra sa valeur par défaut (une chaîne vide pour notre cas). Si les paramètres ne sont pas renseignés, ils seront aussi vides. Il faut aussi noter que nous pouvons utiliser n'importe quelle valeur par défaut. En effet, elle n'est pas validée par FOSRestBundle. C'est d'ailleurs pour cette raison que nous pouvons mettre dans notre exemple une chaîne vide comme valeur par défaut alors que notre expression régulière ne valide que les entiers.



L'expression régulière utilisée dans `requirements` est traité en rajoutant automatiquement un pattern du type `#!notre_regex$#xsu`. En mettant, `\d+` nous validons donc avec `#!\d+$#xsu`. Vous pouvez consulter [la documentation de PHP](#) pour voir l'utilité des options `x` (ignorer les caractères d'espace), `s` (pour utiliser `.` comme métacaractère générique) et `u` (le masque et la chaîne d'entrée sont traitées comme des chaînes UTF-8.).

Pour les traiter, nous avons plusieurs choix. Nous pouvons utiliser un attribut de l'objet *Request* appelé `paramFetcher` que le *Param Fetcher Listener* crée automatiquement. Ou encore, nous pouvons ajouter un paramètre à notre action qui doit être du type `FOS\RestBundle\Request\ParamFetcher`.

Avec la cette dernière méthode, que nous allons utiliser, le *Param Fetcher Listener* injecte automatiquement le param fetcher à notre place.

L'objet ainsi obtenu permet d'accéder aux différents query strings que nous avons déclarés.

III. Amélioration de l'API REST

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5
6 // ...
7 use FOS\RestBundle\Controller\Annotations\QueryParam;
8 use FOS\RestBundle\Request\ParamFetcher;
9 // ...
10
11 class PlaceController extends Controller
12 {
13
14     /**
15      * @Rest\View(serializerGroups={"place"})
16      * @Rest\Get("/places")
17      * @QueryParam(name="offset", requirements="\d+", default="", description="")
18      * @QueryParam(name="limit", requirements="\d+", default="", description="")
19      */
20     public function getPlacesAction(Request $request, ParamFetcher
        $paramFetcher)
21     {
22         $offset = $paramFetcher->get('offset');
23         $limit = $paramFetcher->get('limit');
24
25         $places = $this->get('doctrine.orm.entity_manager')
26             ->getRepository('AppBundle:Place')
27             ->findAll();
28         /* @var $places Place[] */
29
30         return $places;
31     }
32 // ...
33 }
```

Avec le param fetcher, nous pouvons récupérer nos paramètres et les traiter à notre convenance. Pour gérer la pagination avec Doctrine, nous pouvons utiliser le query builder avec les paramètres `offset` et `limit`.

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5
6 // ...
7 use FOS\RestBundle\Controller\Annotations\QueryParam;
8 use FOS\RestBundle\Request\ParamFetcher;
```

III. Amélioration de l'API REST

```
9 // ...
10
11 class PlaceController extends Controller
12 {
13
14     /**
15     * @Rest\View(serializerGroups={"place"})
16     * @Rest\Get("/places")
17     * @QueryParam(name="offset", requirements="\d+", default="", description="")
18     * @QueryParam(name="limit", requirements="\d+", default="", description="")
19     */
20     public function getPlacesAction(Request $request, ParamFetcher
21         $paramFetcher)
22     {
23         $offset = $paramFetcher->get('offset');
24         $limit = $paramFetcher->get('limit');
25
26         $qb = $this->
27             >get('doctrine.orm.entity_manager')->createQueryBuilder();
28         $qb->select('p')
29             ->from('AppBundle:Place', 'p');
30
31         if ($offset != "") {
32             $qb->setFirstResult($offset);
33         }
34
35         if ($limit != "") {
36             $qb->setMaxResults($limit);
37         }
38
39         $places = $qb->getQuery()->getResult();
40
41         return $places;
42     }
43 }
44 // ...
45 }
```

Nous pouvons maintenant tester plusieurs appels API :

- GET `rest-api.local/places?limit=5` permet de lister cinq lieux ;
- GET `rest-api.local/places?offset=3` permet de lister tous les lieux en omettant les trois premiers lieux ;
- GET `rest-api.local/places?offset=1&limit=2` permet de lister deux lieux en omettant le premier lieu dans l'application.

En testant le dernier exemple avec Postman, nous avons :

<http://zestedesavoir.com/media/galleries/3183/>

FIGURE 14.1. – Récupération des lieux avec une pagination

```
1 [
2   {
3     "id": 2,
4     "name": "Mont-Saint-Michel",
5     "address": "50170 Le Mont-Saint-Michel",
6     "prices": [],
7     "themes": [
8       {
9         "id": 3,
10        "name": "history",
11        "value": 3
12      },
13      {
14        "id": 4,
15        "name": "art",
16        "value": 7
17      }
18    ]
19  },
20  {
21    "id": 4,
22    "name": "Disneyland Paris",
23    "address": "77777 Marne-la-Vallée",
24    "prices": [],
25    "themes": []
26  }
27 ]
```

14.3.2. Trier la liste des lieux

Pour pratiquer, nous allons rajouter un paramètre pour trier les lieux selon leur nom.

Le paramètre s'appellera `sort` et pourra avoir deux valeurs : `asc` pour l'ordre croissant et `desc` pour l'ordre décroissant. La valeur par défaut sera `null`.

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
```


III. Amélioration de l'API REST

```
3
4 namespace AppBundle\Controller;
5
6 // ...
7 use FOS\RestBundle\Controller\Annotations\QueryParam;
8 // ...
9
10 class PlaceController extends Controller
11 {
12
13     /**
14      * @Rest\View(serializerGroups={"place"})
15      * @Rest\Get("/places")
16      * @QueryParam(name="offset", requirements="\d+", default="", description="")
17      * @QueryParam(name="limit", requirements="\d+", default="", description="")
18      * @QueryParam(name="sort", requirements="(asc|desc)", nullable=true, desc
19      */
20     public function getPlacesAction(Request $request, ParamFetcher
        $paramFetcher)
21     {
22         $offset = $paramFetcher->get('offset');
23         $limit = $paramFetcher->get('limit');
24         $sort = $paramFetcher->get('sort');
25
26         $qb = $this-
            >get('doctrine.orm.entity_manager')->createQueryBuilder();
27         $qb->select('p')
28             ->from('AppBundle:Place', 'p');
29
30         if ($offset != "") {
31             $qb->setFirstResult($offset);
32         }
33
34         if ($limit != "") {
35             $qb->setMaxResults($limit);
36         }
37
38         if (in_array($sort, ['asc', 'desc'])) {
39             $qb->orderBy('p.name', $sort);
40         }
41
42
43         $places = $qb->getQuery()->getResult();
44
45         return $places;
46     }
47 // ...
48 }
```

La seule différence avec les deux autres query strings est que pour avoir une valeur par défaut à

III. Amélioration de l'API REST

`null`, nous utilisons l'attribut `nullable`.

En testant l'appel précédant avec en plus un tri des noms par ordre décroissant :



http://zestedesavoir.com/media/galleries/3183/

FIGURE 14.2. – Récupération des lieux avec une pagination et un tri par ordre décroissant de nom

La réponse change en :

```
1  [
2  {
3    "id": 6,
4    "name": "test",
5    "address": "test",
6    "prices": [],
7    "themes": []
8  },
9  {
10   "id": 9,
11   "name": "Musée du Louvre",
12   "address": "799, rue de Rivoli, 75001 Paris",
13   "prices": [
14     {
15       "id": 6,
16       "type": "less_than_12",
17       "value": 6
18     },
19     {
20       "id": 7,
21       "type": "for_all",
22       "value": 15
23     }
24   ],
25   "themes": []
26 }
27 ]
```

Il est aussi possible de configurer `FOSRestBundle` pour injecter directement les query strings dans l'objet `Request`. Pour plus d'informations, vous pouvez consulter [la documentation du bundle](#) [↗](#).

III. Amélioration de l'API REST

Les query strings permettent d'étendre facilement une API REST tout en respectant les contraintes que ce style d'architecture nous impose.

Nous venons de broser une infime partie des fonctionnalités que les query strings peuvent apporter à une API.

D'ailleurs, il n'existe pas de limites réelles et vous pouvez laisser libre cours à votre imagination pour étoffer notre API.

De la même façon, le bundle *FOSRestBundle* propose un ensemble de fonctionnalité grâce au *Param Fetcher Listener* qui permettent de gérer les query strings d'une manière assez simple.

[La documentation officielle](#) [↗](#) est complète sur le sujet et pourra toujours vous servir de référence.

15. JMSSerializer : Une alternative au sérialiseur natif de Symfony

Le sérialiseur natif de Symfony est disponible depuis les toutes premières versions du framework. Cependant, les fonctionnalités supportées par celui-ci étaient assez basique.

Par exemple, les groupes de sérialisation - permettant entre autres de gérer les références circulaires - n'ont été supportés qu'à partir de la version 2.7 [sortie en 2015](#) . La sérialisation des dates PHP (*DateTime* et *DateTimeImmutable*) n'a été supporté qu'avec la version 3.1 [sortie en 2016](#) .

Pour pallier à ce retard, un bundle a été développé pour la gestion de la sérialisation dans Symfony : *JMSSerializerBundle*. Il permet d'intégrer la librairie [JMSSerializer](#) et est très largement utilisé dans le cadre du développement d'une API avec Symfony.

15.1. Pourquoi utiliser JMSSerializerBundle ?

Nous avons déjà eu l'occasion de voir le nom *JMSSerializerBundle* dans les premières parties de ce cours. Ce bundle permet d'inclure et de configurer la librairie PHP [jms/serializer](#) dans Symfony.

Cette librairie présente beaucoup d'avantages :

- Elle est beaucoup plus mature que le sérialiseur de Symfony ;
- De par son ancienneté, elle est supportée par beaucoup de bundles et facilite donc l'interopérabilité entre les bundles et/ou composants que nous pouvons utiliser dans notre API ;
- Et pour finir, les ressources (documentation, cours etc.) sur cette librairie sont plus abondantes.

À l'installation de *FOSRestBundle*, nous étions obligés d'utiliser la version 2.0 afin de supporter pleinement le sérialiseur de Symfony. Mais avec *JMSSerializerBundle*, nous pourrions profiter de toutes les fonctionnalités de *jms/serializer* tout en utilisant une version de *FOSRestBundle* inférieure à la 2.0.

15.2. Installation et configuration de JMSSerializerBundle

15.2.1. Installation de JMSSerializerBundle

Comme pour tous les bundles de Symfony, il suffit de le télécharger avec *Composer* et de l'activer. Téléchargement du bundle :

```
1 composer require jms/serializer-bundle
2 # Using version ^1.1 for jms/serializer-bundle
3 ./composer.json has been updated
```

Activation du bundle :

```
1 <?php
2 # app/AppKernel.php
3 use Symfony\Component\HttpKernel\Kernel;
4 use Symfony\Component\Config\Loader\LoaderInterface;
5
6 class AppKernel extends Kernel
7 {
8     public function registerBundles()
9     {
10         $bundles = [
11             // ...
12             new JMS\SerializerBundle\JMSSerializerBundle(),
13             new FOS\RestBundle\FOSRestBundle(),
14             new AppBundle\AppBundle(),
15         ];
16         // ...
17         return $bundles;
18     }
19     // ...
20 }
```

15.2.2. Configuration de JMSSerializerBundle

La configuration par défaut de ce bundle suffit largement pour commencer à l'exploiter. Mais pour notre cas, puisque nous avons déjà pas mal de fonctionnalités qui dépendent du sérialiseur, nous allons modifier sa configuration.

15.2.2.1. Gestion des dates PHP

Comme pour le sérialiseur natif de Symfony (depuis la version 3.1), la sérialisation des dates dans php est supportée nativement par *JMSSerializerBundle*. Nous pouvons, en plus, personnaliser ce comportement avec juste 4 lignes de configuration.

```
1 # app/config/config.yml
2
3 # ...
4 jms_serializer:
5     handlers:
6         datetime:
7             default_format: "Y-m-d\\TH:i:sP"
8             default_timezone: "UTC"
```

La valeur "Y-m-d\\TH:i:sP" désigne [le format ISO 8601](#) pour les dates.



L'attribut `default_format` prend en paramètre le même format que la fonction `date` de PHP.

15.2.2.2. Une question de casse : CamelCase ou snake_case ?

Dans tous les exemples que nous avons pu voir, les attributs dans les requêtes et les réponses sont toutes en minuscules. À part l'attribut `plainPassword` utilisé pour créer un utilisateur et le champ `createdAt` associé à un token d'authentification, toutes nos attributs sont en minuscule. Mais dans le cadre d'une API plus complète, la question de la casse va se poser.

La seule contrainte qu'il faudra garder en tête est **la cohérence**. Si nous décidons d'utiliser des noms d'attributs en camelCase ou en snake_case, il faudra s'en tenir à ça pour tous les appels de l'API.

La configuration de tels paramètres est très simple aussi bien avec le sérialiseur de base de Symfony qu'avec le *JMSSerializer*. Nous allons donc garder la configuration par défaut du sérialiseur de Symfony qui est de conserver le même nom que celui des attributs de nos objets.

```
1 # app/config/config.yml
2
3 imports:
4     - { resource: parameters.yml }
5     - { resource: security.yml }
6     - { resource: services.yml }
7
8 parameters:
9     locale: en
```

```
10     jms_serializer.camel_case_naming_strategy.class:  
11         JMS\Serializer\Naming\IdenticalPropertyNamingStrategy  
12 # ...
```

15.2.2.3. Désactivation du sérialiseur natif

Maintenant que nous avons fini la configuration, il faut désactiver le sérialiseur natif de Symfony.

Vu qu'il n'est pas activé par défaut, nous pouvons retirer la configuration associée ou passer sa valeur à `false`.

```
1 # app/config/config.yml  
2  
3 # ...  
4  
5 framework:  
6     # ...  
7     serializer:  
8         enabled: false  
9 # ...
```

FOSRestBundle va maintenant utiliser directement le sérialiseur fournit par *JMSSerializerBundle*.

15.2.3. Sérialiser les attributs même s'ils sont nuls

Le comportement par défaut de *JMSSerializer* est d'ignorer tous les attributs nuls d'un objet. Ce fonctionnement peut entrainer des réponses avec des payloads partiels manquant certains attributs. Pour éviter ce problème, *FOSRestBundle* propose un paramètre de configuration pour forcer *JMSSerializer* à sérialiser les attributs nuls.

```
1 # app/config/config.yml  
2  
3 # ...  
4  
5 fos_rest:  
6     serializer:  
7         serialize_null: true
```

15.3. Impact sur l'existant

15.3.1. Tests de la configuration

Pour tester notre configuration, nous allons lister les lieux dans notre application.

La réponse obtenue est :

```
1 {
2   "0": {},
3   "1": {}
4 }
```

Nous avons là une bonne et une mauvaise nouvelle. Le bundle est bien utilisé pour sérialiser la réponse mais les groupes de sérialisation, que nous avons définis, ne sont pas encore exploités.



Pourquoi la réponse n'est pas sérialisée correctement ?

Le sérialiseur est pleinement supporté par *FOSRestBundle*. Les configurations dans tous nos contrôleurs sont déjà compatibles. Par contre, le fichier *src/AppBundle/Resources/config/serialization.yml* décrivant les règles de sérialisation, est ignoré par *JMSSerializerBundle*.

La configuration par défaut se base sur une convention simple. Pour un bundle, les fichiers décrivant la sérialisation doivent être dans le dossier *src/NomDuBundle/Resources/config/serializer/*.

Le nom de chaque fichier contenant les règles de sérialisation d'une classe est obtenu en faisant deux opérations :

- le nom du bundle est retiré du namespace (espace de nom) de la classe ;
- les séparateurs anti-slash (\) sont remplacés par des points (.) ;
- et enfin, l'extension yml ou xml est rajouté au nom ainsi obtenu.

Par exemple, pour la classe `NomDuBundle\A\B`, si nous voulons utiliser une configuration en YAML, nous devons avoir un fichier *src/NomDuBundle/Resources/config/serializer/A.B.yml*.



JMSSerializerBundle supporte aussi les annotations et les fichiers XML pour la configuration des règles de sérialisation. D'ailleurs, si nous avons utilisé les annotations, le code fonctionnerait sans adaptation de notre part.

15.3.2. Mise à jour de nos règles de sérialisation

Pour remettre notre API d'aplomb, nous allons créer les fichiers de configuration pour les classes utilisées.

Commençons par l'entité `Place`. La configuration pour cette classe devient :

```
1 # src/AppBundle/Resources/config/serializer/Entity.Place.yml
2 AppBundle\Entity\Place:
3     exclusion_policy: none
4     properties:
5         id:
6             groups: ['place', 'price', 'theme']
7         name:
8             groups: ['place', 'price', 'theme']
9         address:
10            groups: ['place', 'price', 'theme']
11        prices:
12            groups: ['place']
13        themes:
14            groups: ['place']
```

Par défaut, aucune propriété de nos classes n'est affichée pendant la sérialisation. En mettant l'attribut `exclusion_policy` à `none`, nous configurons le sérialiseur pour inclure par défaut toutes les propriétés de la classe. Nous pourrions bien sûr exclure certaines propriétés à la demande (`exclude: true`).

De même, il est aussi possible d'adopter la stratégie inverse à savoir exclure par défaut toutes les propriétés de nos classes et les ajouter à la demande (`expose: true`).

Il faut aussi noter que l'attribut `attributes` dans l'ancien fichier de configuration est remplacé par `properties`. Tout le reste est identique à notre ancien fichier de configuration.

La configuration des nouvelles classes devient maintenant :

```
1 # src/AppBundle/Resources/config/serializer/Entity.Price.yml
2 AppBundle\Entity\Price:
3     exclusion_policy: none
4     properties:
5         id:
6             groups: ['place', 'price']
7         type:
8             groups: ['place', 'price']
9         value:
10            groups: ['place', 'price']
11        place:
12            groups: ['price']
```

III. Amélioration de l'API REST

```
1 # src/AppBundle/Resources/config/serializer/Entity.Theme.yml
2 AppBundle\Entity\Theme:
3   exclusion_policy: none
4   properties:
5     id:
6       groups: ['place', 'theme']
7     name:
8       groups: ['place', 'theme']
9     value:
10      groups: ['place', 'theme']
11     place:
12      groups: ['theme']
```

```
1 # src/AppBundle/Resources/config/serializer/Entity.User.yml
2 AppBundle\Entity\User:
3   exclusion_policy: none
4   properties:
5     id:
6       groups: ['user', 'preference', 'auth-token']
7     firstname:
8       groups: ['user', 'preference', 'auth-token']
9     lastname:
10      groups: ['user', 'preference', 'auth-token']
11     email:
12      groups: ['user', 'preference', 'auth-token']
13     preferences:
14      groups: ['user']
```

```
1 # src/AppBundle/Resources/config/serializer/Entity.Preference.yml
2 AppBundle\Entity\Preference:
3   exclusion_policy: none
4   properties:
5     id:
6       groups: ['user', 'preference']
7     name:
8       groups: ['user', 'preference']
9     value:
10      groups: ['user', 'preference']
11     user:
12      groups: ['preference']
```

III. Amélioration de l'API REST

```
1 # src/AppBundle/Resources/config/serializer/Entity.AuthToken.yml
2 AppBundle\Entity\AuthToken:
3   exclusion_policy: none
4   properties:
5     id:
6       groups: ['auth-token']
7     value:
8       groups: ['auth-token']
9     createdAt:
10      groups: ['auth-token']
11     user:
12      groups: ['auth-token']
```



N'oubliez pas de vider le cache pour éviter tout problème.

En testant cette nouvelle configuration, la liste des lieux dans notre application redevient correcte.

```
1 [
2   {
3     "id": 1,
4     "name": "Tour Eiffel",
5     "address": "5 Avenue Anatole France, 75007 Paris",
6     "prices": [
7       {
8         "id": 1,
9         "type": "less_than_12",
10        "value": 5.75
11      }
12    ],
13    "themes": [
14      {
15        "id": 1,
16        "name": "architecture",
17        "value": 7
18      },
19      {
20        "id": 2,
21        "name": "history",
22        "value": 6
23      }
24    ]
25  },
26  {
27    "id": 2,
```

III. Amélioration de l'API REST

```
28     "name": "Mont-Saint-Michel",
29     "address": "50170 Le Mont-Saint-Michel",
30     "prices": [],
31     "themes": [
32         {
33             "id": 3,
34             "name": "history",
35             "value": 3
36         },
37         {
38             "id": 4,
39             "name": "art",
40             "value": 7
41         }
42     ]
43 }
44 ]
```

Vous pouvez tester l'ensemble des appels que nous avons déjà mis en place. L'API se comporte exactement de la même façon.

L'intégration de *JMSSerializerBundle* avec *FOSRestBundle* est aussi simple qu'avec le sérialiseur natif de Symfony. En effet, *FOSRestBundle* nous offre une interface unique et s'adapte au sérialiseur mis à sa disposition.

En plus, le bundle [JMSSerializerBundle](#) supporte beaucoup de fonctionnalités que nous n'avons pas abordées (gestion des versions, propriétés virtuelles, etc.).

Vous avez pu remarquer que *JMSSerializer* nécessite un peu plus de configuration que le sérialiseur natif de Symfony. Par contre, le travail fourni pour obtenir un résultat correct est très rapidement rentabilisé vu que *JMSSerializerBundle* s'intègre facilement avec beaucoup d'autres bundles de Symfony.

Nous aurons d'ailleurs l'occasion d'exploiter ce bundle dans le chapitre sur la documentation.

16. La documentation avec OpenAPI (Swagger RESTful API)



Que serait une API s'il était impossible de comprendre son mode de fonctionnement ?

Parler de documentation dans une API RESTful se rapproche beaucoup d'un oxymore. En effet, une API dite RESTful devrait pouvoir être utilisée sans documentation.

Mais si vous vous souvenez bien, notre API n'implémente pas le niveau 3 du modèle de maturité de Richardson : HATEOAS qui permettrait de l'explorer automatiquement et d'interagir avec elle. Dès lors, pour faciliter son usage nous devons créer une documentation.

Elle permettra ainsi aux clients de notre API de comprendre son mode de fonctionnement et d'explorer rapidement les différentes fonctionnalités qu'elle expose.

Il existe un standard appelé *OpenAPI*, anciennement connu sous le nom de *Swagger RESTful API*, permettant d'avoir des spécifications simples pour une documentation exhaustive.

L'objectif de cette partie est d'avoir un aperçu de *OpenAPI* et de voir comment mettre en place une documentation en implémentant ces spécifications.

16.1. Qu'est-ce que OpenAPI ?

OpenAPI désigne un ensemble de spécifications [☞](#) permettant de **décrire** et de **documenter** une API REST.

Le terme **décrire** n'est pas utilisé par hasard car implémenter ces spécifications permet entre autres :

- d'obtenir une documentation ([Swagger UI ☞](#));
- et de générer des clients permettant d'interagir avec notre API ([Swagger Codegen ☞](#)).

Les spécifications permettent de créer un fichier JSON qui décrit l'ensemble des éléments d'une API (URL des ressources, code de statut des réponses, verbes HTTP utilisés, etc.). Par convention, ce fichier est souvent nommé *swagger.json*.

Pour cette partie nous allons commencer par la pratique avant d'explorer la théorie autour de la documentation des API REST avec *OpenAPI*.

16.2. Rédaction de la documentation

16.2.1. Quel outil pouvons-nous utiliser pour créer la documentation ?

Bien que le résultat final du fichier *OpenAPI* soit en JSON, il peut être rédigé aussi bien en JSON qu'en YAML. Nous préférons d'ailleurs le YAML par la suite.

Pour créer ce fichier *swagger.json*, il faut suivre les spécifications qui sont disponibles en ligne : [Spécification OpenAPI \(Swagger\)](#) ↗ .

L'un des moyens les plus simples pour rédiger et tester les spécifications est d'utiliser le site [Swagger Editor](#) ↗ . Ce site propose une prévisualisation de la documentation qui sera générée et des exemples de configuration (en YAML) qui permettent de mieux appréhender les spécifications d'*OpenAPI*.

16.2.2. Structure de base du fichier *swagger.json*

Un fichier *swagger.json* a trois attributs obligatoires :

- `swagger` : définit la version des spécifications utilisées ;
- `info` : définit les métadonnées de notre API ;
- et `paths` : définit les différentes URL et opérations disponibles dans l'API.

Le fichier de base ressemble donc a :

```
1 swagger: '2.0' # obligatoire
2 info: # obligatoire
3   title: Proposition de suggestions API # obligatoire
4   description: Proposer des idées de sortie à des utilisateurs en
5     utilisant leurs préférences
6   version: "1.0.0" # obligatoire
7 host: rest-api.local
8 schemes:
9   - http
10 produces:
11   - application/json
12   - application/xml
13 consumes:
14   - application/json
15   - application/xml
16
17 paths: # obligatoire
```

Proposition de suggestions API

Proposer des idées de sortie à des utilisateurs en utilisant leurs préférences

Version 1.0.0

Paths

FIGURE 16.1. – Prévisualisation de la structure de base

Les attributs `produces` et `consumes` permettent de décrire les type **MIME** des réponses renvoyées et des requêtes acceptées par notre API. Il est possible d'utiliser du [Markdown](#) pour formater les différentes descriptions (attribut `description`) dans la documentation.

i

Tous les tests se feront en utilisant directement le site <http://editor.swagger.io>. Le fichier `swagger.json` définitif sera testé en local dans la dernière partie.

16.2.3. Déclarer une opération avec l'API

16.2.3.1. Documentation de la méthode de connexion

Pour commencer, nous allons essayer de rédiger la documentation de la méthode d'authentification à l'API. Pour déclarer une opération, nous devons utiliser l'attribut `paths`.

```
1 swagger: '2.0' # obligatoire
2 info: # obligatoire
3   title: Proposition de suggestions API # obligatoire
4   description: Proposer des idées de sortie à des utilisateurs en
5     utilisant leurs préférences
6   version: "1.0.0" # obligatoire
7 host: rest-api.local
8 schemes:
9   - http
10 produces:
11   - application/json
12   - application/xml
13 consumes:
14   - application/json
15   - application/xml
16
17 paths: # obligatoire
18   /auth-tokens:
19     post:
```

III. Amélioration de l'API REST

```
20     summary: Authentifie un utilisateur
21     description: Crée un token permettant à l'utilisateur
                d'accéder aux contenus protégés
22     responses: # obligatoire
```

Voici la base permettant de créer des opérations. Sous l'attribut `paths`, il faut définir l'URL de notre ressource et ensuite il faut déclarer les différents verbes HTTP qui sont utilisés sur celle-ci. Actuellement, nous avons la méthode `POST` permettant de créer un token. Nous devons maintenant définir :

- le payload de la requête ;
- la réponse en cas de succès ;
- la réponse en cas d'erreur.

Toutes ces données sont déclarées en utilisant les spécifications de [JSON Schema](#) .

```
1 # ...
2 paths: # obligatoire
3   /auth-tokens:
4     post:
5       summary: Authentifie un utilisateur
6       description: Crée un token permettant à l'utilisateur
                d'accéder aux contenus protégés
7       parameters:
8         - name: credentials # obligatoire
9           in: body # obligatoire
10          required: true
11          description: Login et mot de passe de l'utilisateur
12          schema:
13            type: object
14            required: [login, password]
15            properties:
16              login:
17                type: string
18              password:
19                type: string
20
21
22      responses:
23        200:
24          description: Token créé # obligatoire
25          schema:
26            type: object
27            properties:
28              id:
29                type: integer
30              value:
31                type: string
```


III. Amélioration de l'API REST

```
32         created_at:
33             type: string
34             format: date-time
35         user:
36             type: object
37             properties:
38                 id:
39                     type: integer
40                 email:
41                     type: string
42                     format: email
43                 firstname:
44                     type: string
45                 lastname:
46                     type: string
47
48     400:
49         description: Donnée invalide # obligatoire
50         schema:
51             type: object
52             required: [message]
53             properties:
54                 code:
55                     type: integer
56                 message:
57                     type: string
58                 errors:
59                     type: object
60                     properties:
61                         children:
62                             type: object
63                             properties:
64                                 login:
65                                     type: object
66                                     properties:
67                                         errors:
68                                             type: array
69                                             items:
70                                                 type: string
71                                 password:
72                                     type: object
73                                     properties:
74                                         errors:
75                                             type: array
76                                             items:
77                                                 type: string
```

Parameters

| Name | Located in | Description | Required | Schema |
|-------------|------------|--|----------|---|
| credentials | body | Login et mot de passe de l'utilisateur | Yes | <pre> { login: string * password: string * } </pre> |

Responses

| Code | Description | Schema |
|------|-----------------|---|
| 200 | Token créé | <pre> { id: integer value: string created_at: string user: { } } </pre> |
| 400 | Donnée invalide | <pre> { code: integer message: string * errors: { } } </pre> |

FIGURE 16.2. – Documentation de la méthode de création d'un token

Il est aussi possible de mieux organiser le fichier en rajoutant une entrée `definitions` qui permet de regrouper tous les schémas que nous avons déclarés. Ensuite, il suffira de faire référence à ces schémas en utilisant l'attribut `$ref`.

```

1 # ...
2 paths: # obligatoire
3   /auth-tokens:
4     post:
5       summary: Authentifie un utilisateur
6       description: Crée un token permettant à l'utilisateur
7         d'accéder aux contenus protégés
8       parameters:
9         - name: credentials # obligatoire
10          in: body # obligatoire
11          required: true
12          description: Login et mot de passe de l'utilisateur
13          schema:
14            $ref: "#/definitions/Credentials"
                    
```

III. Amélioration de l'API REST

```
15     responses:
16       200:
17         description: Token créé # obligatoire
18         schema:
19           $ref: "#/definitions/AuthToken.auth-token"
20
21       400:
22         description: Donnée invalide # obligatoire
23         schema:
24           $ref: "#/definitions/CredentialsTypeError"
25
26 definitions:
27   Credentials:
28     type: object
29     required: [login, password]
30     properties:
31       login:
32         type: string
33       password:
34         type: string
35
36   AuthToken.auth-token:
37     type: object
38     required: [id, value, created_at, user]
39     properties:
40       id:
41         type: integer
42       value:
43         type: string
44         title: Token d'authentification
45         description: Valeur à utiliser dans l'entête X-Auth-Token
46       created_at:
47         type: string
48         format: date-time
49       user:
50         type: object
51         properties:
52           id:
53             type: integer
54           email:
55             type: string
56             format: email
57           firstname:
58             type: string
59           lastname:
60             type: string
61
62   CredentialsTypeError:
63     type: object
64     required: [message]
```

III. Amélioration de l'API REST

```
65     properties:
66         code:
67             type: integer
68         message:
69             type: string
70         errors:
71             type: object
72             properties:
73                 children:
74                     type: object
75                     properties:
76                         login:
77                             type: object
78                             properties:
79                                 errors:
80                                     type: array
81                                     items:
82                                         type: string
83                         password:
84                             type: object
85                             properties:
86                                 errors:
87                                     type: array
88                                     items:
89                                         type: string
```

Avec ces modifications, le résultat obtenu est exactement identique.

16.2.3.2. Documentation de la méthode de déconnexion

De la même façon pour documenter la suppression d'un token, nous devons rajouter une nouvelle URL. Mais cette fois-ci, elle doit être dynamique comme pour les routes Symfony.

```
1 # ...
2
3 paths: # obligatoire
4     /auth-tokens:
5         # ...
6
7     /auth-tokens/{id}:
8         delete:
9             summary: Déconnecte un utilisateur
10            description: Supprime le token de l'utilisateur
11            parameters:
12                - $ref: "#/parameters/X-Auth-Token"
13                - name: id # obligatoire
14                  in: path # obligatoire
```

III. Amélioration de l'API REST

```
15         type: integer # obligatoire si le paramètre dans in est
16             différent de 'body'
17         required: true
18         description: Identifiant du token à supprimer
19     responses:
20         204:
21             description: Token supprimé # obligatoire
22
23         400:
24             description: Donnée invalide # obligatoire
25             schema:
26                 $ref: "#/definitions/GenericError"
27
28
29     parameters:
30         X-Auth-Token:
31             name: X-Auth-Token # obligatoire
32             in: header # obligatoire
33             type: string # obligatoire si le paramètre dans in est
34                 différent de 'body'
35             required: true
36             description: Valeur du token d'authentification
37
38     definitions:
39         # ...
40         GenericError:
41             type: object
42             required: [code, message]
43             properties:
44                 code:
45                     type: string
46                 message:
47                     type: string
```

À l'instar de la méthode de connexion, nous utilisons aussi le paramètre `in` pour désigner l'identifiant du token. Cet attribut peut valoir :

- `path` : le paramètre est extrait de l'URL de la ressource ;
- `query` : le paramètre est un query string ;
- `header` : le paramètre est une entête HTTP ;
- `body` : le paramètre est dans le payload ;
- `form` : le paramètre est dans le payload qui est encodé au format *application/x-www-form-urlencoded* ou *multipart/form-data* (c'est le format utilisé par un formulaire classique).

L'entête HTTP `X-Auth-Token` est utilisée par plusieurs requêtes de notre API. En le déclarant dans l'attribut `parameters`, cela nous permet de le réutiliser dans les appels API qui nous intéressent.



Il existe deux attributs `securityDefinitions` et `security` permettant de configurer la méthode d'authentification sans passer par l'attribut `parameters`. Mais pour les besoins de cet exemple, nous ne les utiliserons pas.

DELETE /auth-tokens/{id}

Summary
Déconnecte un utilisateur

Description
Supprime le token de l'utilisateur

Parameters

| Name | Located in | Description | Required | Schema |
|--------------|------------|------------------------------------|----------|-----------|
| X-Auth-Token | header | Valeur du token d'authentification | Yes | ⇌ string |
| id | path | Identifiant du token à supprimer | Yes | ⇌ integer |

Responses

| Code | Description | Schema |
|------|-----------------|---|
| 204 | Token supprimé | |
| 400 | Donnée invalide | <pre> ▼ CredentialsTypeError { code: integer message: string * errors: ▶ { } } </pre> |

FIGURE 16.3. – Documentation de la méthode de suppression d'un token

Toutes les informations utilisées pour créer ce fichier sont issues [des spécifications officielles d'OpenAPI](#) [↗](#). Vous pourrez les consulter afin de voir l'ensemble des fonctionnalités qu'offrent *OpenAPI*.

16.3. Installer et utiliser Swagger UI

Swagger UI est un logiciel basé sur les technologies du web (HTML, Javascript, CSS) permettant de générer une documentation en utilisant les spécifications d'*OpenAPI*. Il fournit aussi un bac à sable permettant de tester les appels API directement depuis la documentation générée.

16.3.1. Installation de Swagger UI

Pour installer Swagger UI, il suffit de [le télécharger depuis GitHub](#) . Ensuite, nous allons le décompresser dans un dossier nommé *swagger-ui* dans le répertoire *web*. Nous utiliserons la version v2.1.4.

i

Si vous utilisez *git*, il suffit de se placer dans le dossier *web* et de lancer :

```
1 git clone https://github.com/swagger-api/swagger-ui.git
2 git checkout v2.1.4
```

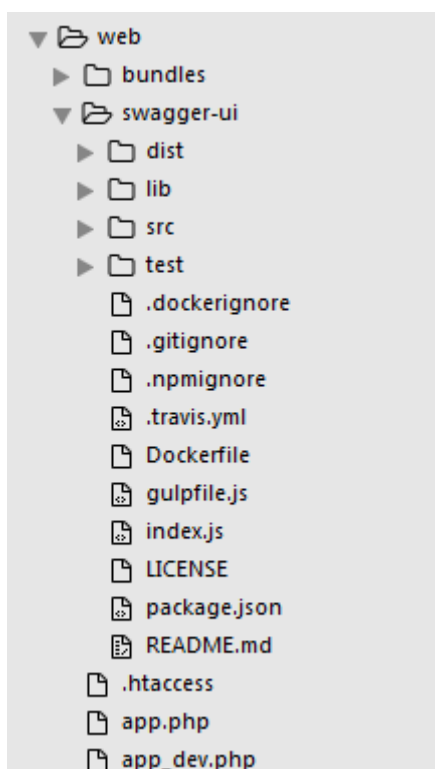
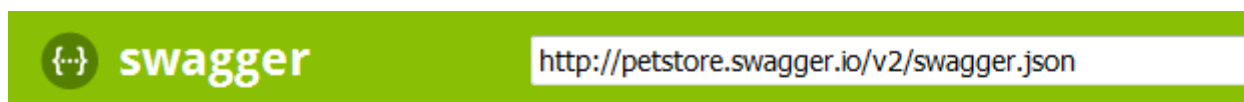


FIGURE 16.4. – Arborescence après l'installation de Swagger UI

Si l'installation s'est bien déroulée, en accédant à l'URL <http://rest-api.local/swagger-ui/dist/index.html> , la page d'accueil de Swagger UI s'affiche.



Swagger Petstore

This is a sample server Petstore server. You can find out more about Swagger at <http://swagger.io> sample, you can use the api key `special-key` to test the authorization filters.

FIGURE 16.5. – Page d'accueil de Swagger UI

16.3.2. Utiliser notre documentation

Depuis l'interface de Swagger Editor, il est possible d'exporter notre documentation au format JSON. Le fichier *swagger.json* ainsi obtenu ressemble à :

```

1 {
2   "swagger": "2.0",
3   "info": {
4     "title": "Proposition de suggestions API",
5     "description":
6       "Proposer des idées de sortie à des utilisateurs en utilisant leurs
7   },
8   "host": "rest-api.local",
9   "schemes": [
10    "http"
11  ],
12  "produces": [
13    "application/json",
14    "application/xml"
15  ],
16  "consumes": [
17    "application/json",
18    "application/xml"
19  ],
20  "paths": {
21    // ...
22  },
23  "parameters": {
24    // ...
25  },
26  "definitions": {
27    // ...
28  }
29 }

```

Pour utiliser ce fichier *swagger.json*, il faut commencer par l'enregistrer dans le dossier *web*. Le fichier doit être disponible depuis un navigateur. Ensuite, il faut éditer le fichier *web/swagger-ui/dist/index.html* et éditer les lignes 34 à 39.

```

1  /*var url = window.location.search.match(/url=([^&]+)/);
2  if (url && url.length > 1) {
3    url = decodeURIComponent(url[1]);
4  } else {

```



```
5     url = "http://petstore.swagger.io/v2/swagger.json";
6     }*/
7     var url = "/swagger.json";
```

En consultant l'URL, nous pouvons maintenant voir notre documentation et même tester les appels API depuis celui-ci.

Proposition de suggestions API

Proposer des idées de sortie à des utilisateurs en utilisant leurs préférences

default

| | |
|--------|-------------------|
| POST | /auth-tokens |
| DELETE | /auth-tokens/{id} |

FIGURE 16.6. – Documentation de notre API avec Swagger UI

Après cette brève initiation à *OpenAPI*, connu aussi sous le nom de *Swagger RESTful API*, vous avez pu remarquer que l'écosystème autour de cette technologie est assez riche.

Ces spécifications se basent sur un ensemble de standards reconnus comme [JSON Schema](#) qui facilitent grandement sa prise en main.

Le fichier *swagger.json* ainsi obtenu peut être exploité par beaucoup d'outils qui permettent d'augmenter notre productivité (Génération de code client, génération de code serveur, interface de documentation avec bac à sable, etc.).

17. Automatiser la documentation avec NelmioApiDocBundle

Bien que les outils de l'écosystème de *OpenAPI (Swagger RESTFull API)* soient assez bien fournis, rédiger manuellement toute la documentation peut se montrer assez rapidement rébarbatif.

En plus, à cause de la séparation entre le code et la documentation, cette dernière risque de ne pas être mise à jour si le code évolue.

Nous allons donc voir comment automatiser la génération de la documentation dans Symfony avec le bundle *NelmioApiDocBundle*.

Cette partie n'abordera pas toutes les fonctionnalités de ce bundle mais permettra d'avoir assez de bagages pour être autonome.

17.1. Installation de NelmioApiDocBundle

La référence en matière de documentation d'une API avec Symfony est le bundle *NelmioApiDocBundle*. Comme pour tous les bundles de Symfony, l'installation est particulièrement simple. Avec *Composer*, nous allons rajouter la dépendance :

```
1 composer require nelmio/api-doc-bundle
2 # Using version ^2.12 for nelmio/api-doc-bundle
3 ./composer.json has been updated
```

Nous pouvons maintenant activer le bundle :

```
1 <?php
2 # app/AppKernel.php
3
4 use Symfony\Component\HttpKernel\Kernel;
5 use Symfony\Component\Config\Loader\LoaderInterface;
6
7 class AppKernel extends Kernel
8 {
9     public function registerBundles()
10    {
11        $bundles = [
```

```
12         // ...
13         new Nelmio\ApiDocBundle\NelmioApiDocBundle(),
14         new AppBundle\AppBundle(),
15     ];
16
17     // ...
18     return $bundles;
19 }
20 // ...
21 }
```

17.2. L'annotation ApiDoc

17.2.1. Configuration

Pour générer de la documentation, le bundle *NelmioApiDocBundle* se base sur une fonctionnalité principale : l'annotation `ApiDoc`.

À son installation, ce bundle met à notre disposition cette annotation qui va nous permettre de **rédigier** notre documentation.

i

Il faut garder en tête que la documentation avec *NelmioApiDocBundle* est grandement liée au code.

Sans plus attendre, nous allons l'utiliser pour documenter l'appel qui liste les lieux de notre application.

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5
6 // ...
7 use Nelmio\ApiDocBundle\Annotation\ApiDoc;
8 // ...
9
10 class PlaceController extends Controller
11 {
12
13     /**
14      * @ApiDoc(
15      *     description="Récupère la liste des lieux de l'application"
16      * )
17      *
18     */
19 }
```

III. Amélioration de l'API REST

```
18 *
19 * @Rest\View(serializerGroups={"place"})
20 * @Rest\Get("/places")
21 * @QueryParam(name="offset", requirements="\d+", default="", description="")
22 * @QueryParam(name="limit", requirements="\d+", default="", description="")
23 * @QueryParam(name="sort", requirements="(asc|desc)", nullable=true, desc="")
24 */
25 public function getPlacesAction(Request $request, ParamFetcher
    $paramFetcher)
26 {
27     // ...
28
29     return $places;
30 }
31 // ...
32 }
```

Avec juste cette annotation, il est possible de consulter la documentation de notre API. Mais avant d'y accéder, nous devons avoir une URL dédiée. Et pour ce faire, le bundle propose un fichier de routage qui permet de configurer cette URL.

```
1 # app/config/routing.yml
2
3 # ...
4 nelmio-api-doc:
5     resource: "@NelmioApiDocBundle/Resources/config/routing.yml"
6     prefix:   /documentation
```

Nous allons aussi rajouter une règle dans le pare-feu de Symfony afin d'autoriser l'accès à la documentation sans authentification.

```
1 # app/config/security.yml
2 security:
3
4 # ...
5     firewalls:
6         # disables authentication for assets and the profiler,
7         # adapt it according to your needs
8         dev:
9             pattern: ^/(_(profiler|wdt)|css|images|js)/
10            security: false
11
12        doc:
13            pattern: ^/documentation
14            security: false
15 # ...
```

III. Amélioration de l'API REST

Notre documentation est maintenant accessible depuis l'URL <http://rest-api.local/documentation> ↗.

En y accédant depuis un navigateur, nous obtenons une page générée automatiquement :

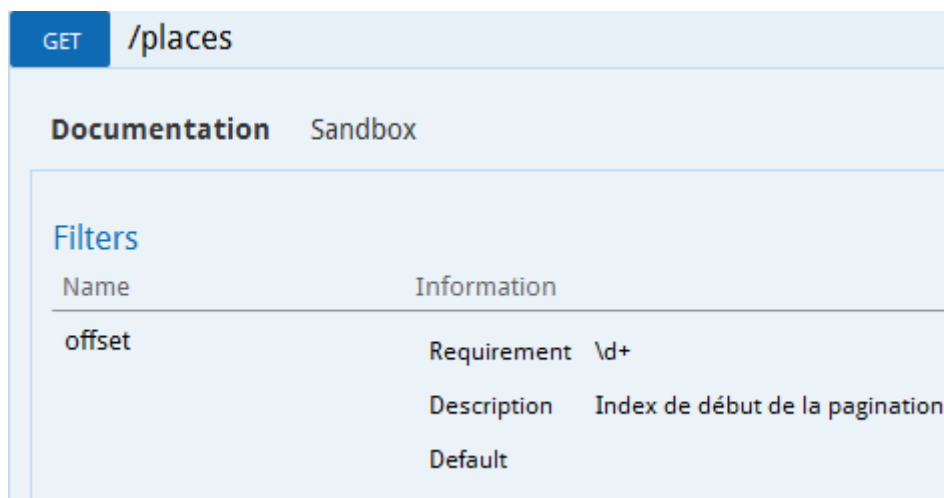


FIGURE 17.1. – Documentation générée par NelmioApiDocBundle

Pour avoir une vue complète (comme sur l'image), il faut cliquer sur la méthode `GET /places` pour dérouler les détails concernant les filtres. La mise en page de la documentation est grandement inspiré de *Swagger UI*.

17.2.2. Intégration avec FOSRestBundle

Le premier point qui devrait vous interpellier est la présence des filtres de *FOSRestBundle* dans la documentation. *NelmioApiDocBundle* a été conçu pour interagir avec la plupart des bundles utilisés dans le cadre d'une API. Ainsi, les annotations de *FOSRestBundle* sont utilisées pour compléter la documentation.

Bien sûr, si nous n'utilisons pas *FOSRestBundle*, nous pouvons rajouter manuellement des filtres en utilisant l'attribut `filters` de l'annotation `ApiDoc`.

De la même façon, le verbe HTTP utilisé est `GET` avec une URL `/places`. Là aussi, les routes générées par Symfony sont utilisées par *NelmioApiDocBundle*.

17.2.3. Définir le type des réponses de l'API

Notre documentation n'est pas encore complète. Le type des réponses renvoyées par notre API n'est pas encore documenté.

Pour ce faire, il existe un attribut nommé `output` qui prend comme paramètre le nom d'une classe ou encore une collection. Cet attribut supporte aussi les groupes de sérialisation que nous avons déjà définis.

Pour le cas des lieux, nous devons renvoyer une collection de lieux. La documentation s'écrit donc :

III. Amélioration de l'API REST

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5 // ...
6 use Nelmio\ApiDocBundle\Annotation\ApiDoc;
7 // ...
8
9 class PlaceController extends Controller
10 {
11
12     /**
13      * @ApiDoc(
14      *     description="Récupère la liste des lieux de l'application",
15      *     output= { "class"=Place::class, "collection"=true, "groups"={"place"}
16      * )
17      * @Rest\View(serializerGroups={"place"})
18      * @Rest\Get("/places")
19      * @QueryParam(name="offset", requirements="\d+", default="", description="")
20      * @QueryParam(name="limit", requirements="\d+", default="", description="")
21      * @QueryParam(name="sort", requirements="(asc|desc)", nullable=true, desc
22      */
23     public function getPlacesAction(Request $request, ParamFetcher
24         $paramFetcher)
25     {
26         // ...
27     }
28 // ...
29 }
```

La documentation devient :

| Return | | | |
|-------------------|--------------------------|----------|-------------|
| Parameter | Type | Versions | Description |
| 200 | | | |
| [] | array of objects (Place) | * | |
| [][name] | string | * | |
| [][address] | string | * | |
| [][prices][] | array of objects (Price) | * | |
| [][prices][id] | integer | * | |
| [][prices][type] | string | * | |
| [][prices][value] | float | * | |
| [][id] | integer | * | |
| [][themes][] | array of objects (Theme) | * | |
| [][themes][id] | integer | * | |
| [][themes][name] | string | * | |
| [][themes][value] | integer | * | |

FIGURE 17.2. – Type de réponse pour la liste des lieux

La documentation est complétée et les attributs ont exactement les bon types définis dans les annotations *Doctrine*. Pour obtenir de telles informations, *NelmioApiDocBundle* utilise le sérialiseur de *JMSSerializerBundle*.



Par contre, si nous étions restés sur le sérialiseur natif de Symfony qui n'est pas encore supporté, nous n'aurions pas pu obtenir ces informations.

Les descriptions de tous les attributs sont vides. Pour les renseigner, il suffit de rajouter dans les entités une description dans le bloc de *PHPDoc*.

Pour l'entité `Place`, nous pouvons rajouter :

```

1 <?php
2     /**
3      * Identifiant unique du lieu
4      *
5      * @ORM\Id
6      * @ORM\Column(type="integer")

```

III. Amélioration de l'API REST

```
7 * @ORM\GeneratedValue
8 */
9 protected $id;
```

La documentation générée devient alors :

| | | | |
|--------|---------|---|----------------------------|
| [[id]] | integer | * | Identifiant unique du lieu |
|--------|---------|---|----------------------------|

FIGURE 17.3. – Description de l'identifiant du lieu dans la documentation

17.2.4. Définir le type des payloads des requêtes

De la même façon, pour définir la structure des payloads des requêtes, nous pouvons utiliser un attribut nommé `input` qui peut prendre en paramètre, entre autres, une classe qui implémente l'interface PHP `JsonSerializable` mais aussi un formulaire Symfony. Et cela tombe bien puisque que tous nos payloads se basent sur ces formulaires.

Pour tester le bon fonctionnement de cet attribut, nous allons rajouter de la documentation pour la méthode de création d'un lieu.

```
1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3
4 namespace AppBundle\Controller;
5
6 // ...
7 use Nelmio\ApiDocBundle\Annotation\ApiDoc;
8 // ...
9
10 class PlaceController extends Controller
11 {
12     // ...
13     /**
14      * @ApiDoc(
15      *     description="Crée un lieu dans l'application",
16      *     input={"class"=PlaceType::class, "name":""}
17      * )
18      *
19      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"place"})
20      * @Rest\Post("/places")
21      */
22     public function postPlacesAction(Request $request)
23     {
24         // ...
25     }
26     // ...
```



```
27 }
```

POST
/places
Crée un lieu dans l'application

Documentation Sandbox

Parameters

| Parameter | Type | Required? | Format | Description |
|-----------------|------------------------------|-----------|--------|-------------|
| name | string | true | | |
| address | string | true | | |
| prices[] | array of objects (PriceType) | true | | |
| prices[][type] | string | true | | |
| prices[][value] | float | true | | |

FIGURE 17.4. – Documentation générée par NelmioApiDocBundle

Pour rajouter des descriptions pour les différents attributs des formulaires, nous pouvons utiliser une option nommée `description` rajoutée aux formulaires Symfony par *NelmioApiDocBundle*.

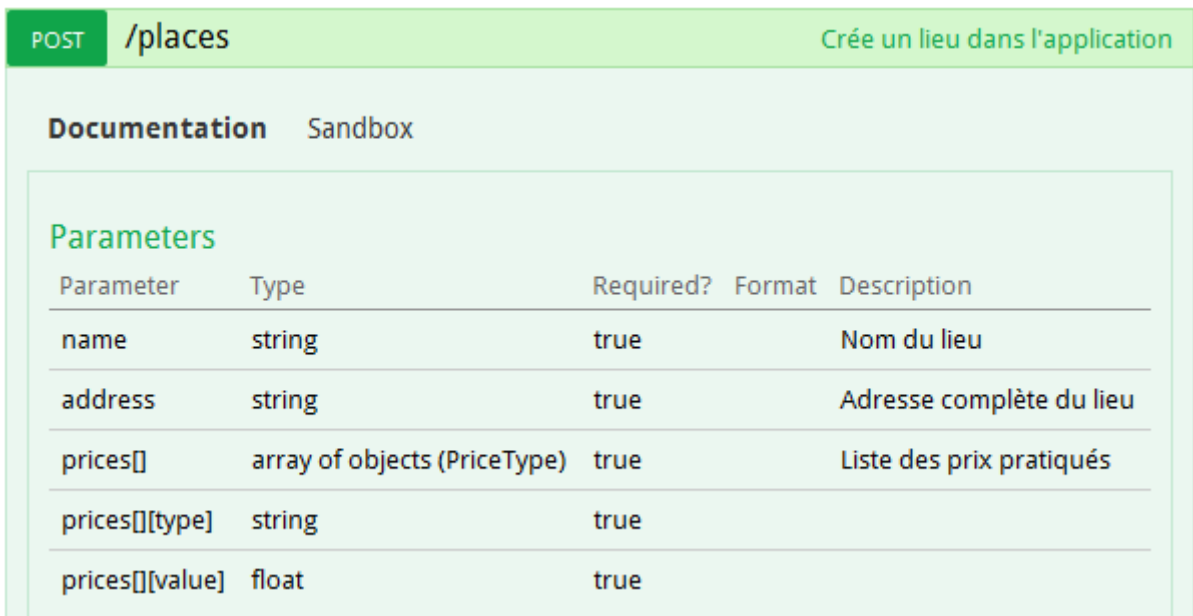
```

1  <?php
2  # src/AppBundle/Form/Type/PlaceType.php
3
4  namespace AppBundle\Form\Type;
5
6  use Symfony\Component\Form\AbstractType;
7  use Symfony\Component\Form\Extension\Core\Type\CollectionType;
8  use Symfony\Component\Form\Extension\Core\Type\TextType;
9  use Symfony\Component\Form\FormBuilderInterface;
10 use Symfony\Component\OptionsResolver\OptionsResolver;
11
12 class PlaceType extends AbstractType
13 {
14     public function buildForm(FormBuilderInterface $builder, array
15         $options)
16     {
17         $builder->add('name', TextType::class, [
18             'description' => "Nom du lieu"
19         ]);
20         $builder->add('address', TextType::class, [
21             'description' => "Adresse complète du lieu"
22         ]);
23     }
24 }
```

```

22     $builder->add('prices', CollectionType::class, [
23         'entry_type' => PriceType::class,
24         'allow_add' => true,
25         'error_bubbling' => false,
26         'description' => "Liste des prix pratiqués"
27     ]);
28 }
29
30 public function configureOptions(OptionsResolver $resolver)
31 {
32     $resolver->setDefaults([
33         'data_class' => 'AppBundle\Entity\Place',
34         'csrf_protection' => false
35     ]);
36 }
37 }

```



| Parameter | Type | Required? | Format | Description |
|-----------------|------------------------------|-----------|--------|--------------------------|
| name | string | true | | Nom du lieu |
| address | string | true | | Adresse complète du lieu |
| prices[] | array of objects (PriceType) | true | | Liste des prix pratiqués |
| prices[][type] | string | true | | |
| prices[][value] | float | true | | |

FIGURE 17.5. – Documentation complétée avec les descriptions des attributs

17.2.5. Gérer plusieurs codes de statut

En définissant l'attribut `output`, le code de statut associé par défaut est 200. Mais pour la création d'un lieu, nous devons avoir un code 201. Et de la même façon si le formulaire est invalide, nous voulons renvoyer une erreur 400 avec les messages de validation. Pour obtenir un tel résultat, *NelmioApiDocBundle* met à notre disposition un attribut `responseMap`.

```

1 <?php
2 # src/AppBundle/Controller/PlaceController.php
3

```

III. Amélioration de l'API REST

```
4 namespace AppBundle\Controller;
5
6 // ...
7 use Nelmio\ApiDocBundle\Annotation\ApiDoc;
8 // ...
9
10 class PlaceController extends Controller
11 {
12     // ...
13     /**
14      * @ApiDoc(
15      *     description="Crée un lieu dans l'application",
16      *     input={"class"]=PlaceType::class, "name"=""},
17      *     statusCodes = {
18      *         201 = "Création avec succès",
19      *         400 = "Formulaire invalide"
20      *     },
21      *     responseMap={
22      *         201 = {"class"]=Place::class, "groups"={"place"}},
23      *         400 = { "class"]=PlaceType::class, "form_errors"=true, "name" =
24      *     }
25      * )
26      *
27      * @Rest\View(statusCode=Response::HTTP_CREATED, serializerGroups={"place"})
28      * @Rest\Post("/places")
29      */
30     public function postPlacesAction(Request $request)
31     {
32         // ...
33     }
34     // ...
35 }
```

Le paramètre `form_errors` permet de spécifier le type de retour que nous voulons à savoir les erreurs de validation.

| 201 - Création avec succès | | | |
|----------------------------|--------------------------|---|----------------------------|
| name | string | * | |
| address | string | * | |
| prices[] | array of objects (Price) | * | |
| prices[][id] | integer | * | |
| prices[][type] | string | * | |
| prices[][value] | float | * | |
| id | integer | * | Identifiant unique du lieu |
| themes[] | array of objects (Theme) | * | |
| themes[][id] | integer | * | |
| themes[][name] | string | * | |
| themes[][value] | integer | * | |

FIGURE 17.6. – Documentation de la création avec succès

| 400 - Formulaire invalide | | | |
|---------------------------|------------------|---|-------------------------|
| status_code | integer | * | The status code |
| message | string | * | The error message |
| errors | errors | * | Errors |
| errors[name] | parameter errors | * | Errors on the parameter |
| errors[name][errors][] | array of errors | * | |
| errors[address] | parameter errors | * | Errors on the parameter |
| errors[prices] | parameter errors | * | Errors on the parameter |

FIGURE 17.7. – Documentation de la création avec des erreurs de validation

Ici, nous avons bien deux réponses selon le code de statut mais pour la réponse lors d'une requête invalide, le format n'est pas correct (pas d'attribut `children`, l'attribut `status_code` s'appelle `code`, etc.).

17.3. Étendre `NelmioApiDocBundle`

17.3.1. Pourquoi étendre le bundle ?

Pour corriger les petits manquements de `NelmioApiDocBundle`, nous allons étendre le code de celui-ci. L'objectif n'est pas d'apprendre le code source de ce bundle mais plutôt de maximiser son efficacité en l'adaptant à nos besoins.

i

Il est possible d'obtenir de la documentation en redéfinissant manuellement toutes ces informations manquantes. Mais l'intérêt réel de ce bundle réside dans le fait d'utiliser les composants déjà existants pour générer la documentation automatiquement. N'hésitez donc pas à consulter [la documentation officielle](#) de `NelmioApiDocBundle` pour plus d'informations.

17.3.2. Correction du format de sortie des réponses en erreur

Il n'y a pas de documentation sur comment étendre `NelmioApiDocBundle`. Mais vu que ce bundle est open source, il suffit de relire avec attention son code pour comprendre son fonctionnement.

Il en ressort que pour traiter les informations disponibles dans les attributs `input` et `output` de l'annotation `ApiDoc`, le bundle utilise des parseurs.

Et [la documentation officielle](#) nous explique comment en créer et comment l'utiliser.

Nous allons donc créer un parseur capable de générer les erreurs de validation au même format que `FOSRestBundle`.

Ce code est grandement inspiré du parseur déjà existant ([FormErrorsParser](#)).

```
1 <?php
2 # src/Component/ApiDoc/Parser/FOSRestFormErrorsParser.php
3
4 namespace Component\ApiDoc\Parser;
5
6 use Nelmio\ApiDocBundle\DataTypes;
7 use Nelmio\ApiDocBundle\Parser\ParserInterface;
8 use Nelmio\ApiDocBundle\Parser\PostParserInterface;
9
10 class FOSRestFormErrorsParser implements ParserInterface,
11     PostParserInterface
12 {
13     public function supports(array $item)
14     {
15         return isset($item['fos_rest_form_errors']) &&
16             $item['fos_rest_form_errors'] === true;
```

```

16     }
17
18     public function parse(array $item)
19     {
20         return array();
21     }
22
23
24     public function postParse(array $item, array $parameters)
25     {
26         $params = [];
27
28         // Il faut d'abord désactiver tous les anciens paramètres
           créer par d'autres parseurs avant de reformater
29         foreach ($parameters as $key => $parameter) {
30             $params[$key] = null;
31         }
32
33         $params['code'] = [
34             'dataType' => 'integer',
35             'actualType' => DataTypes::INTEGER,
36             'subType' => null,
37             'required' => false,
38             'description' => 'The status code',
39             'readonly' => true
40         ];
41
42         $params['message'] = [
43             'dataType' => 'string',
44             'actualType' => DataTypes::STRING,
45             'subType' => null,
46             'required' => true,
47             'description' => 'The error message',
48             'default' => 'Validation failed.',
49         ];
50
51         $params['errors'] = [
52             'dataType' => 'errors',
53             'actualType' => DataTypes::MODEL,
54             'subType' => sprintf('%s.FormErrors', $item['class']),
55             'required' => true,
56             'description' => 'List of errors',
57             'readonly' => true,
58             'children' => [
59                 'children' => [
60                     'dataType' => 'List of form fields',
61                     'actualType' => DataTypes::MODEL,
62                     'subType' => sprintf('%s.Children',
63                                     $item['class']),

```

III. Amélioration de l'API REST

```
64         'description' => 'Errors',
65         'readonly' => true,
66         'children' => []
67     ]
68 ]
69 ];
70
71 foreach ($parameters as $name => $parameter) {
72
73     $params['errors']['children']['children']['children'][$name]
74     = $this->doPostParse($parameter, $name, [$name],
75     $item['class']);
76 }
77
78 return $params;
79 }
80
81 protected function doPostParse($parameter, $name, array
82 $propertyPath, $type)
83 {
84     $data = [
85         'dataType' => 'Form field',
86         'actualType' => DataTypes::MODEL,
87         'subType' => sprintf('%s.FieldErrors[%s]', $type,
88         implode('.', $propertyPath)),
89         'required' => true,
90         'description' => 'Field name',
91         'readonly' => true,
92         'children' => [
93             'errors'=> [
94                 'dataType' => 'errors',
95                 'actualType' => DataTypes::COLLECTION,
96                 'subType' => 'string',
97                 'required' => false,
98                 'description' =>
99                 'List of field error messages',
100                'readonly' => true
101            ]
102        ]
103    ];
104
105     if ($parameter['actualType'] == DataTypes::COLLECTION) {
106         $data['children']['children'] = [
107             'dataType' => 'List of embedded forms fields',
108             'actualType' => DataTypes::COLLECTION,
109             'subType' => sprintf('%s.FormErrors',
110             $parameter['subType']),
111             'required' => true,
112             'description' => 'Validation error messages',
113             'readonly' => true,
```

```
107         'children' => [
108             'children' => [
109                 'dataType' => 'Embedded form field',
110                 'actualType' => DataTypes::MODEL,
111                 'subType' => sprintf('%s.Children',
112                     $parameter['subType']),
113                 'required' => true,
114                 'description' => 'List of errors',
115                 'readonly' => true,
116                 'children' => []
117             ]
118         ];
119
120         foreach ($parameter['children'] as $cName =>
121             $cParameter) {
122             $cPropertyPath = array_merge($propertyPath,
123                 [$cName]);
124
125             $data['children']['children']['children']['children']['children']
126             = $this->doPostParse($cParameter, $cName,
127                 $cPropertyPath, $parameter['subType']);
128         }
129     }
130     return $data;
131 }
```

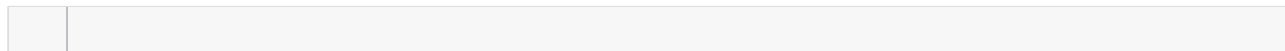
Ce parseur doit toujours être utilisé avec `FormTypeParser` qui apporte l'ensemble des informations issues du formulaire Symfony. Pour l'activer, il faut utiliser l'attribut : `fos_rest_form_errors` (voir la méthode `supports`).

Pour le déclarer en tant que parseur prêt à l'emploi, nous devons créer un service avec le tag `nelmio_api_doc_extractor.parser`.



Tous les parseurs natifs du bundle sont déclarés avec une priorité de 0. En utilisant une priorité de 1, nous nous assurons que notre parseur est toujours appelé en dernier.

Pour utiliser notre parseur, nous allons ajuster l'annotation sur le contrôleur des lieux en utilisant l'attribut `fos_rest_form_errors`.



La réponse pour un formulaire invalide est maintenant correctement formatée.

| 400 - Formulaire invalide | | | |
|-------------------------------------|---------------------|---|------------------------------|
| code | integer | * | The status code |
| message | string | * | The error message |
| errors | errors | * | List of errors |
| errors[children] | List of form fields | * | Errors |
| errors[children][name] | Form field | * | Field name |
| errors[children][name][errors][] | errors | * | List of field error messages |
| errors[children][address] | Form field | * | Field name |
| errors[children][address][errors][] | errors | * | List of field error messages |
| errors[children][prices] | Form field | * | Field name |

FIGURE 17.8. – Documentation de la réponse pour un formulaire invalide

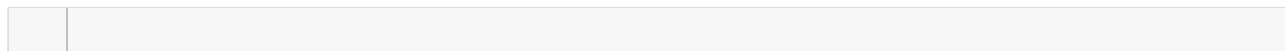
17.4. Le bac à sable

Comme pour *OpenAPI (Swagger RESTful API)*, *NelmioApiDocBundle* propose un bac à sable permettant de tester la documentation. Avant d'utiliser ce bac à sable, nous allons rajouter quelques informations de configuration.

17.4.1. Configuration du bac à sable

La [documentation officielle](#) sur le bac à sable est concise et simple. Les paramètres disponibles sont d'ailleurs assez proches de ceux d'*OpenAPI*.

Voyez donc par vous-même.



Cette configuration est assez explicite et se passe donc de commentaires. En accédant à la documentation avec l'URL <http://rest-api.local/documentation>, nous obtenons :

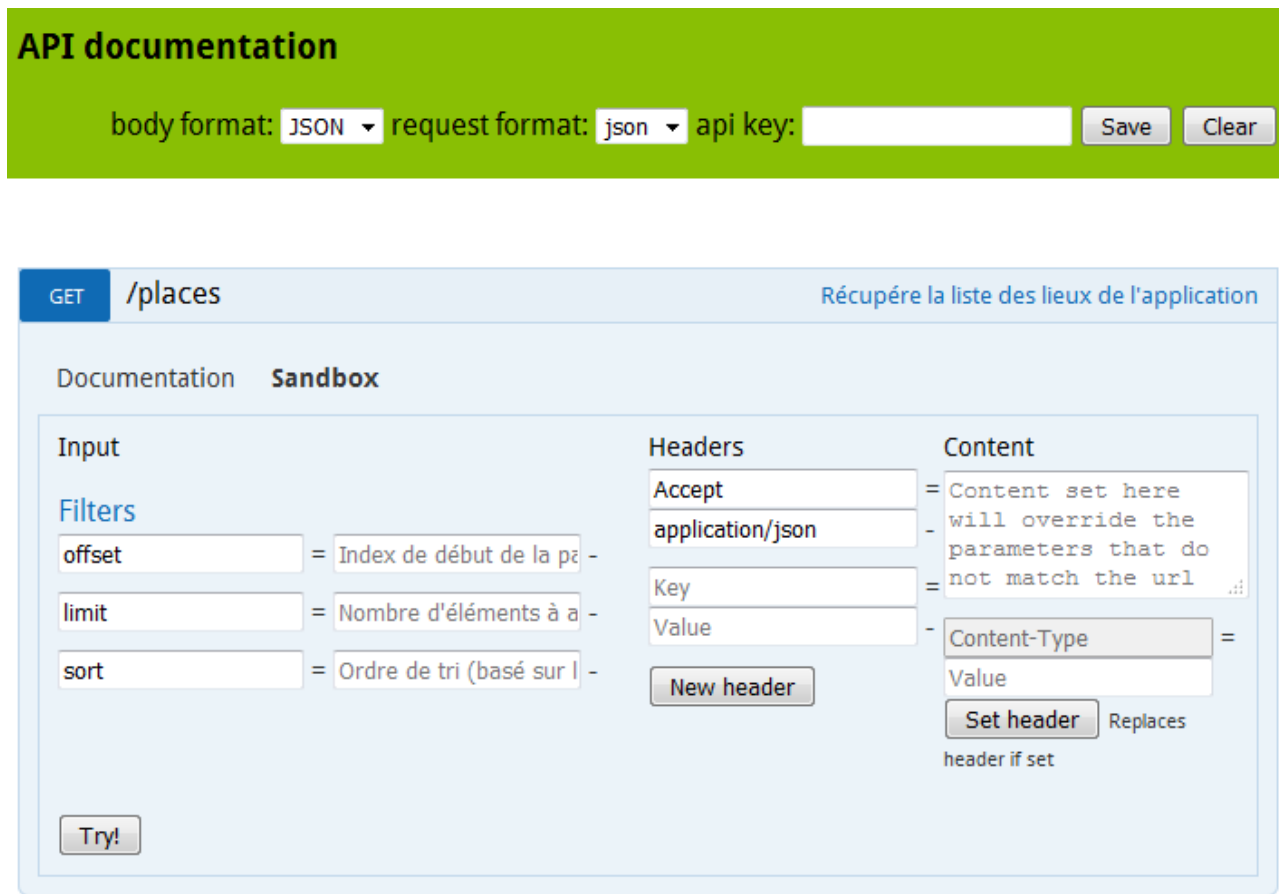
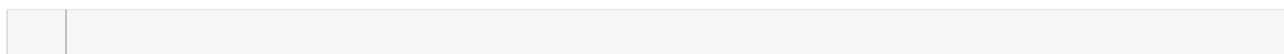


FIGURE 17.9. – Bac à sable de NelmioApiDocBundle

Le bac à sable est disponible en cliquant sur l'onglet `Sandbox`.

17.4.2. Documentation pour la création de token

Avant de tester ce bac à sable, nous allons rajouter de la documentation pour la création de token d'authentification. Cela facilitera grandement nos tests.



Nous pouvons maintenant créer un token depuis le bac à sable.

17.4.3. Tester le bac à sable

Vu que toutes nos méthodes nécessitent une authentification, il faut d'abord créer un token d'authentification. Ce token doit être renseigné dans le formulaire `api_key`.



FIGURE 17.10. – Token renseigné dans le formulaire

Avec la configuration que nous avons mise en place, ce token sera envoyé automatiquement pour toutes nos requêtes.

Maintenant pour récupérer les lieux de l'application, il suffit de cliquer sur le bouton **Try it!**.

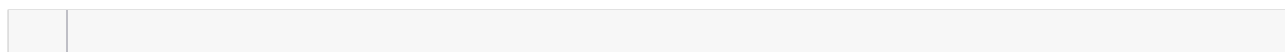


FIGURE 17.11. – Récupération des lieux grâce au bac à sable

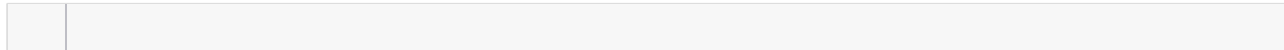
17.5. Générer une documentation compatible OpenAPI

Pour profiter des différents outils disponibles dans l'écosystème de *Swagger*, *NelmioApiDocBundle* propose d'exporter la configuration au format *OpenAPI*.

Pour ce faire, il faut rajouter un attribut `resource` à nos annotations `ApiDoc`. Ensuite, il suffit d'utiliser la commande `php bin/console api:swagger:dump dossier_de_destination`. Voici un exemple de configuration qui remplit ce contrat :

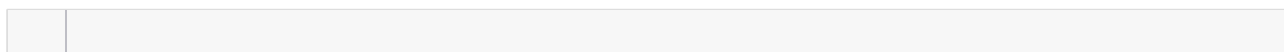


Les métadonnées concernant la documentation peuvent être modifiées en configurant le bundle.



Avec la version 2.13.0, ce bundle génère un fichier **swagger.json** en utilisant [la version 1.2 des spécifications d'OpenAPI](#) [↗](#) alors qu'il existe une version 2.0. Le fichier généré ne sera donc pas à jour même si dans la configuration nous mettons 2.0 comme valeur de l'attribut `swagger_version`.

En exécutant la commande :



Les fichiers ainsi générés dans le dossier **web/swagger** peuvent être exploités par tous les outils compatibles avec *OpenAPI*.

Pour les tester, il suffit d'éditer le fichier **web/swagger-ui/dist/index.html** et de remplacer la ligne `var url = "/swagger.json";` par `var url = "/swagger/auth-tokens.json";`.

En accédant à l'URL <http://rest-api.local/swagger-ui/dist/index.html> [↗](#), la documentation générée s'affiche.

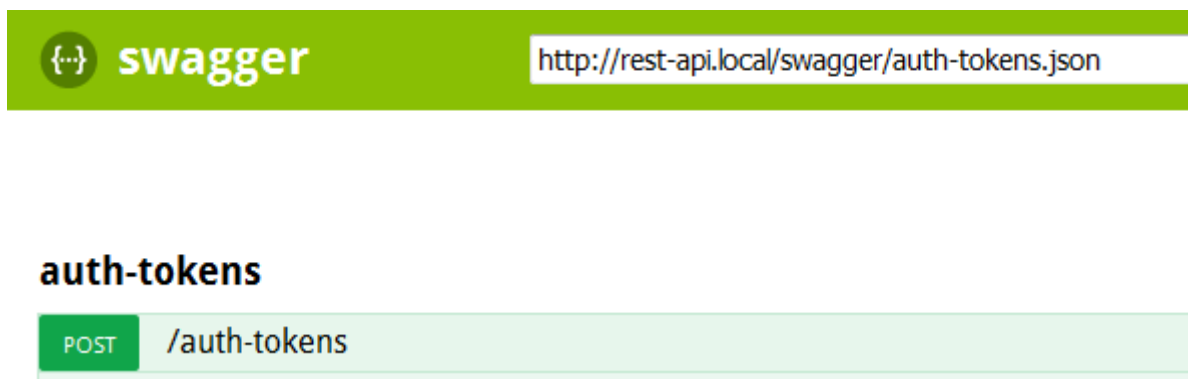


FIGURE 17.12. – Documentation OpenAPI générée par NelmioApiDocBundle

ApiDocBundle supporte les différents bundles de Symfony et le tout permet d'avoir un ensemble harmonieux et facilite les développements.

L'un des problèmes les plus communs lorsque nous écrivons une documentation est de la maintenir à jour. Avec une documentation proche du code, il est maintenant très facile de la corriger en même temps que le code évolue.

En effet, un utilisant les annotations de *FOSRestBundle*, les formulaires de Symfony et les fichiers de sérialisation de *JMSSerializerBundle*, nous avons la garantie que la documentation est toujours à jour avec notre code.

Il ne reste plus qu'à tout mettre en production !

18. FAQ

Dans cette section, nous allons aborder quelques points intéressants qui reviennent souvent dans les questions concernant ce cours.

Les points abordés n'ont pas de relation particulière et peuvent donc être lu dans n'importe quel ordre.

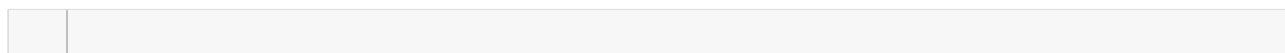
18.1. Comment générer des pages HTML depuis l'application Symfony 3?

La configuration présentée durant ce cours implique que toute l'application ne génère que des réponses en JSON ou en XML. Cependant, il peut arriver qu'une même application puisse servir des réponses en JSON, en HTML voire en CSV.

Pour ce faire nous pouvons utiliser deux options que propose *FOSRestBundle*.

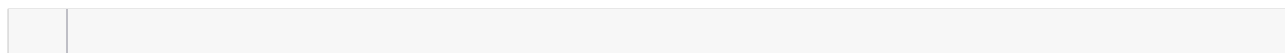
18.1.1. Utiliser plusieurs règles dans le `format_listener`

Dans notre fichier de configuration, nous avons :



Listing 1 – configuration du `format_listener`

Pour générer une page HTML, nous pouvons rajouter une nouvelle règle dans la clé `format_listener.rules`. L'ordre de déclaration étant important, il faut toujours déclarer les règles les plus spécifiques en premier.



Listing 2 – Une nouvelle route pour créer du HTML

Avec cette configuration, toutes les URLs commençant par `/route/json` renverront du JSON. Par contre, si l'URL commence par `/route` (mais sans la partie `/json`, `/route/other` par exemple) les réponses seront en HTML.

Si nous avons inversé ces deux règles, toutes les URLs `/route/json` renverraient aussi du HTML car l'expression régulière `^/route` englobe aussi `^/route/json`.

i

Comme pour les formats JSON et XML, la génération de réponse au format HTML est déjà supporté par défaut. Mais en rajoutant la clé `templating_formats.html`, la configuration est plus lisible. De plus, nous utilisons `templating_formats` au lieu de `formats` car pour les pages HTML, nous aurons besoin d'un *template* pour les afficher.

Nous pouvons rajouter autant de règles que nous voulons mais cela peut rapidement montrer ses limites. Nous avons ainsi la possibilité d'utiliser un autre système plus efficace pour isoler la partie API et la partie IHM² de son application.

18.1.2. Configurer le `zone_listener`

Il existe un *listener* de *FOSRestBundle* que nous n'avons pas abordé qui permet d'isoler la partie API d'une application de manière très simple : le `zone_listener`.

Le `zone_listener` est un *listener* qui nous permet de désactiver toutes les fonctionnalités de *FOSRestBundle* pour un ensemble d'URLs.

Ajoutons d'abord un préfixe `/api` à toutes les routes de notre API. La déclaration des routes pourrait ressembler à :

Listing 3 – Ajout d'un préfixe

Tous les appels d'API restent identiques mais sont maintenant préfixés.

La configuration de *FOSRestBundle* devient maintenant :

Listing 4 – Configuration du `zone_listener`

La partie `zone` permet d'activer le bundle que pour les routes commençant par `/api`. Ainsi, toute requête en dehors de cette *zone* sera gérée nativement par Symfony. Nous pouvons ainsi faire cohabiter notre API et une IHM complète sans soucis.

En utilisant ce système de zone, il ne faut pas oublier de reconfigurer toute la partie liée au pare-feu de Symfony et à notre système de sécurité pour prendre en compte le préfixe.

La configuration finale serait donc :

La clé `pattern` prend en compte le préfixe.

Listing 5 – Prise en compte du `pattern` dans nos contrôleurs

L'URL dans `targetUrl` contient maintenant notre préfixe `/api`.

Il est quand même utile de souligner qu'il est préférable d'utiliser une application à part pour générer ses pages HTML et avoir une application dédiée pour son API. L'intérêt de REST est d'avoir une architecture orientée service et donc de séparer les différents composants.

2. Interface Homme Machine, dans notre cas la page qui s'affiche dans le navigateur.

18.2. Comment autoriser l'accès à certaines urls avec notre système de sécurité?

Comme vous l'avez sans doute remarqué, une fois le système de sécurité est activé, seule la requête de création de token est autorisée. Mais dans les faits, il est assez courant d'avoir plusieurs appels d'API accessible sans authentification.

?

Par exemple, comment autoriser les utilisateurs à s'inscrire ?

Actuellement cela est impossible mais nous pouvons corriger le tir très facilement. Pour rappel, l'authentification est géré par la classe `AuthTokenAuthenticator` dont voici un extrait du code :

Listing 6 – Le nouvel Authenticator

Pour vérifier si une requête a été faite sur une certaine URL, la méthode `checkRequestPath` peut utiliser une route comme nous l'avons spécifié dans l'extrait de code ci-dessus, mais aussi le nom d'une route.

Le code peut donc être simplifié en utilisant directement le nom pour la route `auth-tokens` : `post_auth_tokens`.

Pour obtenir la liste des routes, nous pouvons utiliser la commande `php bin/console debug:router`.

Avec le système de nommage de `FOSRestBundle`, nous avons des noms simples et **surtout qui décrivent aussi le verbe HTTP associé** à la route. Dès lors pour autoriser une action, nous pouvons nous baser uniquement sur le nom de la route correspondante (le verbe HTTP est vérifiée indirectement).

Ainsi pour autoriser la création d'utilisateurs et de tokens d'authentification, nous pouvons simplement utiliser respectivement les routes : `post_users` et `post_auth_tokens`.

Le code peut devenir :

Le service `HttpUtils` étant maintenant inutile, nous pouvons même le retirer de la configuration des services.

Listing 7 – Désactivation du service HTTPUtils

Bien sur, libre à vous de gérer la liste de routes autorisées comme bon vous semble (en injectant un paramètre configurable dans le service, en ayant une liste dans une variable statique, etc.).

Nous avons pu voir tout au long de ce cours que les contraintes REST permettent de mettre en

III. Amélioration de l'API REST

place une API uniforme et facile à prendre en main. La mise en œuvre de ces contraintes offre un ensemble d'avantages et le framework Symfony dispose d'outils suffisamment matures pour aider dans les développements.

Ce cours bien qu'étant assez long n'aborde pas tous les concepts de REST ni toutes les fonctionnalités qu'apportent *FOSRestBundle* et les différents bundles utilisés. Son objectif est de présenter de manière succincte l'essentiel des notions à comprendre pour pouvoir développer une API RESTful et l'améliorer en toute autonomie.

Le style d'architecture REST ne s'occupe pas des détails d'implémentations mais plutôt du rôle de chaque composant de notre application.

N'hésitez surtout pas enrichir l'API et à explorer les documentations officielles des différents outils abordés pour mieux cerner tout ce qu'ils peuvent vous apporter.

Liste des abréviations

MIME Multipurpose Internet Mail Extensions. 131, 132, 134, 202