YoruNoHikage

Zeste de savoir

La théorie REST, RESTful et HATEOAS

12 août 2019

Table des matières

*[REST] : Representational State Transfer

REST est un style d'architecture défini dans la thèse de Roy Fielding dans les années 2000, ce n'est donc ni un protocole ni un format. Les implémentations sont donc multiples et différentes. Cependant, on retrouve souvent le principe dans les API HTTP, comme c'est le cas de GitHub et Twitter par exemple.

Lançons-nous à pieds joints dans un monde merveilleux !

1. Les contraintes

L'auteur définit un certain nombre de contraintes à respecter afin de devenir "REST Compliant" autrement dit comment se conformer à cette architecture. On dit de REST qu'il est sans état.

2. Les contraintes

1. Le serveur et le client sont indépendants. L'interface utilisateur est situé côté client (une application mobile par exemple) et le stockage est située côté serveur (une base de données).
2. Contrairement à l'accès web classique, aucune variable de session ou autre état volatile ne doit être enregistré côté serveur. Chaque requête vers le serveur est donc indépendante.
3. Mise en cache : le serveur indique au client s'il peut mettre en cache les données qu'il reçoit. Cela permet d'éviter des requêtes inutiles et ainsi préserver la bande passante.
4. Une interface uniforme.

   a) On accède à chaque ressource de façon unique. Il n'y a qu'une seule façon d'y accéder.

   b) Les ressources sont manipulées via des représentations définies, elles sont accompagnées de données permettant sa compréhension (vous comprendrez mieux dans l'exemple).
   c) Auto-description. L'encodage, par exemple, est précisé de façon claire ainsi le client peut comprendre le document et appeler le service correspondant à cet encodage.
   d) Hypermédia comme moteur d'application (HATEOAS) : la ressource indique comment accéder aux états suivants (suppression, édition etc...), le client peut ainsi connaître quelles actions sont possibles sur la ressource en l'obtenant

5. Hiérarchie par couche : les ressources sont individuelles. On peut imaginer que nos ressources embarquent des ressources provenant de serveurs distants ou de mise en cache par des serveurs intermédiaires.
6. (Facultatif) Exécution de scripts côté client obtenus par le serveur. Cela permet de rendre le client plus léger et plus générique.

? Qu'est-ce que ça veut dire tout ça concrètement ?

Reprenons ces contraintes avec un l'exemple de l'API de Twitter et vulgarisons son utilisation :

1. L'application Twitter de votre téléphone correspond à un client. Elle est indépendante des serveurs de Twitter. Vous pouvez naviguer sur votre application, les requêtes sont envoyées et les réponses sont traitées par le client afin d'être affichées.
2. Lorsque vous postez un tweet, toutes les informations envoyées permettent de vous identifier. Dans le cadre de Twitter, l'authentification se fait par l'intermédiaire d'une clé d'API.
3. Les tweets sont gardés sur l'application jusqu'à ce que le cache soit vidé.
4. a) L'accès au tweet se fait via l'URI : /statuses/show.json ?id=210462857140252672
   b) Le tweet est encapsulé dans un nuage de méta-données, il n'y a pas que la ressource
   c) La langue du tweet est précisée.
   d) Twitter ne suit pas REST à la lettre et n'intègre pas de lien hypermedia, c'est pour cela que son API est documentée. Cet exemple sera illustré dans la dernière partie de ce tutoriel.
5. Le fonctionnement de ce point est plus interne, on ne peut donc pas savoir mais Twitter doit sûrement faire de la mise en cache et du load-balancing.
6. Aucun exemple ne me vient en tête. (Si quelqu'un connaît une API qui fait ça, je serai ravi de la partager)

3. Le modèle de maturité de Richardson

Développé par Leonard Richardson, ce modèle permet de découper les contraintes de REST en 3 étapes principales à suivre afin de mettre en application la théorie de REST en tant que service web. Et voici pour vous, si vous ne l'avez pas déjà vu quelque part, un schéma très important que l'on retrouve souvent :

FIGURE 3. – Le modèle de maturité de Richardson mis en image par Martin Fowler

Les principes de REST ne sont pas toujours respectés mais lorsque l'on compare la théorie à la pratique, il faut savoir faire quelques concessions et vous serez sûrement amenés à faire des choix divergeant des standards.

4. Un exemple d'API : prendre un rendez-vous chez le médecin

J'ai choisi de reprendre les exemples fournis par Martin Fowler et de les adapter en JSON. J'apprécie davantage ce format et les APIs l'utilisent souvent bien qu'il ait des limites (on verra ça tout à l'heure).

Considérons que l'on veut prendre un RDV chez le médecin. Nous sommes le client et le secrétariat que l'on appelle est le serveur.

5. Niveau 0

Je souhaite donc prendre un rendez-vous le 21 octobre 2015 avec le médecin Doc. Il s'agit dans ce niveau d'interroger le serveur à partir d'un seul point d'entrée. On spécifie les actions ou les ressources à récupérer dans l'objet que l'on envoie.

```
POST /cabinetMedecin HTTP/1.1
Content-Type: application/json
# en-têtes HTTP
...

{
  "prendreRDV": {
    "date" : "2015-10-21",
    "medecin_id" : "doc"
  }
}
```

J'obtiens, par la suite, les créneaux disponibles dans la journée :

```
HTTP/1.1 200 OK
Content-Type: application/json
```

```
...

{
  "creneaux": [
    {
      "debut": "14h00",
      "fin": "14h50",
      "medecin": { "id": "doc", "autre_propriete": "valeur" }
    },
    {
      "debut": "16h00",
      "fin": "16h50",
      "medecin": { "id": "doc", "autre_propriete": "valeur" }
    }
  ]
}
```

i Je simplifie en mettant les dates/heures en chaîne de caractère, on ne va pas s'embêter avec des dates à rallonge

Maintenant que l'on connaît les créneaux libres, on veut pouvoir le réserver ! On renvoie une requête au serveur avec l'action à effectuer et les informations dont il a besoin.

```
POST /cabinetMedecin HTTP/1.1
Content-Type: application/json
...

{
  "demandeRDV": {
    "creneau": {
      "medecin_id": "doc",
      "debut": "14h00",
      "fin": "14h50"
    },
    "patient": { "id": "marty" }
  }
}
```

Dans le cas d'une réussite, j'obtiens cette réponse accompagnée du contenu que j'ai envoyé :

```
HTTP/1.1 200 OK
Content-Type: application/json
...

```

```
{
  "RDV": {
    "creneau": ...,
    "patient": ...
  }
}
```

Et sinon j'obtiens une réponse négative :

```
HTTP/1.1 200 OK
Content-Type: application/json
...

{
  "RDVImpossible": {
    "creneau": ...,
    "patient": ...,
    "erreur": "Ce créneau n'est pas disponible"
  }
}
```

Vous commencez déjà à voir ce qui cloche n'est-ce pas ? Non ? Le 200 OK pour une erreur, bizarre, non ?

Voyons comment s'élever vers la gloire de REST et ainsi améliorer notre architecture !

6. Niveau 1 : Les ressources, diviser pour mieux régner !

Plutôt que d'interroger un service unique, il est plus intéressant de parler de ressources ! Pour faire simple, on va directement appeler le médecin avec qui on veut prendre un RDV ! (c'est bête s'il est très occupé mais au moins on gène pas le secrétariat) La requête ne contiendra plus le médecin à qui s'adresser puisque nous l'avons appelé directement pour lui demander.

```
POST /medecins/doc HTTP/1.1
Content-Type: application/json
...

{ "prendreRDV": { "date" : "2015-10-21" } }
```

La réponse ne change que très peu. Chaque ressource étant accessible, elle porte un identifiant afin de la retrouver de façon unique (contrainte 4.1) :

```
{
  "creneaux": [
    {
      "id": "1337",
      ...
    },
    {
      "id": "1645",
      ...
    }
  ]
}
```

On a plus qu'à choisir notre créneau et à le réserver directement :

```
POST /creneaux/1337 HTTP/1.1
...
```

La réponse est identique au niveau précédent.

Ce niveau permet d'apporter une certaine clarté, on agit sur une ressource et non pas sur un service en précisant à chaque fois ce que l'on veut.

7. Niveau 2 : Les verbes HTTP et les codes de retour

Dans les deux premiers niveaux, vous avez dû trouver étrange l'utilisation du POST. On aurait tout aussi bien pu mettre GET, ça aurait été identique (mais sachant que l'on a modifié l'état du serveur, ça aurait vraiment pas été propre). On ne s'est servi de la requête HTTP uniquement pour que le serveur comprenne notre requête mais on a pas vraiment respecté le protocole. Le modèle de Richardson, quant à lui, définit une utilisation des verbes HTTP la plus proche possible du protocole.

GET correspond à une action sûre, elle sera utilisé pour récupérer une ressource. **En aucun cas, il ne doit y avoir un changement d'état sur le serveur.**

- Si la ressource n'est pas supprimée ou modifiée, GET renverra toujours la même chose, on dit qu'il est idempotent.
- La réponse peut être mise en cache par des éléments intermédiaires (souvenez-vous de la règle n°5 : hiérarchie par couches).

```
GET /medecins/doc/creneaux?date=20151021&status=ouvert HTTP/1.1
...
```

Comme vous pouvez le constater, le corps de la requête est vide et les infos sont passés en arguments. Généralement dans les APIs (mais pas toujours), les paramètres servent de filtres (tri, pagination...). Ici on récupère uniquement les créneaux **ouverts** le **21 octobre 2015**. La réponse, quant à elle, ne change en rien et elle ne changera pas tant que la ressource ne sera pas modifiée (première règle de GET).

Maintenant, on veut réserver le créneau. Là encore, la requête ne change pas et c'est bien un POST que l'on utilise.

Le fait d'utiliser POST ou PUT afin de créer et/ou de mettre à jour une ressource est sujet à débat. Mais comme le pointe cette réponse , le standard veut que POST soit utilisé lorsque l'on envoie une nouvelle sous-ressource ne connaissant pas son URI et PUT à l'inverse, la ressource étant modifiée si elle existe déjà ! Pour prendre un exemple, ceci `POST /creneaux/1337 HTTP/1.1` et `PUT /creneaux/1337/rdv HTTP/1.1` créeront la ressource rdv dans les deux cas sur le créneau 1337.
Pour prendre un exemple plus parlant, imaginons que vous deviez créer un commentaire sur une news, vous feriez `POST /news/ma-super-news/comments` par exemple, mais `PUT /news/ma-super-news/comments/12345` fonctionnerait également, bien qu'il soit plus approprié pour l'édition (on ne connaît rarement l'identifiant à l'avance).

La réponse renverra alors un 201 Created, les codes de retour étant à présent utilisés, **accompagnée de l'en-tête HTTP Location avec l'URI vers la ressource**. Il peut être intéressant aussi de l'embarquer dans le corps de la réponse en supplément afin d'éviter une nouvelle requête pour le client.

Mais si quelqu'un réserve en même temps ? On fait quoi ?

Eh bien, on le dit tout simplement : Attention, il y a un conflit sur ce créneau, il est déjà pris ! Ce qui donne ceci dans la réponse :

```
HTTP/1.1 409 Conflict
Content-Type: application/json
...

{
  "creneau": {
     ...
  }
}
```

Si l'on souhaite supprimer une ressource, il suffit d'envoyer un DELETE de cette façon :

```
DELETE /creneaux/1337/rdv HTTP/1.1
# En-tête d'autorisation et d'identification :
# Utile pour restreindre la suppression à l'auteur du RDV ou au
    médecin
# Il ne faudrait pas que n'importe qui puisse supprimer votre RDV !
```

et la réponse pourra contenir un `200 OK` (avec un corps de réponse fourni), un `202 Accepted` si l'action n'est pas encore effectuée et qu'elle est attente et un `204 No Content` si tout s'est déroulé correctement.

8. Niveau 3 : Les liens hypermédia (HATEOAS)

? HATEOAS, qu'est-ce que c'est ?

HATEOAS, pour Hypermedia As The Engine Of Application State, est la contrainte (4.4) qui est la plus présente sur le web mais malheureusement pas souvent dans les APIs. Arrivé à ce niveau, on reconnaît évidemment que le web est bien ancré sur le principe de REST, à quelques exceptions près. Lorsque l'on obtient une ressource, ou une page sur le web, il est très important de la lier à d'autres ressources via des liens. C'est aussi bête que ça.

Pour prendre un exemple, lorsque vous désirez accéder à un tuto sur Zeste de Savoir, vous n'allez pas directement sur le tuto, vous passez au moins par la page d'accueil qui vous indique où vous rendre pour obtenir le tuto. Et c'est ce à quoi servent les contrôles hypermédia dans une API, à permettre la navigation entre les différentes ressources.

En rajoutant ces liens, si vous avez un client assez intelligent, il saura s'adapter aux changements de l'API, ce qui en fait un atout car les APIs doivent évoluer (les ressources peuvent changer de nom et les actions sont amenées à évoluer).

8.1. Un exemple concret

Il existe actuellement plusieurs formats standardisés ou non qui sont utilisés pour lier des données en JSON bien qu'il ne soit pas un format adapté. On retrouve HAL , JSON-LD ou encore JSON :API . Tous ces formats ont leurs avantages et inconvénients. C'est à vous de choisir et peut-être bien que vous ne voulez pas de JSON du tout. Je vous conseille d'aller faire un tour sur les différents sites pour choisir celui que vous préférez ou qui vous semble le plus pertinent. D'ailleurs, changeons les réponses en un langage plus approprié : xml.

Reprenons la réponse de notre premier GET et ajoutons-y des liens :

```xml
<creneaux>
    <creneau>
        <id>1337</id>
        ...
        <link rel="self"
            href="http://api.example.com/creneaux/1337" />
        <link rel="rdv"
            href="http://api.example.com/creneaux/1337/rdv" />
    </creneau>
    <creneau>
        <id>1645</id>
        ...
        <link rel="self"
            href="http://api.example.com/creneaux/1645" />
        <link rel="rdv"
            href="http://api.example.com/creneaux/1645/rdv" />
    </creneau>

    <link rel="self"
        href="http://api.example.com/medecins/doc/creneaux?date=20151021&statu
        />
</creneaux>
```

Comme vous pouvez le voir, l'ajout de lien permet de connaître les actions possibles sur les ressources ainsi que les relations entre ressources. Imaginez que nous demandions tous les créneaux du médecin, y compris ceux qui sont déjà réservés, le `<link rel="rdv" ... >` ne sera pas présent dans ces derniers puisque l'action n'est pas possible. Ainsi on peut découvrir l'API sur le moment. La documentation n'en deviens que moins utile.

Ce système comporte quand même des limites, on ne peut pas savoir quelle méthode HTTP utiliser sur les ressources mais il s'agit d'une fonctionnalité qui peut être très utile lors du parcours de l'API.

Si vous voulez tester en live, vous pouvez vous rendre sur l'API GitHub , vous verrez que vous pouvez vous déplacer de ressources en ressources sans avoir besoin d'aller sur la doc

Voilà qui clôt le modèle de maturité de Richardson. Il faut savoir rester pragmatique et faire des choix en fonction de ses besoins. Il n'y a pas forcément d'intérêt à atteindre le niveau 3 si vous ne comptez pas interagir avec d'autres systèmes de données intelligents.

Pour résumer :

- Le niveau 1 permet de "diviser pour mieux régner".
- Le niveau 2 introduit des verbes afin de mieux gérer les actions similaires.
- Le niveau 3 introduit la possibilité de découvrir l'API, la permettant d'être auto-documentée.

? Mais, RESTful, c'est quoi en fait ?

Il s'agit d'un terme générique très utilisé pour indiquer le parfait respect de REST. Bien entendu, beaucoup d'APIs se considérant REST ou RESTful ne le sont pas et ainsi, il a été décidé d'utiliser le terme HATEOAS pour parler d'une API respectant le niveau 3 qui est un des points les plus importants de REST.

9. Mémo des codes et réponses HTTP

Voilà un petit mémo des différents codes HTTP utiles pour les API. Je rajouterai peut-être quelques cas de figures qui peuvent être intéressants en pratique. N'hésitez pas à m'envoyer des utilisations pratiques de ces codes.

Code	Message	Signification
200	**OK**	**Succès de réponse pour toutes méthodes sauf la création avec POST.**
201	Created	Réponse à un POST qui crée une ressource. Doit être **obligatoirement** accompagné du header `Location` avec le lien vers la nouvelle ressource. On peut embarquer la nouvelle ressource pour éviter au client de refaire une nouvelle requête.
204	No Content	Une requête réussie qui ne renvoie aucun contenu (comme un DELETE par exemple).
304	Not Modified	Utilisé avec la gestion d'un cache.
400	Bad Request	La requête a échouée car le contenu ne peut pas être compris (un contenu qui ne peut être parsé par exemple).
401	Unauthorized	Lorsqu'une authentification a échouée. Utile pour afficher une popup quand l'authentification se fait par un navigateur.

403	Forbidden	L'authentification est correcte mais l'utilisateur ne peut pas accéder à la ressource.
404	Not Found	La ressource n'a pas pu être trouvée.
405	Method Not Allowed	Quand une méthode non autorisée a été utilisée par l'utilisateur.
410	Gone	La ressource n'est plus disponible. Utile pour les vieilles versions de l'API.
415	Unsupported Media Type	Si le Content-Type est incorrect.
422	Unprocessable Entity	Utilisé pour les erreurs de validation.
429	Too Many Requests	Utilisé lorsque la limite de requêtes autorisées a été dépassée.
500	Internal Error	Un message générique pour indiquer qu'il y a eu un problème mais qu'il ne vient pas de l'utilisateur.
503	Service Unavailable	Le service est down ou en maintenance. Status temporaire, en général.

Voilà, maintenant que vous avez tout compris sur REST, vous allez pouvoir bien structurer votre application ! Ce tuto ne concernait pas vraiment la mise en application au niveau code, mais il est toujours intéressant de savoir comment les choses sont faites et décrites afin de pouvoir les respecter par la suite.

Sources :

— REST sur Wikipédia
— http ://blog.xebia.fr/2010/06/25/rest-richardson-maturity-model/
— http ://martinfowler.com/articles/richardsonMaturityModel.html
— http ://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html
— REST APIs with Symfony2 : The Right Way - William Durand
— Best Practices for Designing a Pragmatic RESTful API - Vinay Sahni

Merci à Dominus Carnufex pour les corrections.

Liste des abréviations

HAL Hypertext Application Language. 8

HATEOAS Hypermedia As The Engine Of Application State. 1, 2, 8–10

JSON-LD JavaScript Object Notation for Linked Data. 8

REST Representational State Transfer. 1–3, 5, 8, 10, 11

URI Uniform Resource Identifier. 2, 7