Modèle de maturité de Rechardson

Le modèle de maturité de Richardson *(Richardson Maturity Model)* est un cadre qui décrit l'évolution des API Web au fil du temps. Il a été développé par Leonard Richardson en 2008 et porte son nom.

Le modèle se compose de quatre niveaux, chaque niveau représentant un degré plus élevé de sophistication et de complexité dans la conception et l'implémentation des APIs Web. Ce modèle définit quatre étapes qui introduisent progressivement les principaux éléments de REST *(Ressources ; Verbes et Codes retours HTTP ; Contrôles hypermédia)* pour passer d'un modèle RPC sur HTTP à un modèle RESTFul.

Les quatre niveaux sont illustrés dans la figure suivante [1] :

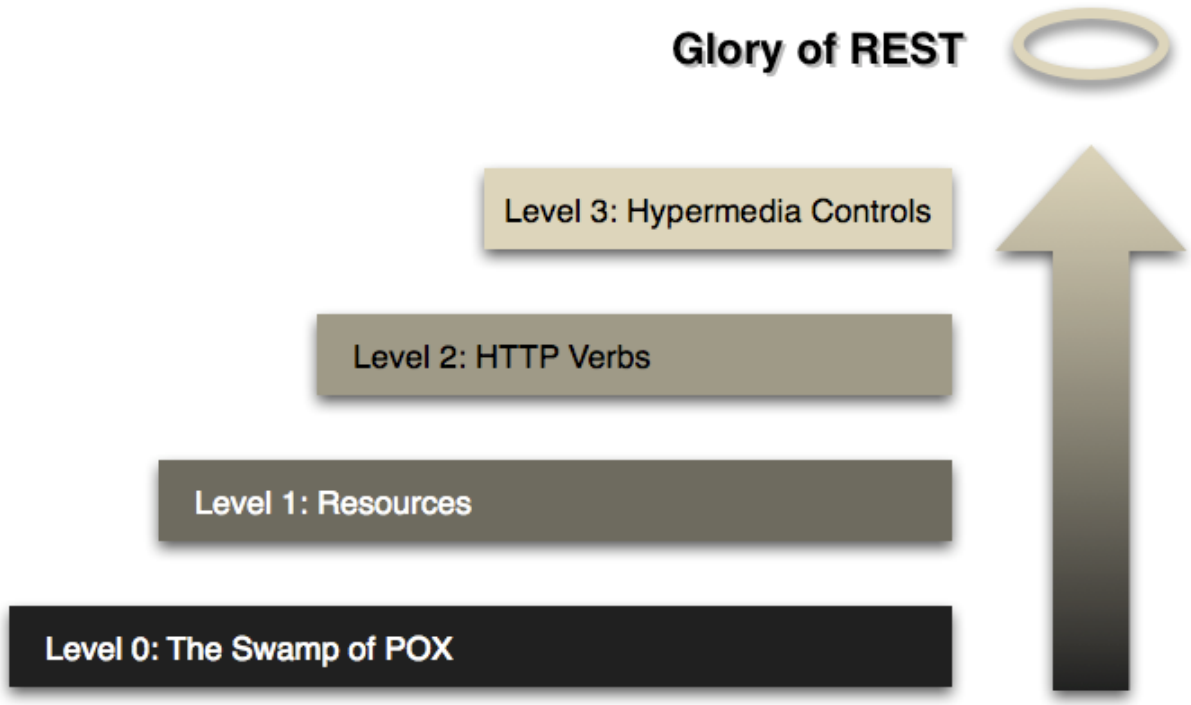

Figure 1 : Les étapes vers une API REST [1]

Niveau 0 - Le RPC sur HTTP en POX *(Plain Old XML)*:

À ce niveau, l'API est essentiellement une collection de pages Web qui renvoient des documents XML statiques. Ici, nous ne pouvons pas réellement parler de REST, nous utilisons HTTP comme moyen de transport pour interagir à distance avec un « service » (Service SOAP par exemple => prochain chapitre). A cette étape, Toutes les requêtes

sont envoyées vers le même endpoint (la même URI) : /appointmentService. Elles sont complètement décrites dans le flux XML envoyé.

Prenant l'exemple proposé par Martin Fowler (réserver un rendez-vous chez le médecin) [1], [2]. Dans cet exemple, la réservation se fait de la façon suivante [1], [2]:

Une première requête est envoyée pour obtenir les créneaux disponibles à une date donnée pour docteur « mjones » :

```
POST /appointmentService HTTP/1.1
[various other headers]

<openSlotRequest date = "2023-05-04" doctor = "mjones"/>
```

Le serveur retourne une liste de créneaux :

```
HTTP/1.1 200 OK
[various headers]


<openSlotList>
  <slot start = "14 :00" end = "14 :50">
    <doctor id = "mjones"/>
  </slot>
  <slot start = "16 :00" end = "16 :50">
    <doctor id = "mjones"/>
  </slot>
</openSlotList>
```

Pour réserver un des créneaux de la liste, une deuxième requête est envoyée *(sur le même endpoint)*:

```
POST /appointmentService HTTP/1.1
[various other headers]


<appointmentRequest>
  <slot doctor = "mjones" start = "14:00" end = "14:50"/>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Si la demande aboutit, le serveur retourne un rendez-vous :

```
HTTP/1.1 200 OK
[various headers]


<appointment>
  <slot doctor = "mjones" start = "14:00" end = "14:50"/>
  <patient id = "jsmith"/>
</appointment>
```

En cas problème, le serveur retourne un message d'erreur :

```
HTTP/1.1 200 OK
[various headers]

  <appointmentRequestFailure>
  <slot doctor = "mjones" start = "14:00" end = "14:50"/>
  <patient id = "jsmith"/>
  <reason>Slot not available</reason>
</appointmentRequestFailure>
```

L'objectif de cette étape est de mettre un système RPC simple. C'est pour cette raison dans la dernière réponse de cet exemple en trouve le code « 200 ok », alors que la réponse représente un message d'erreur.

Niveau 0 - Le RPC sur HTTP en POX *(Plain Old XML)*:

Le niveau 1 : l'utilisation de ressources différenciées

Puisque REST est un modèle d'architecture basé sur la manipulation de ressources *(tout est ressource)*, iinsi, plutôt que de présenter toutes les requêtes à un seul endpoint de service, nous commençons maintenant à manipuler des ressources individuelles.

Donc, pour notre requête initiale, pour demander les créneaux disponibles pour docteur « mjones », nous spécifions un URI d'une ressource de type « docteurs » :

```
POST /doctors/mjones HTTP/1.1
[various other headers]
```

La réponse porte les mêmes informations de base, mais chaque créneau est maintenant une ressource qui peut être adressée individuellement (avec identifiant).

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "14 :00" end = "14:50"/>
  <slot id = "5678" doctor = "mjones" start = "16 :00" end = "16:50"/>
</openSlotList>
```

La requête de réservation d'un créneau se fait maintenant sur la ressource spécifique :

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

La réponse renvoyée par le serveur est la même qu'au niveau 0.

L'introduction des ressources à ce niveau nous permet de gérer la complexité du système et de lui diviser en plusieurs ressources.

Le niveau 2 : L'utilisation des verbes et des codes retours HTTP

Cette étape consiste à utiliser des verbes (méthodes) et des codes retours HTTP. Aux niveaux 0 et 1, nous avons utilisé les verbes HTTP POST pour toutes les interactions. À ces niveaux, cela ne fait pas beaucoup de différence, ils sont tous deux utilisés comme des mécanismes de tunnel. Le niveau 2 s'en éloigne, en utilisant les verbes HTTP aussi près que possible de la façon dont ils sont utilisés dans HTTP lui-même (sémantique HTTP).

Pour la liste des créneaux, cela signifie que nous voulons utiliser GET.

```
GET /doctors/mjones/slots?date=20230504&status=open  HTTP/1.1
Host: doctors.com
```

La réponse est la même qu'avec le verbe POST :

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "14 :00" end = "14:50"/>
  <slot id = "5678" doctor = "mjones" start = "16 :00" end = "16:50"/>
</openSlotList>
```

Au niveau 2, l'utilisation de GET pour une demande comme celle-ci est cruciale. HTTP définit GET comme une opération sûre, c'est-à-dire qu'il n'apporte aucune modification significative à l'état de quoi que ce soit. Cela nous permet d'invoquer les GET en toute sécurité un certain nombre de fois dans n'importe quel ordre et d'obtenir les mêmes résultats à chaque fois (sauf bien sûr si l'état de la ressource a été modifié par ailleurs). Une conséquence importante de ceci est qu'il permet à n'importe quel participant (nœud intermédiaire) dans le routage des requêtes d'utiliser la **mise en cache**, qui est un élément clé dans la réalisation du web (particulièrement REST). Le protocole HTTP inclut diverses mesures pour supporter la mise en cache, qui peuvent être utilisées par tous les participants à la communication (voir le chapitre précédent).

Donc, pour réserver un créneau, nous avons besoin d'un verbe HTTP qui change d'état, (POST, PUT, ..). Nous allons utiliser POST.

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Les réponses retournées par le serveur sont les mêmes que le niveau 1, sauf qu'il y a une autre différence significative dans la façon dont le service à distance répond. Si tout se passe bien, le service répond avec un code de réponse de 201 pour indiquer qu'une nouvelle ressource a été créée.

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "14 :00" end = "14 :50"/>
  <patient id = "jsmith"/>
</appointment>
```

La réponse 201 comprend un attribut de localisation avec une URI que le client peut utiliser pour GET l'état actuel de cette ressource à l'avenir. La réponse ici comprend également une représentation de cette ressource pour épargner au client un appel supplémentaire en ce moment.

Il y a une autre différence s'i l y a un problème, comme si quelqu'un d'autre réservait la session.

```
HTTP/1.1 409 Conflict
[various headers]

<openSlotList>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

La partie importante de cette réponse est l'utilisation d'un code de réponse HTTP pour indiquer que quelque chose a mal tourné. Dans ce cas, un 409 semble un bon choix pour indiquer que quelqu'un d'autre a déjà mis à jour la ressource d'une manière incompatible. Plutôt que d'utiliser un code de retour de 200 mais incluant une réponse d'erreur, au niveau 2 nous utilisons explicitement une sorte de réponse d'erreur comme ceci. C'est au concepteur du système de décider quels codes utiliser, mais il devrait y avoir une réponse non 2xx si une erreur survient. Le niveau 2 introduit l'utilisation de verbes HTTP et de codes de retours HTTP.

Il y a une incohérence ici. Les partisans de REST parlent d'utiliser tous les verbes HTTP. Mais pratiquement, PUT ou DELETE sont très peu utilisées pour plusieurs raisons : par exemple des pare-feux qui peuvent être configurés pour ne pas laisser ces opérations PUT et DELETE.

Les éléments clés qui sont soutenus par l'existence du Web sont la forte séparation entre les opérations sécurisées (par exemple GET) et non sécurisées, ainsi que l'utilisation de codes d'état pour aider à communiquer les types d'erreurs rencontrées.

Le niveau 3 : L'utilisation des contrôles hypermédia

Le dernier niveau introduit la notion de HATEOAS (Hypertext As The Engine Of Application State). Le principe ici est de fournir les transitions possibles vers les états suivants par des liens hypermédia. Par exemple, il aborde la question de savoir comment passer d'une liste de créneaux ouverts à savoir quoi faire pour prendre rendez-vous.

Nous commençons avec le même GET initial que nous avons envoyé au niveau 2 :

```
GET /doctors/mjones/slots?date=20230504&status=open  HTTP/1.1
Host: doctors.com
```

Mais la réponse a un nouvel élément :

```
HTTP/1.1 200 OK
[various headers]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "14 :00" end = "14 :50">
    <link rel = "/linkrels/slot/book"
          uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "16 :00" end = "16 :50">
    <link rel = "/linkrels/slot/book"
          uri = "/slots/5678"/>
  </slot>
</openSlotList>
```

Chaque créneau a maintenant un élément de lien qui contient une URI pour nous dire comment prendre rendez-vous. Le but des contrôles hypermédias est qu'ils nous disent ce que nous pouvons faire ensuite, et l'URI de la ressource que nous devons manipuler

pour le faire. Plutôt que d'avoir à savoir où afficher notre demande de rendez-vous, les contrôles hypermédias dans la réponse nous disent comment le faire.

La requête POST de réservation d'un créneau est la même:

```
POST /slots/1234 HTTP/1.1
[various other headers]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

Mais la réponse à cette requête sera comme suit :

```
HTTP/1.1 201 Created
Location: http://doctors.com/slots/1234/appointment
[various headers]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel"
       uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest"
       uri = "/slots/1234/appointment/tests"/>
  <link rel = "self"
       uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
       uri = "/doctors/mjones/slots?date=20230504&status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
       uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help"
       uri = "/help/appointment"/>
</appointment>
```

Un avantage évident des contrôles hypermédias est qu'il permet au serveur de changer son schéma URI sans impacter les clients. Tant que les clients consultent l'URI de lien "addTest", l'équipe de développement coté serveur peut manipuler les URI autres que les points d'entrée initiaux.

Un autre avantage est qu'il aide les développeurs clients à explorer le protocole. L'utilisation des contrôles hypermédia permet d'auto-documenter l'API REST. Les liens donnent aux développeurs clients un indice de ce qui pourrait être possible ensuite. Il ne donne pas toutes les informations : les contrôles "self" et "cancel" pointent tous deux vers la même URI - ils doivent comprendre que l'un est un GET et l'autre un DELETE. Mais au moins cela leur donne un point de départ quant à ce à quoi penser pour plus d'informations et de chercher une URI similaire dans la documentation de l'API.

Conclusion

Le modèle de matyrité de Richardson est un processus conducteur permettant de capturer pas à pas les concepts sous-jacents à une approche RESTFul :

- Le niveau 1 aborde la question de la gestion de la complexité en décomposant un grand endpoint de service en plusieurs ressources.
- Le niveau 2 introduit un ensemble standard de verbes afin que nous gérions des situations similaires de la même manière, en supprimant les variations inutiles. Aussi, à ce niveau choisissons les codes de retours HTTP les plus appropriés.
- Le niveau 3 introduit la découvrabilité, fournissant un moyen de rendre un système plus auto-documenté.

Références

[1] Martin Fowler "Richardson Maturity Model: steps toward the glory of REST", https://martinfowler.com/articles/richardsonMaturityModel.html

[2] Christophe Heubès "REST : Richardson Maturity Model", https://blog.engineering.publicissapient.fr/2010/06/25/rest-richardson-maturity-model/

Bibliographie

Leonard Richardson, Sam Ruby, David Heinemeier Hansson (Foreword) "RESTful Web Services", 1st Edition, O'Reilly.

Jim Webber, Savas Parastatidis, Ian Robinso "REST in Practice: Hypermedia and Systems Architecture", 1st Edition, O'Reilly.