

Chapitre 9 - HATEOAS

- [Les liens hypermédias](#)
- [HAL et JSON:API](#)
- [Des liens qui dépendent de l'état](#)
- [Pourquoi HATEOAS est rare](#)

Les liens hypermédias

Sur un site web, vous ne tapez jamais d'URL à la main : vous arrivez sur la page d'accueil et vous cliquez sur des liens. Le site peut réorganiser toutes ses adresses, vous ne vous en rendez pas compte.

HATEOAS (*Hypermedia As The Engine Of Application State*) applique la même idée aux API : chaque réponse contient des **liens** vers les ressources liées et les actions possibles. Le client n'a besoin de connaître qu'une seule URL, le point d'entrée.

C'est la dernière contrainte de l'interface uniforme de REST, et le niveau 3 du modèle de Richardson (chapitre 2).

Partir du point d'entrée

```
curl -s http://api-ritif.portlatch.com/v1/
```

```
{
  "name": "API-ritif",
  "version": "1.0.0",
  "_links": {
    "self":      {"href": "/v1/"},
    "cocktails": {"href": "/v1/cocktails"},
    "customers": {"href": "/v1/customers", "method": "POST"},
    "me":        {"href": "/v1/me"},
    "orders":    {"href": "/v1/orders"},
    "docs":      {"href": "/docs"}
  }
}
```

On suit `cocktails`, puis le lien `self` d'un cocktail :

```
{
  "id": 3,
  "name": "Segfault Sunrise",
  "price": 9.5,
```

```
"_links": {
  "self":    {"href": "/v1/cocktails/3"},
  "reviews": {"href": "/v1/cocktails/3/reviews"},
  "order":   {"href": "/v1/orders", "method": "POST"}
}
```

Le cocktail indique lui-même où trouver ses avis et comment le commander. Aucune URL n'a été construite par le client.

Anatomie d'un lien

- La **clé** est la **relation** (*rel*) : elle dit ce que représente le lien par rapport à la ressource courante.
- `href` est l'adresse.
- `method` (extension d'API-ritif) précise la méthode HTTP quand ce n'est pas `GET`.

Les relations courantes

Certaines relations sont standardisées par l'IANA, et tout le monde les comprend :

Relation	Sens
<code>self</code>	La ressource elle-même
<code>next</code> , <code>prev</code> , <code>first</code> , <code>last</code>	Navigation dans une liste paginée
<code>collection</code>	La collection à laquelle appartient l'élément
<code>item</code>	Un élément d'une collection
<code>related</code>	Une ressource liée
<code>edit</code>	L'adresse pour modifier la ressource

Les autres sont propres à votre métier : `cancel`, `reviews`, `order`, `pay`... Choisissez des noms explicites et documentez-les.

Ce que ça change pour le client

- Le serveur peut **changer ses URL** sans casser les clients, puisqu'ils suivent les liens.
- Le client **découvre** ce qui est possible au lieu de le deviner.

- Les **règles métier** restent côté serveur : si une action n'est pas permise, son lien n'apparaît pas (page « Des liens qui dépendent de l'état »).

Vous utilisez HATEOAS depuis le chapitre 4 sans le savoir : parcourir une liste en suivant `_links.next`, c'est exactement ça.

HAL et JSON:API

REST dit qu'il faut des liens, mais pas comment les écrire. Plusieurs formats se sont imposés. Voici la même commande dans les deux plus répandus.

HAL

HAL (*Hypertext Application Language*, type `application/hal+json`) est le plus simple : on garde son JSON habituel et on ajoute deux clés réservées.

- `_links` : les liens, indexés par relation ;
- `_embedded` : des ressources liées incluses directement, pour éviter des requêtes supplémentaires.

```
{
  "id": 42,
  "status": "pending",
  "price": 9.5,
  "_links": {
    "self": {"href": "/v1/orders/42"},
    "cocktail": {"href": "/v1/cocktails/3"}
  },
  "_embedded": {
    "cocktail": {
      "id": 3,
      "name": "Segfault Sunrise",
      "_links": {"self": {"href": "/v1/cocktails/3"}}
    }
  }
}
```

HAL permet aussi des liens **modèles** (*templated*), à compléter par le client :

```
"search": {"href": "/v1/cocktails{?tag,max_price}", "templated": true}
```

JSON:API

JSON:API (type `application/vnd.api+json`) va beaucoup plus loin : il impose la structure complète du document, la pagination, les filtres, le tri, les erreurs et les relations.

```
{
  "data": {
    "type": "orders",
    "id": "42",
    "attributes": {
      "status": "pending",
      "price": 9.5
    },
    "relationships": {
      "cocktail": {
        "data": {"type": "cocktails", "id": "3"},
        "links": {"related": "/v1/cocktails/3"}
      }
    },
    "links": {"self": "/v1/orders/42"}
  },
  "included": [
    {
      "type": "cocktails",
      "id": "3",
      "attributes": {"name": "Segfault Sunrise"}
    }
  ]
}
```

La syntaxe de tri `?sort=-price,name` utilisée dans ce cours vient de JSON:API.

Comparatif

	HAL	JSON:API
Philosophie	Ajouter des liens à son JSON	Un format complet et strict
Données	À la racine, comme d'habitude	Dans <code>data.attributes</code>
Ressources incluses	<code>_embedded</code>	<code>included</code>
Actions (méthode, champs)	Non prévu	Non prévu

	HAL	JSON:API
Prise en main	Immédiate	Il faut lire la spécification
Outillage	Correct	Bibliothèques dans la plupart des langages

Les autres

- **Siren** décrit aussi les **actions** possibles : méthode, URL et champs à envoyer, comme un formulaire HTML.
- **JSON-LD** (et son extension Hydra) vient du web sémantique : il relie les données à des vocabulaires partagés. Google l'utilise pour les données structurées des pages web.

Et API-ritif ?

API-ritif utilise un style **inspiré de HAL** : des `_links` indexés par relation, avec un champ `method` en plus pour les actions. Ce n'est pas du HAL strict, et c'est très courant : beaucoup d'API prennent ce qui les arrange dans HAL.

Pour vos propres API, aucun format n'est imposé. Ce qui compte : des liens cohérents, partout pareils, et documentés dans le contrat.

Des liens qui dépendent de l'état

C'est là que HATEOAS devient vraiment utile : les liens ne décrivent pas seulement les ressources liées, mais aussi **ce qu'on a le droit de faire maintenant**.

La vie d'une commande

```
pending —(30 s)→ preparing —(2 min)→ ready → served
|
└→ cancelled (uniquement tant qu'elle est pending)
```

Juste après la commande :

```
{
  "id": 42,
  "status": "pending",
  "cocktail": "Segfault Sunrise",
  "_links": {
    "self": {"href": "/v1/orders/42"},
    "cancel": {"href": "/v1/orders/42", "method": "DELETE"},
    "cocktail": {"href": "/v1/cocktails/3"}
  }
}
```

30 secondes plus tard, le barman a commencé. Même requête :

```
{
  "id": 42,
  "status": "preparing",
  "cocktail": "Segfault Sunrise",
  "_links": {
    "self": {"href": "/v1/orders/42"},
    "cocktail": {"href": "/v1/cocktails/3"}
  }
}
```

```
}
```

Le lien `cancel` a disparu. Le serveur ne dit pas « tu ne peux plus annuler parce que le statut est `preparing` » : il ne propose simplement plus l'action.

Faites l'essai : commandez un cocktail et interrogez la commande toutes les 10 secondes avec curl. Regardez le lien disparaître en direct.

Côté client : suivre les liens, pas les règles

Sans HATEOAS, le client doit connaître la règle métier :

```
# □ La règle est dupliquée dans le client
if order["status"] == "pending":
    show_cancel_button()
```

Le jour où le bar autorise l'annulation pendant la préparation, il faut mettre à jour toutes les applications : web, mobile, caisse... Avec HATEOAS, le client se contente de regarder si le lien existe :

```
order = requests.get(urljoin(BASE, "/v1/orders/" + str(order_id)), headers=headers).json()

cancel = order["_links"].get("cancel")
if cancel:
    print("Annulation possible")
    requests.request(cancel["method"], urljoin(BASE, cancel["href"]), headers=headers)
else:
    print("Trop tard, le shaker tourne déjà")
```

La règle n'existe qu'à un seul endroit : le serveur. Il peut la changer, tous les clients suivent.

Ça marche aussi pour les droits

Les liens peuvent dépendre de **qui** demande, pas seulement de l'état :

- un barman voit un lien `serve` sur les commandes `ready`, un client non ;
- l'auteur d'un avis voit un lien `delete`, les autres non ;

- un client authentifié voit un lien `order` sur un cocktail, un visiteur anonyme voit un lien `register`.

Masquer un lien n'est **pas** une protection : le serveur doit toujours vérifier l'état et les droits quand l'action arrive. Un `DELETE` envoyé à la main sur une commande en préparation doit toujours répondre `409`.

Pourquoi HATEOAS est rare

Fielding le répète depuis 2008 : sans hypermédia, ce n'est pas REST. Pourtant, la grande majorité des API publiques s'arrêtent au niveau 2 de Richardson. Pourquoi ?

Les raisons

Les clients codent quand même les URL en dur

La promesse « le serveur peut changer ses URL sans casser les clients » suppose que les clients suivent réellement les liens. En pratique, les développeurs lisent la documentation et écrivent `/v1/orders/42` directement dans leur code. Le bénéfice disparaît.

Un client ne comprend pas une relation inconnue

Un navigateur affiche un lien et laisse un humain décider. Un programme, lui, ne sait pas quoi faire d'un lien `refund` qu'il n'a jamais vu : quelqu'un doit avoir écrit le code qui le gère. Les liens ne remplacent donc pas la documentation.

Pas de standard unique

HAL, JSON:API, Siren, JSON-LD... Chaque format a ses outils, et aucun ne s'est imposé. Beaucoup d'équipes préfèrent ne rien choisir.

Des réponses plus lourdes

Des liens dans chaque élément d'une liste de 100 éléments, c'est du volume en plus, et du code serveur en plus pour les calculer selon l'état et les droits.

OpenAPI couvre une partie du besoin

La découverte de l'API se fait avec la documentation et les clients générés plutôt qu'à l'exécution.

Là où c'est utile

HATEOAS n'est pas tout ou rien. Certaines parties rendent service dans presque toutes les API :

- **La pagination** : `next`, `prev`, `first`, `last`. Presque toutes les API sérieuses le font.
- **Les workflows** : commandes, paiements, validations... les liens d'actions selon l'état évitent de dupliquer les règles métier dans chaque client.
- **Le header Location** après une création : c'est déjà un lien hypermédia.
- **Les agents IA** : un LLM qui découvre une API, lui, sait lire un lien qu'il n'a jamais vu et en déduire quoi faire. Un point d'entrée avec des liens explicites l'aide à naviguer sans documentation.

L'API REST de PayPal, par exemple, renvoie dans chaque paiement une liste de liens (`approve`, `capture`...) avec leur méthode HTTP. Le client suit le lien `approve` pour rediriger l'acheteur, sans construire l'URL.

En résumé

☐ Pour	☐ Contre
Le client suit des liens au lieu de construire des URL	Les clients codent souvent les URL en dur quand même
Les règles métier restent côté serveur	Un client ne sait pas gérer une relation inconnue
L'API se découvre depuis un point d'entrée	Pas de format standard unique
Utile pour la pagination, les workflows et les IA	Réponses plus lourdes, serveur plus complexe

La position raisonnable : des liens de pagination partout, des liens d'actions là où l'état compte, et une bonne documentation OpenAPI à côté.