

Chapitre 5 - Concevoir une API

- [Nommer les ressources et les routes](#)
- [Filtres, tri et pagination](#)
- [Versioning](#)
- [Les erreurs avec problem+json](#)
- [Exercice : L'API du stagiaire de 2009](#)

Nommer les ressources et les routes

Une URL est la première chose qu'un développeur voit de votre API. Si elle est claire, il devine la suite. Si elle ne l'est pas, il ouvre la documentation... ou abandonne.

Les règles

Des noms, pas des verbes

L'URL désigne une **ressource**. L'action, c'est la méthode HTTP qui la porte.

□ GET /getCocktails	□ GET /cocktails
□ POST /createOrder	□ POST /orders
□ POST /deleteOrder?id=42	□ DELETE /orders/42

Au pluriel

Une collection contient plusieurs éléments : `/cocktails` et `/cocktails/3`, plutôt que `/cocktail/3`. Surtout : **ne mélangez pas**.

En minuscules, avec des tirets

Les URL sont sensibles à la casse : `/Cocktails` et `/cocktails` sont deux adresses différentes. On écrit tout en minuscules, et on sépare les mots par des tirets (*kebab-case*).

□ /happyHours	□ /happy_hours	□ /happy-hours
---------------	----------------	----------------

Pour les champs JSON, la convention est différente : `snake_case` ou `camelCase`, au choix, mais le même partout.

Une hiérarchie courte

Les sous-ressources expriment une appartenance : `/cocktails/3/reviews` se lit « les avis du cocktail 3 ». Au-delà de deux niveaux, ça devient illisible.

□ /bars/1/customers/12/orders/42/items/3

□ /orders/42/items/3

Pas de détails techniques

Pas d'extension (`.php`, `.json`), pas de nom de technologie, pas de nom de table. Si vous réécrivez le back en Go, les URL ne doivent pas changer.

□ /api/v1/getCocktails.php

□ /tbl_cocktails

□ /v1/cocktails

Une seule langue

L'anglais est recommandé : c'est la langue de la plupart des API et des outils. Une API en français (comme la bibliothèque des chapitres 1 et 2) reste acceptable, à condition de ne jamais mélanger les deux : `/livres/3/reviews`, non merci.

Et les actions qui ne sont pas du CRUD ?

Annuler une commande, payer, rembourser... Ces actions ne sont pas « créer, lire, modifier, supprimer ». Plusieurs solutions, de la plus REST à la plus pragmatique :

- **Changer l'état** de la ressource : `PATCH /orders/42` avec `{"status": "cancelled"}`.
- **Utiliser le verbe qui correspond**, quand il existe : annuler une commande, c'est `DELETE /orders/42` (choix d'API-ritif).
- **Créer une sous-ressource** qui représente l'action : `POST /orders/42/payments` crée un paiement.
- En dernier recours, une **route d'action** : `POST /orders/42/cancel`. Ce n'est pas très REST, mais c'est clair et c'est très courant.

La règle d'or : un développeur qui découvre votre API doit pouvoir **deviner** une URL qu'il n'a jamais vue. Si `/cocktails/3/reviews` existe, il s'attend à trouver `/orders/42`, pas `/getOrderById?order=42`.

Filtres, tri et pagination

Une collection peut contenir des milliers d'éléments. Le client doit pouvoir n'en demander qu'une partie, dans l'ordre qui l'intéresse. Tout passe par les **paramètres de query** : la ressource reste la même, on ne change que la vue.

Les filtres

Un paramètre par critère, avec le nom du champ :

```
GET /v1/cocktails?alcohol_free=true
GET /v1/cocktails?tag=classic&max_price=10
GET /v1/orders?status=pending
```

Pas de filtre dans le chemin : `/cocktails/alcohol-free` laisse croire qu'il s'agit d'une ressource à part, et ne se combine pas avec les autres filtres.

Le tri

Un paramètre `sort` avec le nom du champ. Un `-` devant inverse l'ordre, et on peut trier sur plusieurs champs séparés par des virgules :

GET /v1/cocktails?sort=price	prix croissant
GET /v1/cocktails?sort=-price	prix décroissant
GET /v1/cocktails?sort=-price,name	prix décroissant, puis nom

Un tri par défaut doit toujours exister (souvent par `id` ou par date) : sans lui, l'ordre peut changer d'une page à l'autre et des éléments apparaissent en double.

La pagination

Par numéro de page (offset)

```
GET /v1/cocktails?page=2&per_page=5
```

- ☐ Simple, on peut sauter directement à la page 5.
- ☐ Si des éléments sont ajoutés pendant qu'on parcourt, on voit des doublons ou on en rate.
- ☐ Lent sur de très grosses tables (la base doit compter les lignes à sauter).

Par curseur

```
GET /v1/orders?limit=20
GET /v1/orders?limit=20&cursor=eyJpZCI6NDJ9
```

Le serveur renvoie un **curseur** opaque qui marque la position atteinte. C'est le choix de Stripe, Slack ou Twitter.

- ☐ Stable même si des données arrivent, et rapide.
- ☐ Impossible de sauter à une page précise.

Ce que la réponse doit contenir

Le client doit savoir où il en est et comment continuer. Deux approches, souvent combinées :

- des métadonnées et des liens dans le body ;
- le header `Link` (RFC 8288), utilisé par exemple par GitHub.

```
{
  "items": [ ... ],
  "page": 2,
  "per_page": 5,
  "total": 13,
  "_links": {
    "self": { "href": "/v1/cocktails?page=2&per_page=5" },
    "first": { "href": "/v1/cocktails?page=1&per_page=5" },
    "prev": { "href": "/v1/cocktails?page=1&per_page=5" },
    "next": { "href": "/v1/cocktails?page=3&per_page=5" },
    "last": { "href": "/v1/cocktails?page=3&per_page=5" }
  }
}
```

```
Link: </v1/cocktails?page=3&per_page=5>; rel="next", </v1/cocktails?page=3&per_page=5>;
rel="last"
```

Fixez toujours une **limite maximale** à `per_page`. Sinon, quelqu'un finira par demander `?per_page=1000000`, et votre serveur s'en souviendra.

En résumé

Besoin	Convention
Filtrer	<code>?champ=valeur</code>
Trier	<code>?sort=-champ1,champ2</code>
Paginer	<code>?page=&per_page=</code> ou <code>?cursor=&limit=</code>
Naviguer	<code>_links</code> et/ou header <code>Link</code>

Versioning

Une API publiée a des clients que vous ne connaissez pas : des applications mobiles qui ne seront pas mises à jour avant des mois, des scripts oubliés sur un serveur, des intégrations de partenaires... Changer l'API peut tous les casser d'un coup.

Ce qui casse, ce qui ne casse pas

□ Compatible (pas de nouvelle version)	□ Rupture (<i>breaking change</i>)
Ajouter une route	Supprimer ou renommer une route
Ajouter un champ dans une réponse	Supprimer ou renommer un champ
Ajouter un paramètre facultatif	Rendre un paramètre obligatoire
Ajouter une valeur possible à un filtre	Changer le type d'un champ ("9.5" → 9.5)
	Changer le sens d'un code de statut

Côté client, la règle inverse : ignorez les champs que vous ne connaissez pas. Un client qui plante parce qu'un champ a été ajouté est un client mal écrit, même si cela existe parfois.

Où mettre la version ?

Dans l'URL

```
GET /v1/cocktails
GET /v2/cocktails
```

La solution la plus répandue : visible, simple à tester avec curl, facile à router. C'est le choix d'API-ritif et la méthode recommandée dans ce cours.

Dans un header

```
GET /cocktails
X-API-Version: 2
```

Les URL restent stables, mais la version est invisible dans un navigateur ou un lien. GitHub (`X-GitHub-API-Version: 2022-11-28`) et Stripe (`Stripe-Version`) utilisent des dates comme numéro de version.

Dans le media type

```
GET /cocktails
Accept: application/vnd.api-ritif.v2+json
```

La plus « REST » selon les puristes, puisque c'est la représentation qui change. La moins pratique au quotidien.

Faire vivre les versions

- On ne change de version que pour une rupture : `v1`, `v2` ... pas de `v1.3.7` dans l'URL.
- On maintient l'ancienne version pendant une période annoncée.
- On prévient les clients, y compris dans les réponses elles-mêmes, avec les headers `Deprecation` (la version est dépréciée) et `Sunset` (date de fin) :

```
HTTP/1.1 200 OK
Deprecation: @1767225600
Sunset: Wed, 30 Jun 2027 23:59:59 GMT
Link: </v2/cocktails>; rel="successor-version"
```

Le meilleur versioning est celui dont on n'a jamais besoin : une API bien conçue dès le départ, et qui n'évolue que par ajouts compatibles, peut vivre des années en `v1`.

Les erreurs avec problem+json

Le code de statut dit **qu'il** y a une erreur. Le body doit dire **laquelle**, et comment la corriger. Plutôt que d'inventer un format par API, il existe un standard : `application/problem+json`.

Le format

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

{
  "type": "https://api-ritif.example/problems/order-not-cancellable",
  "title": "La commande ne peut plus être annulée",
  "status": 409,
  "detail": "La commande 42 est en préparation. Le shaker tourne déjà.",
  "instance": "/v1/orders/42"
}
```

Champ	Rôle
<code>type</code>	Une URI qui identifie le type d'erreur. Idéalement, une page de documentation qui l'explique. Par défaut : <code>about:blank</code> .
<code>title</code>	Un résumé court, le même pour toutes les erreurs de ce type.
<code>status</code>	Le code de statut HTTP, répété pour faciliter la lecture.
<code>detail</code>	L'explication propre à cette occurrence, lisible par un humain.
<code>instance</code>	Une URI qui identifie cette occurrence précise (la ressource concernée, ou un identifiant de log).

Tous les champs sont facultatifs, mais `type`, `title` et `status` sont un minimum raisonnable.

Les extensions

On peut ajouter ses propres champs. Le cas le plus courant : les erreurs de validation, avec la liste des champs en faute.

```
HTTP/1.1 422 Unprocessable Content
Content-Type: application/problem+json

{
  "type": "https://api-ritif.example/problems/validation-error",
  "title": "Données invalides",
  "status": 422,
  "detail": "La requête contient 2 erreurs.",
  "errors": [
    {"field": "rating", "message": "doit être compris entre 1 et 5"},
    {"field": "comment", "message": "ne doit pas dépasser 500 caractères"}
  ]
}
```

Le client peut afficher chaque message sous le bon champ du formulaire, sans parser une phrase.

Les bonnes pratiques

- Le même format pour **toutes** les erreurs de l'API, y compris les `404` et les `500`.
- Un `detail` qui dit quoi corriger : « la note doit être comprise entre 1 et 5 », pas « erreur de validation ».
- Jamais d'informations internes dans une erreur.

Ceci n'est pas un message d'erreur, c'est une fuite d'informations :

```
{"error": "SQLSTATE[42S22]: Column not found: 1054 Unknown column 'prix' in  
/var/www/api/models/Cocktail.php line 42"}
```

Un attaquant y apprend votre base de données, votre langage, votre arborescence et le nom de vos colonnes. Loggez le détail côté serveur, renvoyez un message neutre au client.

Exercice : L'API du stagiaire de 2009

Objectif

En rangeant la réserve du bar, le patron d'API-ritif a retrouvé une clé USB étiquetée « *API v0 — NE PAS TOUCHER — Kevin, stage 2009* ». Elle contient la documentation de l'ancienne API du bar, toujours utilisée par la vieille caisse enregistreuse.

Votre mission, en groupe : **critiquer cette API et proposer une version propre.**

La documentation retrouvée

```
=== API DU BAR - by Kevin - 2009 ===  
(marche bien normalement)
```

GET /getCocktails.php	-> liste des cocktails (tous, d'un coup)
GET /getCocktails.php?alcool=0	-> cocktails sans alcool
GET /Cocktail?id=3	-> un cocktail
GET /cocktails/3	-> un cocktail aussi (nouvelle version)
GET /api/cocktail/delete?id=3	-> supprime un cocktail
POST /api/cocktail/updatePriceV2_final	-> change le prix (envoyer id et prix)
POST /orders/list	-> liste des commandes
POST /addOrderNew	-> passer commande
GET /orders?user_id=12	-> commandes d'un client
GET /api/getUserPassword?user=bob	-> pour le mot de passe oublié

Quelques réponses relevées :

```
GET /Cocktail?id=999  
HTTP/1.1 200 OK  
{ "error": true, "msg": "pas trouvé" }  
  
POST /addOrderNew
```

```
HTTP/1.1 200 OK
{"error": false, "data": {"Id": 42, "prix": "9,50 €", "Date": "12/03/09"}}

GET /getCocktails.php?alcool=peut-etre
HTTP/1.1 200 OK
{"error": true, "msg": "SQLSTATE[42000]: Syntax error near 'peut-etre' in
/var/www/bar/getCocktails.php line 12"}
```

Consignes

1 - Critiquer

Pour chaque route et chaque réponse, listez **tous** les problèmes que vous trouvez. Pensez à tout ce qui a été vu jusqu'ici :

- nommage des routes ;
- choix des méthodes HTTP ;
- codes de statut ;
- format des données et des erreurs ;
- pagination ;
- sécurité.

Il y a plus de problèmes que de lignes. Certains sont gênants, d'autres sont graves.

2 - Corriger

Proposez la nouvelle version de l'API sous forme de tableau : méthode, route, rôle, codes de statut possibles. Proposez aussi le format corrigé des réponses ci-dessus.

Si une route ne devrait tout simplement pas exister, dites-le et expliquez pourquoi.

3 - Migrer

La vieille caisse enregistreuse ne peut pas être mise à jour avant six mois. Comment faire cohabiter l'ancienne et la nouvelle API ?

Résultat attendu

Une restitution orale par groupe :

- les 3 problèmes que vous jugez les plus graves, et pourquoi ;
- votre tableau des nouvelles routes ;
- votre stratégie de migration.