

Chapitre 6 - Documenter avec OpenAPI

- [Lire un contrat OpenAPI](#)
- [Structure YAML et schémas](#)
- [Code-first ou design-first](#)
- [Swagger UI, Redoc et Scalar](#)
- [Générer des clients](#)

Lire un contrat OpenAPI

Pendant la chasse au trésor, vous avez découvert API-ritif à tâtons, en lisant les réponses une par une. Il existe une méthode plus reposante : lire son **contrat**.

Qu'est-ce qu'un contrat d'API ?

Un contrat décrit **tout** ce qu'une API sait faire, dans un format lisible par les humains et par les machines :

- les routes et les méthodes ;
- les paramètres attendus et leur type ;
- le format des bodies envoyés et reçus ;
- les codes de statut possibles ;
- la méthode d'authentification.

Le standard est **OpenAPI** (anciennement *Swagger*), écrit en YAML ou en JSON. Ce cours utilise la version **3.1**. La 3.2, sortie en 2025, est encore peu supportée par les outils.

Le contrat est la **source de vérité** partagée entre l'équipe front, l'équipe back, les testeurs, les partenaires... et les IA. Quand le code et le contrat ne disent pas la même chose, c'est un bug.

Ouvrir le contrat d'API-ritif

- Téléchargez le contrat : [api-ritif.openapi.yaml](#)
- Ouvrez [editor.swagger.io](#).
- Menu **File** → **Import file**, ou copiez-collez le contenu à gauche.

À gauche, le YAML. À droite, la documentation générée.

Swagger Editor sert à **lire et écrire** le contrat. Pour envoyer de vraies requêtes, utilisez la documentation servie par API-ritif elle-même : <http://api-ritif.portlatch.com/docs> (bouton **Try it out**). Depuis editor.swagger.io, les requêtes seraient bloquées par le navigateur : vous comprendrez pourquoi au chapitre 8

(CORS).

Swagger Editor signale en direct les erreurs de syntaxe : c'est aussi l'outil idéal pour **écrire** un contrat.

Ce qu'on y lit

Un extrait, pour la route qui annule une commande :

```
/orders/{order_id}:
  delete:
    tags: [Commandes]
    operationId: cancelOrder
    summary: Annuler une commande
    description: Possible uniquement tant que la commande est `pending`.
    security:
      - ApiKeyAuth: []
    responses:
      '204':
        description: Commande annulée
      '401':
        $ref: '#/components/responses/Unauthorized'
      '404':
        $ref: '#/components/responses/NotFound'
      '409':
        description: La commande n'est plus annulable
```

Sans même connaître OpenAPI, on comprend : `DELETE /orders/{order_id}`, authentifié par clé d'API, répond `204` en cas de succès et `409` si c'est trop tard. Tout ce que vous avez découvert à la main pendant la chasse au trésor était écrit là.

À vous de chercher

Trouvez dans le contrat la réponse à ces questions :

1. Quelles routes peut-on appeler sans être authentifié ?
2. Combien de caractères peut contenir un `message_to_bartender` ?
3. Combien d'éléments une page contient-elle par défaut ? Et au maximum ?

4. Quels codes de statut `POST /orders` peut-il renvoyer ?
5. Quels sont les statuts possibles d'une commande ?
6. Quelles sont les deux façons de s'authentifier ?

Structure YAML et schémas

YAML en 30 secondes

YAML est un format de données lisible, basé sur l'**indentation** (des espaces, jamais de tabulations).

```
nom: Segfault Sunrise      # une clé et une valeur
prix: 9.5                  # un nombre
sans_alcool: false        # un booléen
tags:                      # une liste
  - classic
  - fruity
recette:                   # un objet imbriqué
  base: rhum
  dose_cl: 4
code: '404'                # des guillemets pour forcer une chaîne
```

Une indentation décalée d'un espace change complètement le sens du document. Si Swagger Editor affiche une erreur incompréhensible, vérifiez d'abord l'indentation.

La structure d'un contrat

```
openapi: 3.1.0             # version d'OpenAPI

info:                      # présentation de l'API
  title: API-ritif
  version: 1.0.0
  description: Le bar de l'école de dev.

servers:                   # où se trouve l'API
  - url: http://api-ritif.portlatch.com/v1

paths:                     # les routes
```

```
/cocktails: ...  
/cocktails/{cocktail_id}: ...
```

```
components:           # les éléments réutilisables  
  schemas: ...         #   formats de données  
  parameters: ...      #   paramètres communs  
  responses: ...        #   réponses communes  
  securitySchemes: ...  #   méthodes d'authentification
```

Une opération

Sous chaque chemin, une clé par méthode HTTP. Chaque opération décrit ses paramètres, son body et ses réponses.

```
paths:  
  /cocktails/{cocktail_id}/reviews:  
    post:  
      summary: Donner son avis sur un cocktail  
      operationId: createReview  
      parameters:  
        - name: cocktail_id  
          in: path           # path, query, header ou cookie  
          required: true  
          schema:  
            type: integer  
      requestBody:  
        required: true  
        content:  
          application/json:  
            schema:  
              $ref: '#/components/schemas/ReviewCreate'  
      responses:  
        '201':  
          description: Avis créé  
          content:  
            application/json:  
              schema:  
                $ref: '#/components/schemas/Review'  
        '422':
```

L' `operationId` donne un nom unique à l'opération. Les générateurs de code s'en servent pour nommer les fonctions : choisissez-le bien.

Les schémas

Les schémas décrivent la forme des données, avec la syntaxe **JSON Schema**.

```
components:
  schemas:
    ReviewCreate:
      type: object
      required: [rating]           # champs obligatoires
      additionalProperties: false  # aucun autre champ accepté
      properties:
        rating:
          type: integer
          minimum: 1
          maximum: 5
          example: 4
        comment:
          type: string
          maxLength: 500
          example: Plante rarement, mais quel goût.

    OrderStatus:
      type: string
      enum: [pending, preparing, ready, served, cancelled]
```

Mot-clé	Rôle
type	string, integer, number, boolean, array, object, null
required	Liste des champs obligatoires
enum	Liste des valeurs autorisées
minimum, maximum, minLength, maxLength, pattern	Contraintes de validation
format	Précision : date-time, email, uri...

Mot-clé	Rôle
example	Un exemple, affiché dans la documentation
readOnly	Champ renvoyé par l'API, mais jamais accepté en entrée

Ne pas se répéter : `$ref`

`$ref` fait référence à un élément défini ailleurs. Le schéma `Problem`, les paramètres de pagination ou la réponse `429` ne sont écrits qu'une fois dans `components`, puis réutilisés partout.

```
responses:  
  '429':  
    $ref: '#/components/responses/TooManyRequests'
```

L'authentification

```
components:  
  securitySchemes:  
    ApiKeyAuth:  
      type: apiKey  
      in: header  
      name: X-API-Key  
  
paths:  
  /me:  
    get:  
      security:  
        - ApiKeyAuth: []    # cette route demande une clé  
  /cocktails:  
    get:  
      security: []          # cette route est publique
```

Avec `securitySchemes`, le bouton **Authorize** de Swagger UI permet de saisir sa clé une fois pour toutes les requêtes de test.

Code-first ou design-first

Le contrat et le code doivent dire la même chose. Mais lequel écrit-on en premier ?

Code-first

On écrit le code, et le contrat est **généré** à partir de lui (annotations, types, décorateurs...).

C'est ce que fait **FastAPI** : les types Python et les modèles Pydantic suffisent à produire un contrat OpenAPI complet, disponible sur `/openapi.json`, avec sa documentation sur `/docs`.

```
from fastapi import FastAPI
from pydantic import BaseModel, Field

app = FastAPI(title="API-ritif")

class ReviewCreate(BaseModel):
    rating: int = Field(ge=1, le=5)
    comment: str | None = Field(default=None, max_length=500)

@app.post("/v1/cocktails/{cocktail_id}/reviews", status_code=201)
def create_review(cocktail_id: int, review: ReviewCreate):
    ...
```

Rien d'autre à écrire : `rating` apparaît dans le contrat comme un entier entre 1 et 5.

- ☐ Le contrat est toujours synchronisé avec le code.
- ☐ Rapide pour démarrer.
- ☐ Le contrat n'existe qu'une fois le code écrit : le front attend le back.
- ☐ La conception de l'API est guidée par l'implémentation, pas par les besoins du client.

Tous les frameworks n'en sont pas capables : **Flask** ou **Express** (JavaScript) ne connaissent pas les types des données. Il faut ajouter des bibliothèques et des

Design-first

On écrit d'abord le **contrat**, on le fait valider par ceux qui vont l'utiliser, puis on code.

- ☐ On réfléchit à l'API du point de vue de ses clients.
- ☐ Le front et le back travaillent en parallèle : le front utilise un **mock** généré à partir du contrat (voir chapitre 7).
- ☐ Le contrat sert à générer des tests, des clients, de la validation.
- ☐ Il faut une discipline pour garder le code conforme au contrat.

Des outils vérifient les requêtes et les réponses directement à partir du contrat, par exemple Prism en mode proxy (chapitre 7). Le contrat devient du code exécutable.

Comparatif

	Code-first	Design-first
Point de départ	Le code	Le contrat
Synchronisation	Automatique	Discipline et outils de validation
Travail en parallèle	Difficile	Naturel
Adapté à...	Une API interne, une petite équipe, un prototype	Une API publique, plusieurs équipes, des partenaires

Dans le TP des avis (chapitre 7), le contrat OpenAPI vous est fourni **avant** le code : c'est du design-first, et c'est lui qui fait foi.

Swagger UI, Redoc et Scalar

Un contrat en YAML, c'est précis mais austère. Des outils le transforment en une documentation agréable à lire, sans rien écrire de plus.

Trois outils, un même contrat

Outil	Points forts
Swagger UI	Le classique. Chaque route est testable directement (Try it out). Intégré à FastAPI sur <code>/docs</code> .
Redoc	Une documentation en trois colonnes, très lisible, idéale pour une API publique. Lecture seule. Intégré à FastAPI sur <code>/redoc</code> .
Scalar	Le plus récent : moderne, avec un client de test intégré et des exemples de code générés dans de nombreux langages.

Publier la documentation

Chaque outil tient en une page HTML qui charge le contrat. Placez la page à côté du fichier `api-ritif.openapi.yaml`.

Swagger UI

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="https://unpkg.com/swagger-ui-dist@5/swagger-ui.css">
</head>
<body>
  <div id="swagger-ui"></div>
  <script src="https://unpkg.com/swagger-ui-dist@5/swagger-ui-bundle.js"></script>
  <script>
    SwaggerUIBundle({ url: "api-ritif.openapi.yaml", dom_id: "#swagger-ui" });
  </script>
```

```
</body>
</html>
```

Redoc

```
<!doctype html>
<html>
<head><meta charset="utf-8"></head>
<body>
  <redoc spec-url="api-ritif.openapi.yaml"></redoc>
  <script src="https://cdn.redoc.ly/redoc/latest/bundles/redoc.standalone.js"></script>
</body>
</html>
```

Scalar

```
<!doctype html>
<html>
<head><meta charset="utf-8"></head>
<body>
  <div id="app"></div>
  <script src="https://cdn.jsdelivr.net/npm/@scalar/api-reference"></script>
  <script>
    Scalar.createApiReference("#app", { url: "api-ritif.openapi.yaml" });
  </script>
</body>
</html>
```

Ouverte en double-cliquant (`file://`), la page ne pourra pas charger le contrat.
Servez le dossier avec un petit serveur web : `python3 -m http.server` , puis ouvrez
<http://localhost:8000>.

Ce qui fait une bonne documentation

Les outils mettent en forme. Le contenu, c'est vous qui l'écrivez dans le contrat :

- une `description` générale : à quoi sert l'API, comment s'authentifier, les limites de requêtes ;
- un `summary` court et clair pour chaque opération ;

- une `description` pour les règles métier (« annulable uniquement tant que la commande est `pending` ») ;
- des `example` réalistes pour chaque schéma ;
- **toutes** les réponses d'erreur possibles, pas seulement le cas qui marche ;
- des `tags` pour regrouper les routes par thème.

Le test ultime : un développeur qui n'a jamais vu votre API arrive-t-il à l'utiliser avec la doc seule, sans vous poser de question ?

Générer des clients

Un contrat OpenAPI est lisible par les machines. Elles peuvent donc écrire à votre place le code qui appelle l'API : c'est ce qu'on appelle un **client** ou un **SDK**.

Avantages :

- plus besoin d'écrire les URL, les headers et la sérialisation à la main ;
- l'autocomplétion de l'éditeur connaît toutes les routes et tous les champs ;
- quand le contrat change, on régénère, et les erreurs apparaissent dans l'éditeur plutôt qu'en production.

En Python : openapi-python-client

```
pipx run openapi-python-client generate --path api-ritif.openapi.yaml
```

L'outil génère un package Python complet, avec un module par opération (nommé d'après l'`operationId`) :

```
from api_ritif_client import Client
from api_ritif_client.api.cocktails import get_cocktail

client = Client(base_url="http://api-ritif.portlatch.com/v1")
cocktail = get_cocktail.sync(client=client, cocktail_id=3)
print(cocktail.name)
```

Les noms exacts des modules dépendent du titre du contrat, de ses tags et de ses `operationId`. Consultez le `README.md` généré.

Pour tous les langages : openapi-generator

[openapi-generator](#) sait produire des clients (et des squelettes de serveurs) dans plus de 50 langages : Java, C#, Go, PHP, Kotlin, Swift... Il nécessite Java.

```
npx @openapitools/openapi-generator-cli generate \  
-i api-ritif.openapi.yaml -g php -o ./client-php
```

Les limites

- Le code généré n'est jamais meilleur que le contrat : un contrat approximatif donne un client approximatif.
- On ne modifie **jamais** le code généré à la main : il sera écrasé à la prochaine génération.
- Un contrat qui oublie de déclarer une erreur donne un client qui ne sait pas la gérer.

Les grandes API publiques (Stripe, GitHub, OpenAI...) publient leur contrat OpenAPI et génèrent ainsi une bonne partie de leurs SDK officiels.