

# Outils et méthodes de développement

Johann Chopin

Premier semestre de la 3ème années de la licence Informatique  
et ingénierie du web.



## Introduction aux APIs REST

# Plan

---

## 1. Définition des API REST

- 1.1. Fonctionnement des APIs
- 1.2. Pourquoi REST?
- 1.3. Architecture de REST
  - 1.3.1. Contraintes
  - 1.3.2. Définitions
    - 1.3.2.1. Gestion des URIs
    - 1.3.2.2. Méthodes HTTP
    - 1.3.2.3. Type de médias
      - 1.3.2.3.1. application/xml
      - 1.3.2.3.2. application/json
      - 1.3.2.3.3. multipart/form-data
    - 1.3.3. Codes d'état des réponses HTTP
    - 1.3.4. Gestion des filtres

## 2. Utilisation d'APIs REST

- 2.1. Découverte de Postman
  - 2.1.1. Exécution d'une requête
  - 2.1.2. Collections
  - 2.1.3. Exemple de réponse
  - 2.1.4. Variables et environnements
  - 2.1.5. Envoi de données

## 2.2. Authentification et autorisation

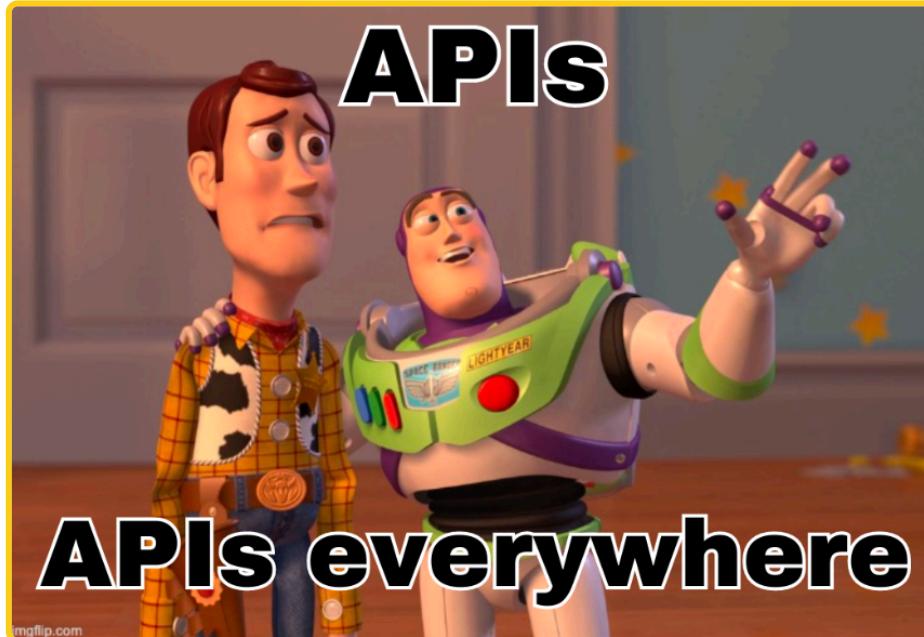
- 2.2.1. Basic Authentication
- 2.2.2. API Keys
- 2.2.3. Token-Based Authentication
  - 2.2.3.1. JWT
- 2.2.4. OAuth 2.0
- 2.2.5. Exercices

## 3. Conception d'APIs REST

- 3.1. Qu'est-ce qu'une documentation API ?
- 3.2. OpenAPI
  - 3.2.1. Spécification
    - 3.2.1.1. Métadonnées
    - 3.2.1.2. Serveurs
    - 3.2.1.3. Endpoints
    - 3.2.1.4. Paramètres
    - 3.2.1.5. Request Body
    - 3.2.1.6. Définition de Schémas
    - 3.2.1.7. Authentification
- 3.3. Exercice
- 4. Ressources

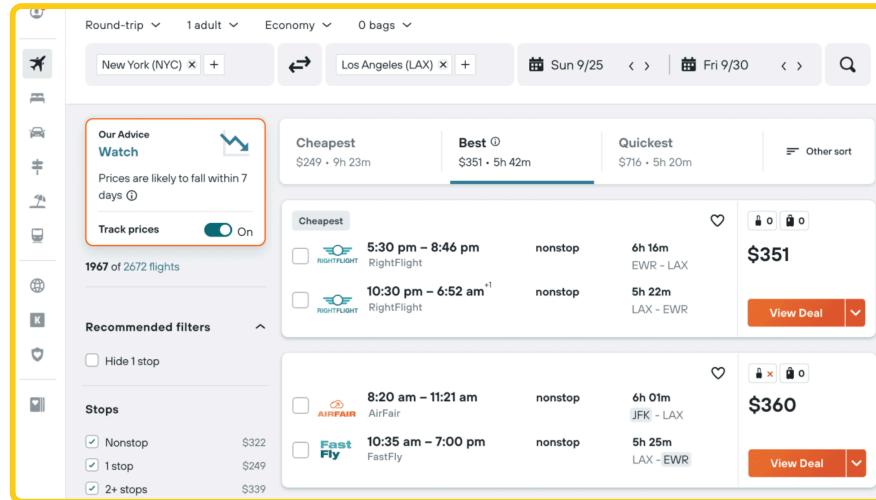
# Définition des API REST

---

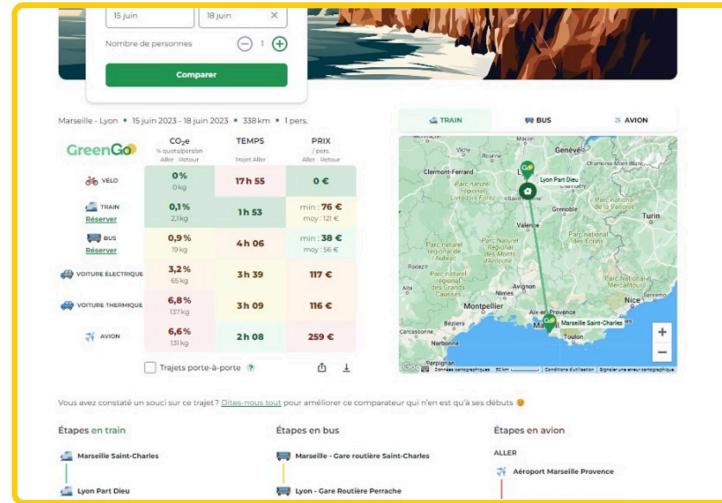


# Fonctionnement des APIs

- Vous voulez partir en vacances et cherchez toutes les possibilités de vols en avion.
- Pour ce faire, vous vous rendez généralement sur un site de comparaison de vols.



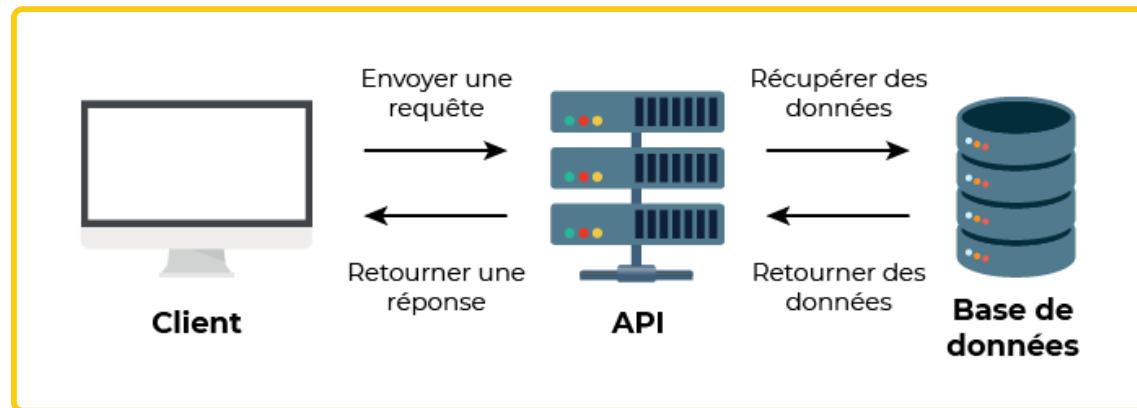
- Finalement, l'avion n'est pas très écologique et les horaires ne sont pas très bons, alors vous essayez de trouver une alternative en train.



- Vous avez facilement accès à des données agrégées sans avoir à rechercher des informations sur le site web de chaque compagnie ferroviaire ou aérienne.
- Vous utilisez et manipulez des données quotidiennement:
  - Informations météorologiques
  - Paiement en ligne
  - Système de cartographie en ligne
  - Les blogs
  - Authentification des utilisateurs (Login with ...)
  - Les réseaux sociaux

Ce partage transparent des données est rendu possible grâce aux interfaces de programmation d'applications (**API**, abréviation de *Application Programming Interface*).

**API:** une façade clairement délimitée par laquelle un logiciel offre des services à d'autres logiciels afin que des données ou des fonctionnalités soient échangées.



**Exemple:** Application de comparateur de trajets.

- Celle-ci ne peut pas accéder directement aux informations de SNCF, Air France, Blablacar, ...
- Heureusement, ces services fournissent une API qui permet de partager leurs données.
  - Exemple: un trajet Lyon-Bordeaux
- Notre application peut donc appeler toutes ces API avec des paramètres précis et les comparer.

## Les API peuvent communiquer:

- d'un logiciel à un logiciel
- d'un client à un serveur
- d'un logiciel à des développeurs

## Il existe deux types principaux d'APIs:

- **Les API privées**
  - Une API privée permet uniquement aux **utilisateurs autorisés** (salariés d'un entreprise, utilisateurs d'une application) d'être utilisé.
  - **Contrôle:** Les API privées permettent à une organisation d'avoir un contrôle total sur ses données et ses services, car non destinées à la consommation externe.
  - **Personnalisation:** Les API privées peuvent être personnalisées pour répondre aux besoins spécifiques d'une organisation, ce qui lui permet de tirer le meilleur parti de l'API.
  - **Sécurité:** Les API privées offrent le niveau de sécurité le plus élevé, car elles ne sont pas exposées aux menaces externes et peuvent être sécurisées à l'aide de protocoles de sécurité internes.
- **Les API publiques**
  - Une API publique est utilisable par d'autres parties, qu'elles soient sur votre application ou non
  - **Adoption généralisée:** Les API publiques sont accessibles à un large éventail de développeurs, ce qui peut conduire à leur adoption généralisée et à leur utilisation dans un grand nombre d'applications.
  - **Facilité d'utilisation:** les API publiques étant ouvertes à tous, elles sont généralement bien documentées et faciles d'utilisation.
  - **Innovation:** Les API publiques peuvent favoriser l'innovation en permettant aux développeurs de créer de nouvelles applications et de nouveaux services en utilisant les données et les services fournis par l'API.

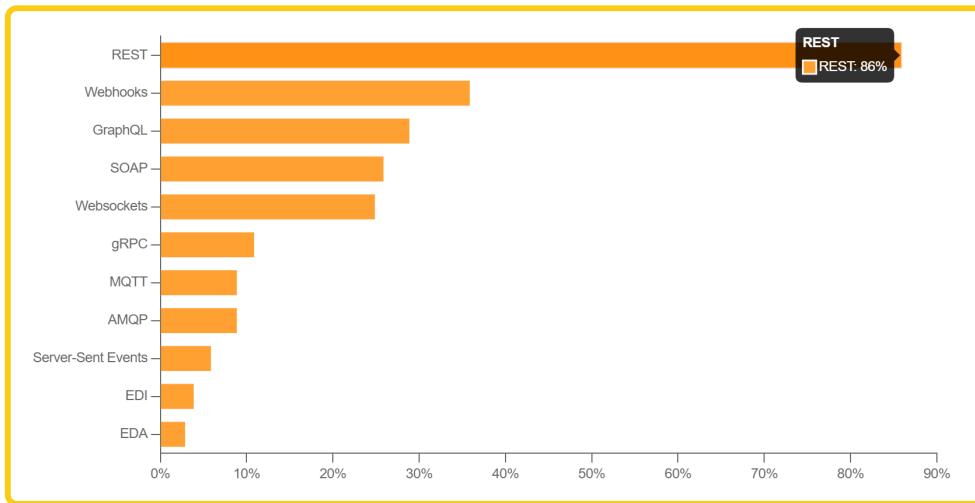
## Il existe aussi:

- **Les API partenaires:** interfaces spécialisées qui permettent aux organisations d'accéder aux données et aux offres de services d'autres entreprises (B2B).
- **Les API composites:** regroupement de plusieurs API en une seule interface offrant aux développeurs une vue unifiée des données provenant de différentes sources.

## En résumé:

- Les API permettent de communiquer des données.
- Elles permettent la communication entre différents composants de votre application et entre votre application et d'autres développeurs, par l'utilisation de requêtes et de réponses.
- Elles donnent un moyen d'accès aux données de façon réutilisable et standardisée.
- Les API publiques sont utilisables par tous sans restriction.
- Les API privées sont utilisables seulement par ceux qui ont un accès et y sont autorisés.

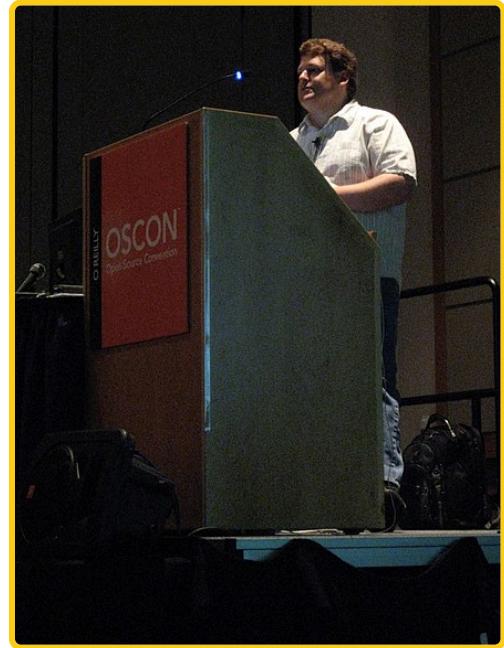
# Pourquoi REST?



Chiffres du 2023 State of the API Report de Postman, basés sur un échantillon de 40.000 développeurs.

## Définition des API REST / Pourquoi REST?

- Le Web a commencé à être utilisé au quotidien en 1993-1994
- Les groupes de travail du W3C ont commencé à travailler ensemble sur la création de descriptions formelles des trois normes principales du Web:
  - **URI:** Uniform Resource Identifier
  - **HTTP:** Hypertext Transfer Protocol, protocole de communication client-serveur
  - **HTML:** HyperText Markup Language, langage de balisage conçu pour représenter les pages web
- Roy Fielding a participé à la création de ces normes notamment HTTP et URI.
- Il définit **REST** dans sa thèse de doctorat en 2000 intitulée "Architectural Styles and the Design of Network-based Software Architectures"



# Representational State Transfer

(Transfert d'État de représentation)

REST définit un ensemble de contraintes relatives au comportement de l'architecture d'un système hypermédia distribué à l'échelle de l'Internet, tel que le Web.

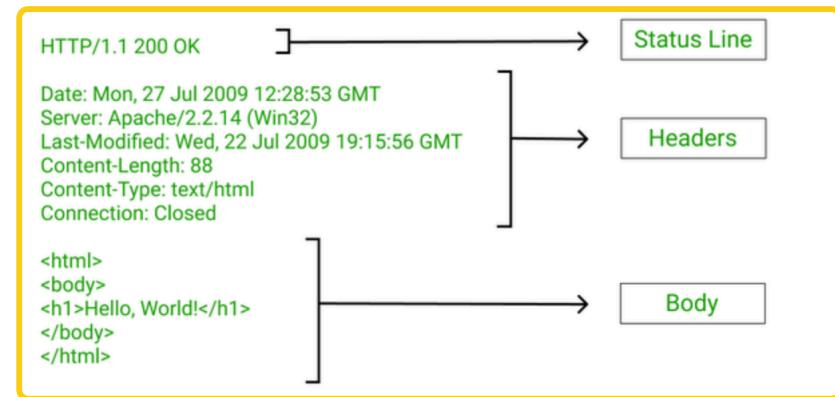
# Architecture de REST

Un système ou API **RESTful** utilise le protocole HTTP.

## HTTP request



## HTTP response



## Contraintes

---

REST définit 6 contraintes:

- **Client/Server:** Les responsabilités sont séparées entre le client et le serveur.
- **Stateless:** Entre deux requêtes successives, la communication client–serveur s'effectue sans conservation de l'état de la session de communication. Les requêtes du client contiennent donc toute l'information nécessaire pour que le serveur puisse y répondre.
- **Cacheable:** Les clients et les serveurs intermédiaires peuvent mettre en cache les réponses. Les réponses doivent donc, implicitement ou explicitement, se définir comme pouvant être mises en cache ou non.
- **Uniforme Interface:** L'utilisation des mêmes normes est fondamentale dans la conception d'un système RESTful.
- **Layered system:** Un client ne peut généralement pas savoir s'il est connecté directement au serveur final ou à un intermédiaire en cours de route.
- **Code on demand (optionnel):** Les serveurs peuvent temporairement étendre ou modifier les fonctionnalités d'un client en lui transférant du code exécutable (exemple: *applets Java* ou des scripts *JavaScript*).

Ces contraintes permettent à un système (RESTful) une meilleure:

- Performance
- Extensibilité
- Simplicité
- Évolutivité
- Visibilité
- Portabilité
- Fiabilité

## Définitions

---

Les API REST basées sur HTTP sont définies par:

- un **URI** de base, comme `http://api.example.com/v1/collection/`
- des **méthodes HTTP** standards comme `GET`, `POST`, `PUT`, `PATCH` et `DELETE`
- un **type de médias** pour les données permettant une transition d'état comme `application/json`, `application/xml`, `multipart/form-data`

## Gestion des URIs

Dans une API REST, les données sont représentées par ce qu'on appelle des **ressources** et des **collections**.

- Une **ressource** est un objet suffisamment important pour être référencé en lui-même. Elle possède
  - des données,
  - des relations avec d'autres ressources
  - des méthodes qui opèrent sur elle pour permettre l'accès et la manipulation des informations associées
  - **Exemple:** un utilisateur (`user`), une localisation (`location`), un personnage (`character`)
- Une **collection** est un groupe de ressources.
  - On s'y réfère avec la forme au **pluriel** du nom de la ressource.
  - **Exemple:** `characters` est une collection de `character`.

**À vous de jouer:** Vous souhaitez créer une API pour une boulangerie qui vend différents types de pain. Quelles seraient les **collections**, les **ressources** et leurs **informations complémentaires** impliquées dans l'achat d'un pain par un client auprès d'un vendeur ?



**Collections:** Clients , StaffMembers , Breads , Orders

En règle générale, il n'existe que deux chemins pour une ressource donnée lorsqu'il s'agit d'un endpoint API RESTful.

- `/{collection}` : utilisé pour identifier une collection.
- `/{collection}/{ressource-id}` : utilisé pour identifier une ressource spécifique de cette collection.

Ces chemins peuvent être imbriqués afin de créer une relation entre les ressources:

- `/bakeries/42/clients`
- `/bakeries/42/staff-members/4`

## Quelques bonnes pratiques:

- Utiliser des noms pour représenter les ressources, **pas de verbes**.
- Les **collections** doivent toujours être **au pluriel**
- Utiliser des **lettres minuscules** dans les URIs
- Utiliser des traits d'union `-` pour améliorer la lisibilité des URIs, ne pas utiliser d'underscores `_` :
  - **✗** `/bakeries/42/opening_hours`
  - **✓** `/bakeries/42/opening-hours`
- Ne pas utiliser d'extensions de fichiers car elles sont peu esthétiques et n'apportent aucun avantage.

💡 Si on ne peut utiliser de verbes, comment effectuer des actions sur des données avec une API REST?

## Méthodes HTTP

Méthode HTTP	Action	Opération CRUD
GET	Permet de lire une ressource	READ
POST	Permet de créer une ressource	CREATE
PUT	Permet de modifier la valeur d'une ressource	UPDATE
PATCH	Permet de modifier une partie de la valeur d'une ressource	UPDATE
DELETE	Permet de supprimer une ressource	DELETE

**À vous de jouer:** Donnez une URIs RESTful ainsi que son code HTTP permettant de réaliser les actions suivantes:

- Récupérez la liste des `staff-member` de la `bakery` identifié par `{bakery-id}` :  
→ `GET /bakeries/{bakery-id}/staff-members`
- Supprimez de vos données un membre du staff de la boulangerie identifié `42` :  
→ `DELETE /bakeries/42/staff-members/{staff-member-id}`
- Dans cette même boulangerie ajoutez une nouvelle recette de pain `bread` :  
→ `POST /bakeries/42/breads`
- Un `client` passe commande (`order`) dans une boulangerie:  
→ `POST /bakeries/{bakery-id}/clients/{client-id}/orders`
- Le `client` modifie sa commande ayant l'ID unique `1` :  
→ `PUT /orders/1`
- La boulangerie `42` pert un client, il faut le supprimer:
  - → `DELETE /bakeries/{bakery-id}/clients/{client-id}`
  - → `DELETE /clients/{client-id}`

## Type de médias

Les méthodes HTTP `GET`, `POST`, `PUT`, `PATCH` et parfois même `DELETE` impliquent le transfert de données du client vers le serveur et du serveur vers le client. Mais quel format utiliser ?

REST utilise généralement 3 types de médias HTTP:

- `application/xml`
- `application/json`
- `multipart/form-data`

L'EXtensible Markup Language est un métalanguage de **balisage générique**. Sa syntaxe permet de définir différents langages avec pour chacun son vocabulaire et sa grammaire, comme **XHTML, RSS ou SVG**:

```
<breads>
  <bread>
    <type>baguette</type>
    <price>0.95</price>
    <cooking-time>15</cooking-time>
  </bread>
  <bread>
    <type>épeautre</type>
    <price>1.35</price>
    <cooking-time>25</cooking-time>
  </bread>
</breads>
```

Le **JavaScript Object Notation (JSON)** est un format de données textuel dérivé de la notation des objets du langage JavaScript:

```
{  
  "breads": [  
    {  
      "type": "baguette",  
      "price": 0.95,  
      "cooking-time": 15  
    },  
    {  
      "type": "épeautre",  
      "price": 1.35,  
      "cooking-time": 25  
    }  
  ]  
}
```

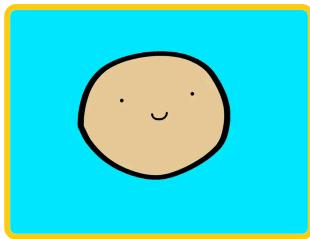
## multipart/form-data

---

Le type de média `multipart/form-data` doit être utilisé pour soumettre des formulaires contenant des fichiers, des données non ASCII et des données binaires.

Il est souvent utilisé pour envoyer des images :

```
POST /users/{id}/avatar
```



## Codes d'état des réponses HTTP

La première ligne d'un message de réponse HTTP est la ligne d'état, qui comprend la version du protocole, suivie d'un **code d'état** numérique et de la phrase textuelle associée:

HTTP/1.x 200 OK

Code	Catégorie	Description
<b>1xx</b>	Information	Demande reçue, poursuite de la procédure
<b>2xx</b>	Succès	L'action a été reçue, comprise et acceptée avec succès.
<b>3xx</b>	Redirection	Une action supplémentaire doit être entreprise afin de compléter la demande.
<b>4xx</b>	Erreur du client	La demande contient une mauvaise syntaxe ou ne peut être satisfaite
<b>5xx</b>	Erreur du serveur	Le serveur n'a pas réussi à répondre à une demande apparemment valide.

Code	Message	Description
<b>200</b>	OK	Requête traitée avec succès. La réponse dépendra de la méthode de requête utilisée.
<b>201</b>	Created	Requête traitée avec succès et création d'une ressource.
<b>202</b>	Accepted	Requête traitée, mais sans garantie de résultat.
<b>204</b>	No Content	Requête traitée avec succès mais pas d'information à renvoyer.
<b>400</b>	Bad Request	La syntaxe de la requête est erronée.
<b>401</b>	Unauthorized	Une authentification est nécessaire pour accéder à la ressource.

Code	Message	Description
<b>403</b>	Forbidden	Le serveur a compris la requête, mais refuse de l'exécuter car les droits d'accès ne permettent pas d'accéder à la ressource.
<b>404</b>	Not Found	Ressource non trouvée.
<b>500</b>	Internal Server Error	Erreur interne du serveur.

## Gestion des filtres

- Dans le cadre des APIs, un filtre est un moyen de restreindre une recherche dans une collection de ressources.
- ?
- Comment récupérer une liste des 30 derniers utilisateurs, ou tous les articles contenant le mot `boulangerie` ?

💡 REST n'apporte pas de standard à l'implémentation de filtres dans une requête.

2 flows sont néanmoins très populaires parmi la plupart des API :

- Ajouter les filtres dans le body d'une requête `POST` pour l'utilisation de filtres assez complexes (⚠ la méthode POST n'est pas **cacheable**).
- Ajouter les filtres via des query strings dans l'URI de la requête:

```
GET /users?orderBy=desc&perPage=30
```

## Example d'implémentation avec l'API REST de Strapi:

- Pagination:

```
#/ Renvoyer que 10 entrées sur la page 1  
GET /users?pagination[page]=1&pagination[pageSize]=10
```

- Filtres:

```
#/ Trouver les utilisateurs dont le prénom est 'John'  
GET /users?filters[username][$eq]=John
```

Exemple de syntaxe de query strings permettant un filtrage avancé: <https://docs.strapi.io/dev-docs/api/rest/filters-locale-publication#filtering>.

**À vous de jouer:** En suivant la syntaxe d'une API REST Strapi, donnez les requêtes permettant de récupérer:

- Tous les articles contenant le mot `boulangerie` dans son champ `description`.
- Les boulangeries comptant entre 5 et 10 employés (utilisez le champ `employees_amount`).
- Tous les auteurs `authors` dont le `firstName` commence par la lettre `a` ou `b`.

## Définition des API REST / Architecture de REST / Gestion des filtres

```
#/ Correction  
  
#/ Tous les articles contenant le mot `boulangerie` dans son champ `description`:  
  
#/ Les boulangeries comptant entre 5 et 10 employés (utilisez le champ `employees_amount`):  
  
#/ Tous les auteurs `authors` dont le `firstName` commence par la lettre `a` ou `b`:  
  
#
```

# Utilisation d'APIs REST

---



## Utilisation d'APIs REST

Il existe actuellement de nombreux outils facilitant la conception, la documentation, le débogage, la simulation et les tests sur de APIs:



**POSTMAN**



**Insomnia**



**APIDOG**



**Hoppscotch**

Dans les documentations, vous verrez souvent une commande curl comme référence:

```
curl -L \
-X DELETE \
-H "Accept: application/json" \
-H "Authorization: Bearer token" \
https://api.com/collections/ressource \
-d '{"key": "value"}'
```

# Découverte de Postman

Créez vous un compte sur **Postman**.

The screenshot shows the Postman application interface. On the left, there's a sidebar with 'Notion's Public Workspace' selected. Under 'Collection', there's a tree view of 'Notion API' with 'Databases' expanded, showing 'Retrieve a database'. The main panel shows a 'GET' request to 'https://api.notion.com/v1/databases/{id}'. The 'Params' tab shows a 'id' parameter with value '({DATABASE\_ID})'. The 'Body' tab shows a JSON response with a single object containing a 'Publisher' field. The right side of the screen displays the 'Documentation' for the 'Retrieve a database' endpoint, which includes the URL 'https://api.notion.com/v1/databases/{id}', a description of the endpoint, an 'Authorization' section using a 'Bearer Token' helper from the 'Notion API' collection, and a 'Request Header' section with 'Notion-Version' set to '22-02-22'. A 'Path Variables' section also shows the 'id' variable with its description.

```
1 {
2   "Publisher": [
3     {
4       "id": "33E824Ph",
5       "name": "Publisher",
6       "type": "select",
7       "select": {
8         "options": [
9           {
10             "id": "c5ee499a-f307-4176-99ee-6e424fa89afa",
11             "name": "VTL",
12             "color": "default"
13           }
14         ]
15       }
16     ]
17   }
18 }
```

<https://web.postman.co/>

Nous allons faire nos premiers pas avec Postman en utilisant l'API REST de GitHub.

↗ <https://docs.github.com/fr/rest>

⚠ L'API de GitHub possède de très nombreuses features. Nous nous concentrerons sur les endpoints des **utilisateurs** ainsi que les **gists**

## Exécution d'une requête

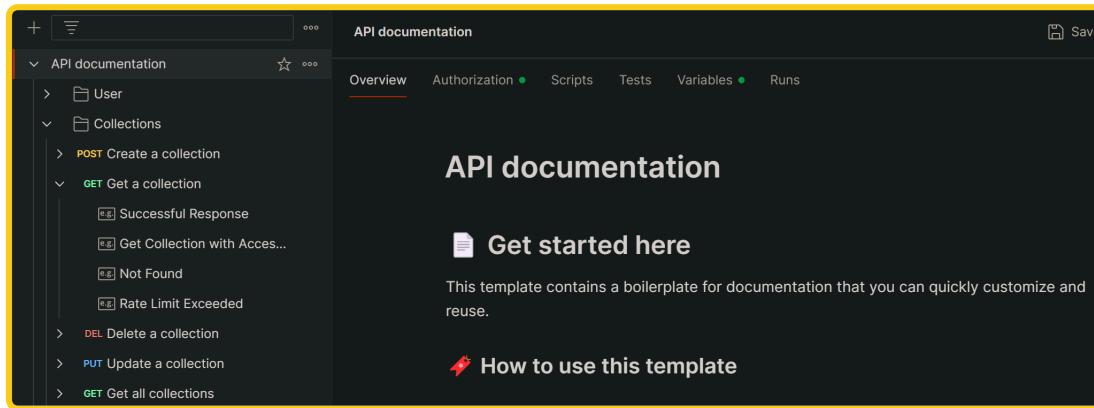
**À vous de jouer:** Récupérez via API vos informations de votre profile GitHub.

1. Quel endpoint devez-vous appeler pour récupérer les informations de votre utilisateur ?
2. Ajoutez cette requête à Postman et exécutez-la.
3. Analyse de la requête et de sa réponse.
4. Appelez la requête en demandant une réponse au format `xml` .

```
{  
  "message": "Unsupported 'Accept' header: 'application/xml'. Must accept 'application/json'.",  
  "documentation_url": "https://docs.github.com/v3/media",  
  "status": "415"  
}
```

## Collections

Dans Postman, les **collections** constituent le standard permettant l'**organisation des APIs**. Grâce aux collections, vous pouvez lier des éléments d'API apparentés pour faciliter l'édition, le partage, le test et la réutilisation.



**À vous de jouer:** Déplacez votre requête dans une collection nommée, dans un sous dossier portant le nom de la collection REST.

## Exemple de réponse

- Les exemples montrent les endpoints de votre API en action et donnent plus de détails sur le fonctionnement des requêtes et des réponses.
- Vous pouvez ajouter un exemple à une requête en enregistrant une réponse, ou vous pouvez créer un exemple avec une réponse personnalisée pour illustrer un cas d'utilisation spécifique.



The screenshot shows the Postman interface with a yellow border around the main content area. At the top, there are tabs for Body, Cookies (1), Headers (6), and Test Results. On the right, status information is displayed: 200 OK, 144 ms, 2.06 KB, Save as example, and a help icon. Below the tabs, there are buttons for Pretty, Raw, Preview, Visualize (with a green dot), and JSON. The JSON response body is shown with line numbers:

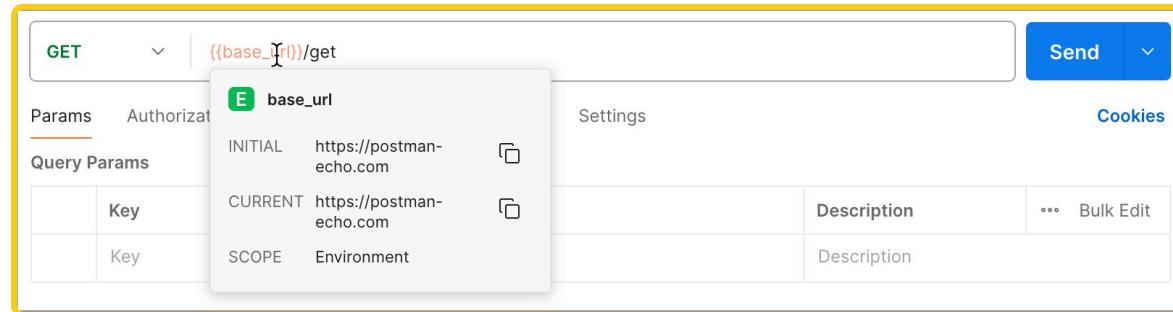
```
1 {  
2   "args": {},  
3   "data": {  
4     "message": "Go to Visualize tab on the right ➡ located alongside the Pretty, Raw, and Preview options. We have added the visualization code to the Tests tab for a request.",  
5     "contacts": [  
6       {  
7         "name": "Diana Green",  
8         "email": "Kyler79@gmail.com"  
9       },  
10      {  
11        "name": "Ricardo Blunsch"  
12      }  
13    ]  
14 }
```

**À vous de jouer:** Ajoutez un exemple à votre requête illustrant une réponse **404**. L'utilisateur `__unknown__` n'existe pas encore.

## Variables et environnements

- Les variables permettent de stocker et de réutiliser des valeurs dans Postman.
- En stockant une valeur en tant que variable, il est possible de la référencer dans des collections, environnements, requêtes et scripts.

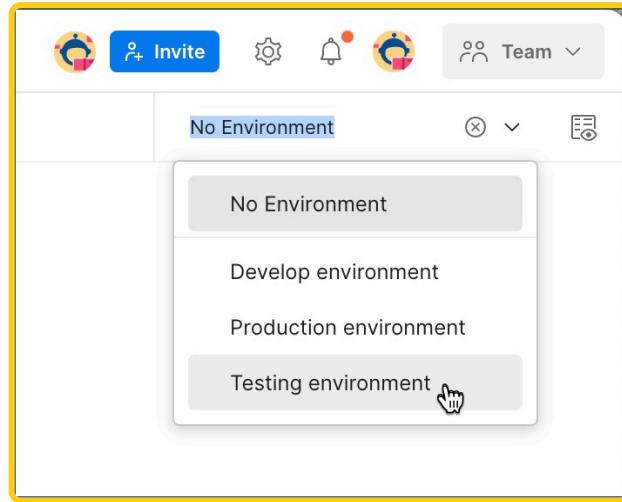
**Exemple:** Si vous avez le même URL dans plusieurs requête, mais que l'URL peut changer ultérieurement, vous pouvez stocker l'URL dans une variable `base_url` et y faire référence à l'aide de la syntaxe entre doubles crochets `{{base_url}}`.

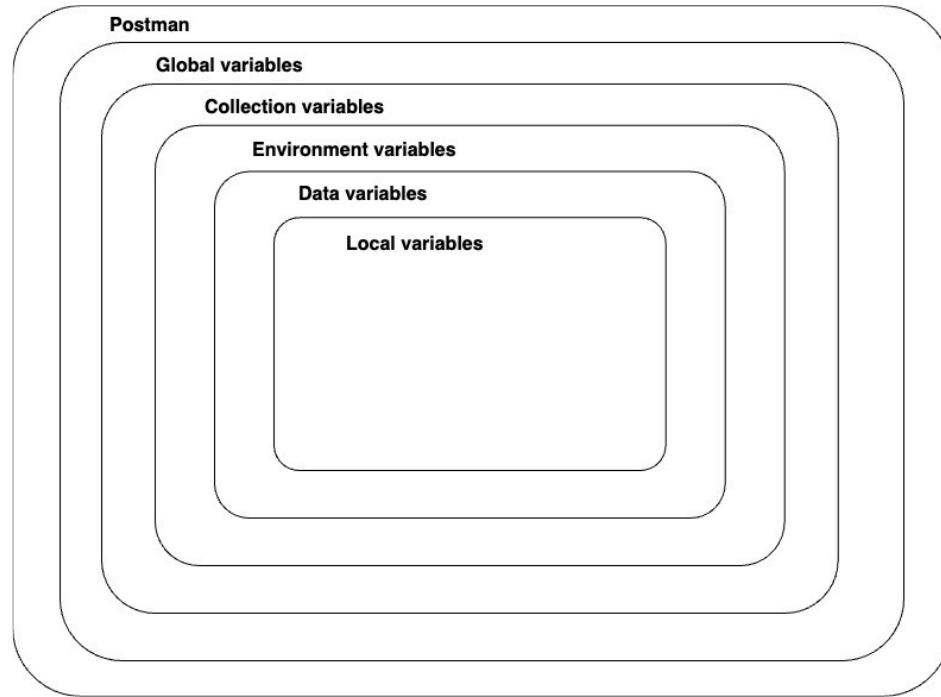


Chacune des variables possèdent une **valeur initiale** et une **valeur courante** :

- La **valeur initiale** est une valeur synchronisée avec les serveurs de Postman et peut donc être partagée avec l'équipe.
- La **valeur courante** est utilisée lors de l'envoi d'une requête. Il s'agit de valeurs locales, qui ne sont pas synchronisées avec les serveurs de Postman.

- Il est possible de regrouper des variables dans différents **environnements**.
- Lorsque vous passez d'un environnement à l'autre, toutes les variables de vos requêtes et de vos scripts utilisent les valeurs de l'environnement actuel.
- Cela est utile pour utiliser des valeurs différentes fonction du contexte: serveur de test ou serveur de production.



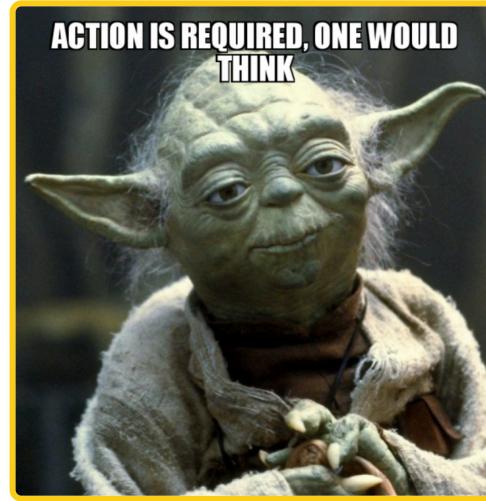


## À vous de jouer:

1. Créez une variable dans votre collection contenant votre nom d'utilisateur.
2. Utilisez cette variable dans votre requête `GET /users/{id}`.
3. Modifiez sa valeur courante pour récupérer les informations d'un autre utilisateur.
4. Définir 2 environnements `github-api-test` et `github-api-test` :
  1. Y ajouter une variable `version`, l'une sera la dernière version de l'API et l'autre sera une fausse version (Consultez la documentation sur la version de l'API GitHub: <https://docs.github.com/fr/rest/about-the-rest-api/api-versions>).
  2. Utiliser cette variable dans votre requête de sorte à pouvoir définir qu'elle version d'API utiliser.

## Envoi de données

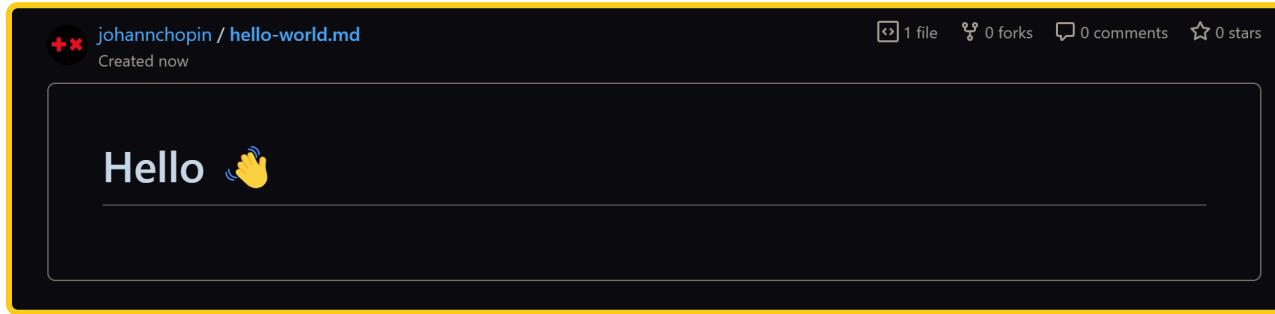
---



Utilisons la section Body d'une requête dans Postman.

Nous allons utiliser l'API liée aux Gists dans GitHub: [/gists](#).

Gist est un service de GitHub permettant de simplement partager des snippets de code avec d'autres personnes.



## À vous de jouer:

1. Créez un simple gist au format markdown depuis l'interface de GitHub:  
<https://gist.github.com>
2. Ajoutez dans Postman un nouveau **Dossier** contenant une requête appelant la liste des gists de votre utilisateur.
3. Ajoutez une requête permettant de créer un nouveau **gist public** au format markdown.
4. Exécutez la requête. Vous devriez recevoir une erreur **401**.

# Authentification et autorisation

- L'**authentification** API permet de vérifier l'identité d'un utilisateur afin qu'il puisse accéder à une API.
- L'**autorisation** permet de définir ce à quoi un utilisateur peut ou ne peut pas accéder.



L'authentification est utilisée pour :

- Vérifier l'identité d'un client ou d'un utilisateur.
- Permettre aux clients et utilisateurs autorisés d'accéder à une API.
- Empêcher tout accès non autorisé.

L'autorisation est utilisée pour :

- Accorder **l'accès et l'exposition à des ressources** ou des données particulières pour différents utilisateurs.
- Régir les actions que les différents utilisateurs et clients peuvent effectuer avec une API.
- Appliquer les politiques de contrôle d'accès définies.

**À vous de jouer:** Quelles codes HTTP conviendraient pour une erreur d'authentification et une erreur d'autorisation?

Code	Message	Description
<b>401</b>	Unauthorized	Une authentification est nécessaire pour accéder à la ressource.
<b>403</b>	Forbidden	Le serveur a compris la requête, mais refuse de l'exécuter car les droits d'accès ne permettent pas d'accéder à la ressource.

Il existe quatre façons principales de mettre en œuvre l'authentification dans une API REST :

- Basic Authentication
- API Keys
- Token-Based Authentication
- OAuth 2.0

## Basic Authentication

- Basic Authentication est un schéma d'authentification simple intégré au protocole HTTP.
- Le client envoie des requêtes HTTP avec le header `Authorization` qui contient le mot `Basic` suivi d'un espace et d'une chaîne de caractères `username:password` encodée en base64.
- Exemple de header: `Authorization: Basic ZGVtbzpwQDU1dzByZA==`

! Comme le base64 est facilement décodé, l'authentification de base ne devrait être utilisée qu'avec d'autres mécanismes de sécurité tels que HTTPS/SSL.

## API Keys

Une clé API est un token (jeton) que le client fournit lorsqu'il effectue des appels API. Il s'agit d'une longue chaîne alphanumérique identifiant de manière unique un client ou une application accédant à une API.

- La clé peut être envoyée via une query string :

```
GET /collections?api_key=abcdef12345
```

- ou en tant que header d'une requête :

```
GET /something HTTP/1.1  
X-API-Key: abcdef12345
```

- ou en tant que cookie HTTP :

```
GET /something HTTP/1.1  
Cookie: X-API-KEY=abcdef12345
```

⚠ Les clés API sont censées être un secret que seuls le client et le serveur connaissent. Comme la Basic Authentication, l'authentification basée sur les clés API n'est considérée comme sûre que si elle est utilisée avec d'autres mécanismes de sécurité tels que HTTPS/SSL.

-  L'authentification par API key est d'une grande simplicité. La méthode utilise une clé unique qui permet de s'authentifier simplement en incluant la clé.
-  L'authentification par API key est très répandue.
-  Si la clé est rendue publique, quelqu'un peut se faire passer pour vous
-  Il s'agit d'un moyen simple de gérer l'authentification pour les API qui n'ont pas besoin d'autorisations d'écriture, tout en limitant les risques.

## Token-Based Authentication

- Un système d'authentification HTTP qui implique des tokens de sécurité appelés **bearer tokens**.
- Le bearer token est une **chaîne cryptée**, généralement générée par le serveur en réponse à une demande de connexion.
- Le client doit envoyer ce token dans le header `Authorization` lorsqu'il demande des ressources protégées:

```
Authorization: Bearer <token>
```

💡 Dans le domaine du développement web, les « web tokens » font presque toujours référence aux JWTs (JSON Web Token).

## JWT

- JSON Web Token (JWT) est une **norme ouverte** (RFC 7519) qui définit un moyen compact et autonome de transmettre en toute sécurité des informations entre des parties sous la forme d'un objet JSON.
- Ces informations sont vérifiables et fiables car elles sont signées numériquement.

Structure of a JSON Web Token (JWT)



↗ <https://jwt.io>

⚠ Bien qu'un JWT soit protégées contre la falsification, ses données sont lisibles par n'importe qui. Ne mettez pas d'informations secrètes dans le payload ou header d'un JWT, à moins qu'elles ne soient cryptées.

- Dans certains cas, JWT permet de mettre en place un mécanisme d'autorisation sans état.
- Les routes protégées du serveur vérifieront la présence d'un **JWT valide (vérification de la signature)**.
- Si le JWT contient les données nécessaires, la nécessité d'interroger la base de données pour certaines opérations peut être réduite.

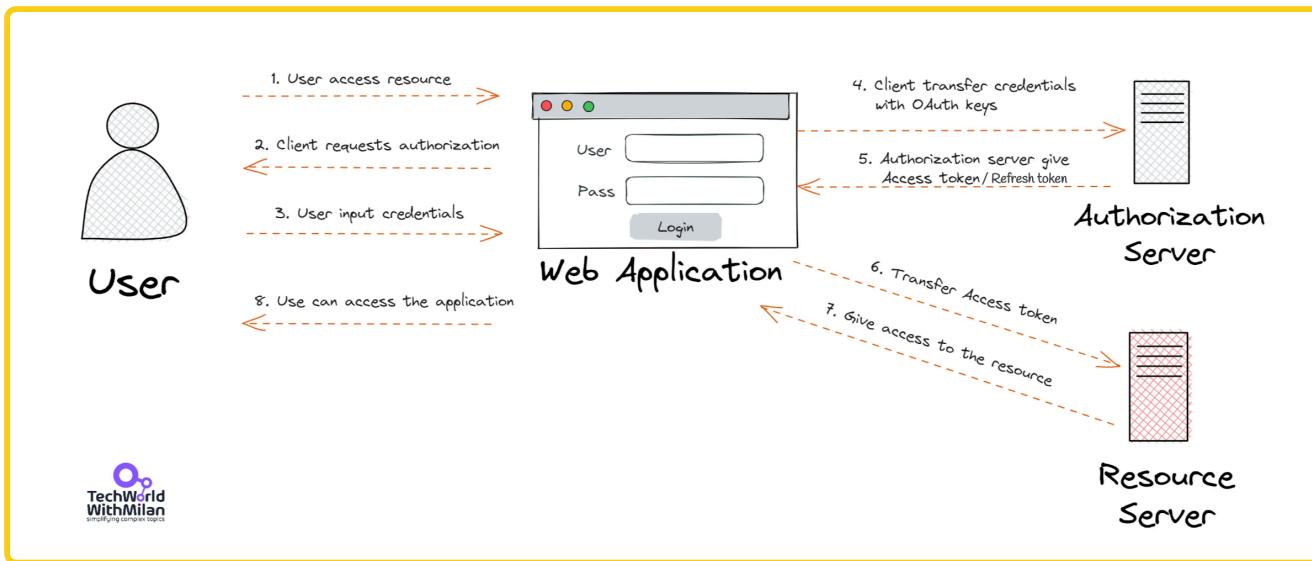
Veiller à ce qu'un token JWT ne soient **pas trop volumineux** si vous l'envoyez par le biais d'un header HTTP. Certains serveurs n'acceptent pas plus de **8KB** dans les headers

## Avantages et inconvénients:

- **👍 Sécurité:** Les JWT sont signés numériquement à l'aide d'un secret ou d'une paire de clés publique/privée, ce qui les protège contre toute modification par le client ou un attaquant.
- **👍 Stockage chez le client:** Vous générez des JWT sur le serveur et les envoyez au client. Ce dernier soumet ensuite la JWT à chaque demande. Cela permet d'économiser de l'espace dans la base de données.
- **👍 Efficace et sans état:** La vérification d'un JWT est rapide car elle ne nécessite pas de consultation de la base de données. Cela est particulièrement utile dans les grands systèmes distribués.
- **👎 Non révocable:** En raison de leur nature autonome et du processus de vérification sans état, il peut être difficile de révoquer un JWT avant qu'il n'expire naturellement.
- **👎 Dépendance à une clé secrète:** La création d'un JWT dépend d'une clé secrète. Si cette clé est compromise, l'attaquant peut fabriquer son propre JWT que l'API acceptera.

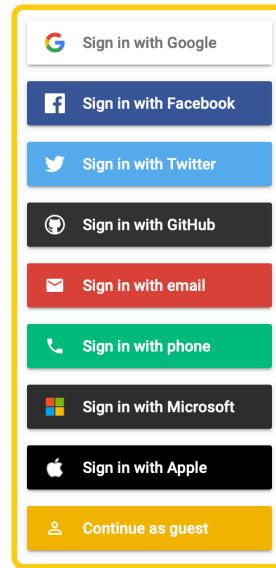
## OAuth 2.0

- OAuth 2.0 (Open Authorization 2.0) est un protocole largement adopté et normalisé pour l'authentification et l'autorisation dans les APIs RESTfuls.



- **Access token**: Les tokens d'accès sont des informations d'identification de courte durée qui représentent l'autorisation accordée au client.
- **Refresh token**: Les tokens d'accès ont une durée de vie limitée. Pour prolonger la session de l'utilisateur sans nouvelle autorisation, le client peut utiliser un refresh token.

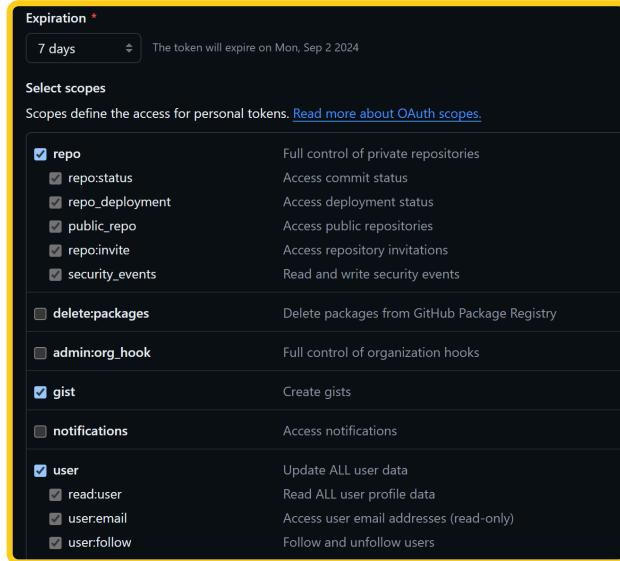
Le protocole OAuth est largement utilisé dans la mise en œuvre de Single Sign-On (SSO).



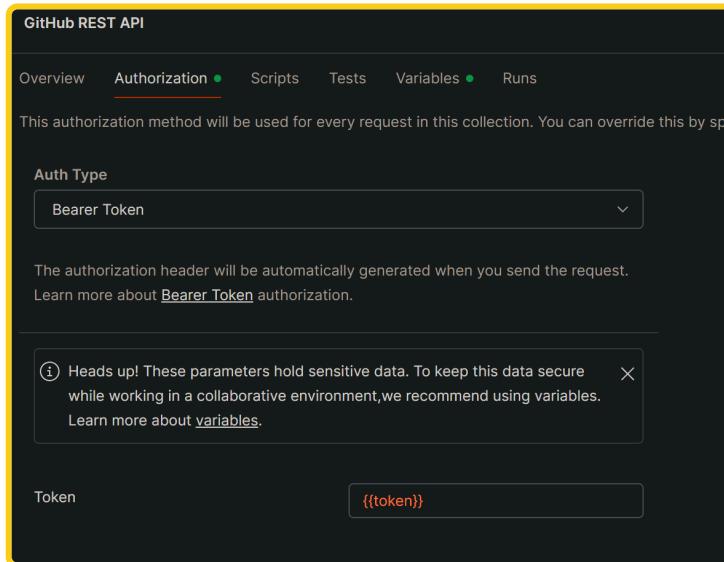
## Exercices

Quel type d'authentification utilise l'API de GitHub ? Référez-vous à la documentation.

1. Créez vous un **Classic Personal access tokens** sur GitHub: <https://github.com/settings/tokens>



2. Ajoutez ce token dans votre **Collection** Postman, dans la section **Authorization**:



3. Utilisez ce token pour créer un gist au format markdown.

4. Éditez ce gist. Quelle réponse et status code est attendu conformément aux standards d'une API RESTful?
5. Supprimez le gist! Quelle réponse et status code est attendu conformément aux standards d'une API RESTful?

# Conception d'APIs REST

Nous avons vu comment utiliser une API externe, voyons maintenant comment documenter nous-même une API REST.



# Qu'est-ce qu'une documentation API ?

- La documentation de l'API est un **ensemble d'instructions lisibles par l'Homme (et la machine)** pour l'utilisation et l'intégration d'une API.
  
- La documentation de l'API comprend:
  - des informations détaillées sur les endpoints
  - les méthodes
  - les ressources
  - les protocoles d'authentification
  - les paramètres
  - les headers
  - des exemples de requêtes et de réponses.

Mais comment partagez une telle documentation?



Open API  
Specification

# OpenAPI

- Les descriptions d'API lisibles par machine sont aujourd'hui omniprésentes et OpenAPI est la **norme industrielle** la plus largement adoptée pour décrire les nouvelles API.
- La spécification OpenAPI (**OAS**) permet de décrire une API distante accessible via HTTP ou des protocoles de type HTTP. Cette description, qui peut être stockée sous la forme d'**un ou de plusieurs documents JSON ou YAML**, est appelée **description OpenAPI (OAD)**.

The screenshot shows the Swagger Editor interface. On the left, the OpenAPI specification is displayed in a code editor:

```
1 openapi: "3.0.0"
2 info:
3   version: 1.0.0
4   title: Swagger Petstore
5   license:
6     name: MIT
7   servers:
8     - url: http://petstore.swagger.io/v1
9   paths:
10  /pets:
11    get:
12      summary: List all pets
13      operationId: listPets
14      tags:
15        - pets
16      parameters:
17        - name: limit
18        in: query
19        description: How many items to return at one time (max 100)
20        required: false
21      schema:
22        type: integer
23        format: int32
24    responses:
25      '200':
26        description: A paged array of pets
27        headers:
28          x-next:
29            description: A link to the next page of responses
30        schema:
31          type: string
32          content:
```

On the right, the UI generated from the specification is shown for the **pets** endpoint:

- GET /pets** List all pets
- POST /pets** Create a pet
- GET /pets/{petId}** Info for a specific pet

- La description d'une API dans un format lisible par une machine permet aux outils automatisés de la traiter :
  - **Validation de la description et linting** : Vérifier que votre fichier de description est syntaxiquement correct et qu'il respecte une version spécifique de la spécification ainsi que les directives de formatage du reste de votre équipe.
  - **Validation des données** : Vérifier que les données qui transitent par votre API sont correctes, pendant le développement et une fois déployées.
  - **Génération de documentation** : Création d'une documentation traditionnelle lisible par l'homme, basée sur la description lisible par la machine.
  - **Génération de code** : Créez le code du serveur et du client dans n'importe quel langage de programmation, ce qui évite aux développeurs d'avoir à effectuer la validation des données ou à écrire le code du SDK, par exemple.
  - **Mock Servers** : Créez des serveurs mockés fournissant des exemples de réponses que vous et vos clients pouvez commencer à tester avant d'écrire une seule ligne de code.
  - **Analyse de sécurité** : Découvrez les éventuelles vulnérabilités dès la phase de conception de l'API, plutôt que beaucoup, beaucoup plus tard.

 OAS est une spécification au format non propriétaire.

- Elle était à l'origine basé sur la spécification **Swagger 2.0**, donnée par SmartBear Software en 2015.
- Actuellement, l'OAS est maintenu, développé et promu par l'OpenAPI Initiative (**OAI**), un consortium d'experts industriels doté d'une structure de gouvernance ouverte sous la tutelle de la **Fondation Linux**.
- Toutes **les réunions et décisions sont publiques** et les modifications de l'OAS peuvent être proposées et discutées par **tout le monde**.

## Spécification

Apprenons la syntaxe d'**OAS** en utilisant l'éditeur en ligne <https://editor-next.swagger.io>.

- Remplissez la configuration minimale pour un document **OAS** au format YAML:

```
openapi: 3.0.3
info:
  title: My API - OpenAPI 3.0
  version: 0.0.1
paths: {} # No endpoints defined
```

## Métadonnées

- Chaque définition d'API doit inclure la version OAS sur laquelle cette définition est basée :

```
openapi: 3.0.3 # Suit la norme SemVer
```

- La section `info` contient des informations sur l'API :
  - `title` : Nom de l'API
  - `description` (optionnelle) : Complément d'information sur l' API au format text ou markdown
  - `version` : Chaîne de caractères arbitraire qui spécifie la version de l'API

**À vous de jouer:** Remplissez les métadonnées comme si vous étiez éditeur de l'API de GitHub.

## Serveurs

- La section `servers` spécifie le serveur API et l'URL de base. Vous pouvez définir un ou plusieurs serveurs, tels que production et sandbox:

```
servers:  
  - url: http://api.example.com/v1  
    description: Optional server description, e.g. Main (production) server  
  - url: http://staging-api.example.com  
    description: Optional server description, e.g. Internal staging server for testing
```

- Tous les endpoints de l'API sont relatifs à l'URL du serveur

## Endpoints

La section `paths` définit **les endpoints** de votre API et **les méthodes HTTP** prises en charge:

```
paths:  
  /users:  
    get:  
      # ...  
    post:  
      # ...
```

OpenAPI 3.0 prend en charge les méthodes `get`, `post`, `put`, `patch`, `delete`, `head`, `options` et `trace`.

Les endpoints peuvent être accompagnés d'un court `summary` facultatif et d'une `description` plus longue à des fins de documentation.

```
paths:  
  /users:  
    get:  
      summary: Get all the users  
      description: Get the complete list of users ordered by creation date  
      parameters:  
        # ...  
    responses:  
      # ...
```

## Paramètres

Il est possible de décrire un paramètre, en indiquant son nom (`name`), son emplacement (`in`), son type de données (`schema`) et d'autres attributs, tels que `description` ou `required`.

```
paths:  
  /users/{userId}:  
    get:  
      summary: Get a user by ID  
      parameters:  
        - in: path  
          name: userId  
          schema:  
            type: string  
            required: true  
          description: Numeric ID of the user to get
```

`in` peut contenir la valeur `path`, `header`, `query` ou `cookie`.

`type` peut contenir la valeur `string`, `number`, `integer`, `boolean`, `array` ou `object`.

**À vous de jouer:** Documentez l'endpoint de l'API GitHub `GET /users/{username}/gists`, de sorte à lister tous les paramètres du **path** ainsi que des **query strings**.

## Request Body

Il est possible de décrire très précisément le body attendu d'une requête:

```
paths:  
  /users:  
    post:  
      summary: Add a new user  
      requestBody:  
        description: Optional description in *Markdown*  
        required: true  
        content:  
          application/json:  
            schema:  
              $ref: '#/components/schemas/User'  
          application/xml:  
            schema:  
              $ref: '#/components/schemas/User'  
          application/x-www-form-urlencoded:  
            schema:  
              $ref: '#/components/schemas/UserForm'
```

Pour éviter la duplication du code, vous pouvez placer les définitions communes dans la section global `components` et les référencer à l'aide de `$ref` :

```
paths:  
  /users:  
    post:  
      requestBody:  
        content:  
          application/json:  
            schema:  
              $ref: '#/components/schemas/User'  
  
components:  
  schemas:  
    User:  
      type: object  
      properties:  
        id:  
          type: integer  
        name:  
          type: string
```

## Définition de Schémas

Le type de données d'un schéma est défini par le mot-clé `type` :

- `string` (cela comprend les dates et les fichiers)
- `number`
- `integer`
- `boolean`
- `array`
- `object`

Exemple:

```
User:  
  type: object
```

```
User:  
  oneOf:  
    - type: string  
    - type: object
```

<https://swagger.io/docs/specification/data-models/data-types>

Les tableaux sont définis de la sorte:

```
Users:  
  type: array  
  items:  
    type: string
```

```
Users:  
  type: array  
  items:  
    oneOf:  
      - type: string  
      - type: integer
```

Vous pouvez définir la longueur minimale et maximale d'un tableau à l'aide de `minItems` et `maxItems`.

```
Users:  
  type: array  
  items:  
    type: string  
  minItems: 1  
  maxItems: 10
```

Un objet est une collection de paires propriété/valeur:

```
User:  
  type: object  
  properties:  
    id:  
      type: integer  
    name:  
      type: string  
    contact_info:  
      type: object  
      properties:  
        email:  
          type: string  
          format: email  
  required:  
    - id
```

Par défaut, toutes les propriétés des objets sont **optionnelles**. Vous pouvez spécifier les propriétés requises dans la liste `required`.

Utilisez le mot-clé `enum` pour spécifier les valeurs possibles d'un type `string` :

```
#/ GET /users?sort=[asc|desc]
```

```
paths:  
  /users:  
    get:  
      parameters:  
        - in: query  
          name: sort  
          description: Sort order  
      schema:  
        type: string  
        enum:  
          - asc  
          - desc
```

💡 Il est fortement recommandé de placer les `enum`s dans la section globale `components.schemas`.

OpenAPI 3.0 fournit plusieurs mots-clés que vous pouvez utiliser pour combiner des schémas.

Vous pouvez utiliser ces mots-clés pour créer un schéma complexe ou pour valider une valeur en fonction de plusieurs critères.

- `oneOf` - valide la valeur par rapport à un seul des sous-schémas
- `allOf` - valide la valeur par rapport à tous les sous-schémas
- `anyOf` - valide la valeur par rapport à n'importe quel sous-schéma (un ou plusieurs)

## Exemple d'utilisation:

```
paths:  
  /pets:  
    patch:  
      requestBody:  
        content:  
          application/json:  
            schema:  
              oneOf:  
                - $ref: '#/components/schemas/Cat'  
                - $ref: '#/components/schemas/Dog'
```

```
components:  
  schemas:  
    Pet:  
      type: object  
      # ...  
    Dog:  
      allOf:  
        - $ref: '#/components/schemas/Pet'  
        - type: object  
          # ...  
    Cat:  
      allOf:  
        - $ref: '#/components/schemas/Pet'  
        - type: object  
          # ...
```

## Authentification

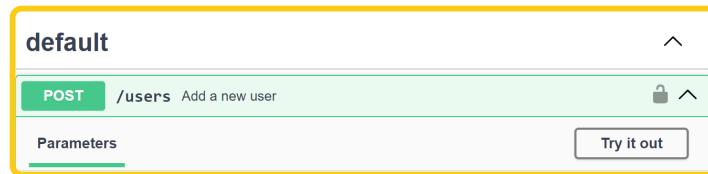
OpenAPI utilise le terme **chéma de sécurité** pour les schémas d'authentification et d'autorisation. Les types gérés sont:

- `http` - pour les authentifications Basic, Bearer et autres schémas d'authentification HTTP
- `apiKey` - pour les clés d'API et l'authentification par cookie
- `oauth2` - pour OAuth 2
- `openIdConnect` - pour OpenID Connect

Tous les schémas de sécurité utilisés par l'API doivent être définis dans la section `global components/securitySchemes`.

Exemple pour une authentification à l'aide d'un Bearer token:

```
paths:  
  /users:  
    post:  
      security:  
        - bearerAuth: []  
      # ...  
  
components:  
  securitySchemes:  
    bearerAuth: # nom arbitraire  
      type: http  
      scheme: bearer  
      bearerFormat: JWT
```



# Exercice

---

Définissez à l'aide d'un document **OAS** une API RESTfull permettant de:

- **1. Récupérer la liste d'une ressource de votre choix.**
  - La liste peut être paginée par `page` avec un nombre `per-pages` qui peut être défini (veuillez leur donner une **description**).
  - La liste sera typée et fournira un exemple.
- **2. Ajouter une ressource à cette liste**
  - Ce point d'accès doit être protégé par une authentification utilisant un Bearer token JWT
- **3. Modifier une ressource**
  - Assurez-vous que vous détailliez ce que vous devez utiliser comme paramètre dans le path
  - Ce point d'accès doit être protégé par une authentification utilisant un Bearer token JWT
- Assurez-vous de ne pas dupliquer les données des schémas.
- L'API ne prendra en charge que le type de média JSON.
- Documentez les différents codes de statut qui peuvent être retournés par vos endpoints (**404, 500, ...**).

# Ressources

---

Ce cours a été élaboré à partir des ressources suivantes :

- <https://openclassrooms.com/fr/courses/6573181-adoptez-les-api-rest-pour-vos-projets-web>
- <https://openclassrooms.com/fr/courses/6031886-debutez-avec-les-api-rest>
- [https://fr.wikipedia.org/wiki/Interface\\_de\\_programmation](https://fr.wikipedia.org/wiki/Interface_de_programmation)
- <https://www.turing.com/kb/7-examples-of-apis>
- <https://konghq.com/blog/learning-center/different-api-types-and-use-cases>
- <https://www.postman.com/state-of-api/>
- <https://en.wikipedia.org/wiki/REST>
- <https://swagger.io/docs/>
- <https://www.slideshare.net/slideshow/an-introduction-to-rest-api/76492672#2>
- <https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html>
- <https://www.knowi.com/blog/4-ways-of-rest-api-authentication-methods/>
- <https://blog.stoplight.io/api-keys-best-practices-to-authenticate-apis>
- <https://jwt.io/>
- <https://supertokens.com/blog/what-is-jwt>
- <https://medium.com/@techworldwithmilan/how-does-oauth-2-0-work-bea67a760aa5>
- <https://curl.se/docs/httpscripting.html>
- <https://everything.curl.dev/index.html>

