

## API WEB PRESENTATION

C'est dimanche. Vous vous réveillez tard, et même si vous savez que vous devriez vous lever, vous voulez juste rester enroulé bien au chaud dans votre couette et vous vous dites qu'en ce mois de février, partir profiter de la douceur du Portugal ne serait pas une mauvaise idée. Alors, vous ouvrez une application de comparateur de vols, ou un site comme « tripadvisor » ou on peut consulter les vols pour une destination de voyage après avoir choisi un hôtel un site comme « kayak » et voilà !

Vous trouvez plusieurs offres de différentes compagnies aériennes pour Lisbonne ! Bon, l'avion, ce n'est pas très écolo, alors vous tentez le train.

Vous ouvrez une application pour réserver un billet de train et pareil ! Vous trouvez plusieurs offres de trains pour Lisbonne en passant par l'Espagne. Et tout ça sans avoir à aller chercher l'information sur le site de chaque compagnie ferroviaire ou aérienne. Pas mal, non ?

Vous avez l'habitude de voir toutes ces données défiler sans arrêt sous vos yeux comme par magie, que ce soit sur les réseaux sociaux ou les applications de comparatifs en tout genre. De votre point de vue, c'est assez simple : les informations arrivent au fur et à mesure sans effort ; sauf qu'en coulisse, c'est autre chose ! En effet, un gros travail est effectué pour rendre cela possible et pour que cela fonctionne, il nous faut l'un des outils les plus importants du web : **une API**.

API par ci, API par là, OK OK, mais que signifie *API*

API est une abréviation et signifie *Application Programming Interface* (ou *interface de programmation d'application*, en français).

Pour faire simple : c'est un moyen de communication entre deux logiciels, que ce soit entre différents composants d'une application ou entre deux applications différentes.

OK OK OK... mais du coup, quel est le lien entre une API et le fait de réserver un hôtel à très bon prix sur le site booking.com par exemple, ou comparer des vols pour Lisbonne ?

### Découvrez le fonctionnement des API

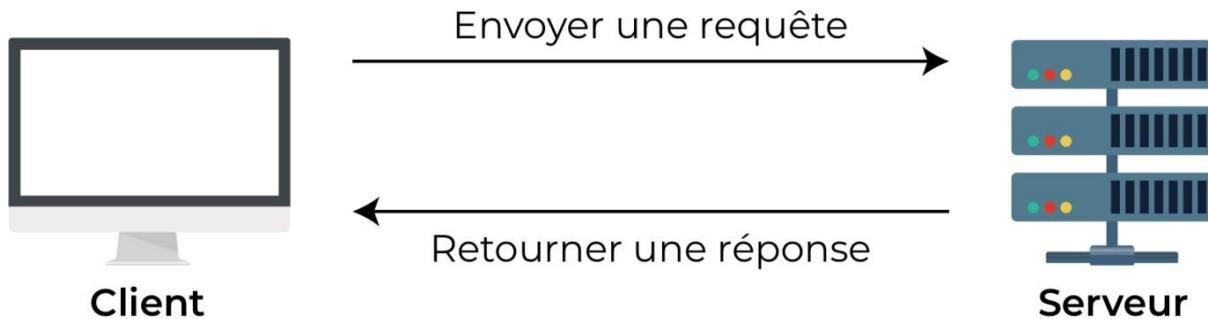
Eh bien, pour répondre à cette question, vous devez connaître un peu plus le fonctionnement d'une API. Mais avant, revoyons ensemble les bases d'une communication serveur et client.

Prenons l'exemple d'Air France, une compagnie aérienne française. Quelque part dans le monde, les serveurs d'Air France ont accès à toutes les données que vous voulez voir pour un trajet Paris-Lisbonne : les différents avions, les tarifs, les statuts des vols, etc. Pour que vous puissiez y avoir accès, votre navigateur (que l'on appelle le *client*) doit recevoir ces informations de quelqu'un.

Ce quelqu'un, c'est le **serveur**. L'application doit avoir une conversation avec le serveur.

# Conception & Développement Informatique

**DEVELOPPEUR WEB ET WEB MOBILE**  
TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



Une conversation typique entre client et serveur  
Cela ressemble à ça :

- Client : « Salut serveur, est-ce que je pourrais avoir un avion pour Lisbonne le 10 décembre 2023 ?»
- Serveur : « Voilà, tous les transports disponibles vers Lisbonne le 10 décembre ! »

Ou alors, si le serveur ne parvient pas à trouver les données, il pourrait répondre comme ceci :

- Serveur : « Désolé, en fait il n'y a pas de vols disponibles le 10 décembre. »

C'est ce qu'on appelle la **communication entre client et serveur** : le **client** formule une **requête** (ou une demande) pour obtenir une information et le **serveur** envoie une **réponse** contenant les données demandées si cela est possible.

Et du coup, l'API, elle se place où, dans ce schéma ? Et le rapport avec Air France ?

En web, un service web et une API sont tous les deux des moyens de communication. Un service web standard facilite seulement la communication entre deux machines via un réseau. Une API facilite l'interaction entre deux applications différentes afin qu'elles puissent communiquer entre elles : elle sert d'intermédiaire. Le client va demander à l'API une information, celle-ci va aller chercher cette information dans la base de données puis la renvoyer au client dans un second temps.

# Conception & Développement Informatique

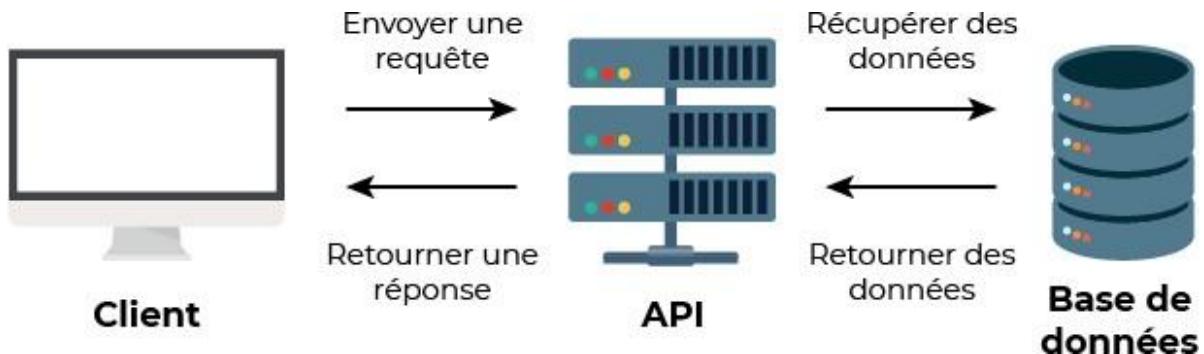
## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle



Une conversation entre le client, l'API et la base de données

Les API permettent la communication entre de nombreux **composants** différents de votre application, mais aussi entre des composants de votre application et d'autres **développeurs**. Elles agissent ainsi comme un intermédiaire qui transmet des messages à travers un système de requêtes et de réponses.

Elle permet aussi de distribuer une information unique avec des traitements à différentes applications clientes qui l'interrogent ( site web, application smartphone, logiciel client).

Reprendons notre exemple avec Air France. ☺

On crée une application de comparateur de vols que l'on va l'appeler VolScanner. Celle-ci ne peut pas accéder directement aux informations d'Air France ou de toute autre compagnie aérienne. En effet, l'application n'a pas accès à leurs base de données... Mais si Air France a une API à qui on peut demander des informations et qui partage certaines données de la base de données avec d'autres applications, alors VolScanner peut demander des informations à l'API d'Air France. L'API lui renvoie alors des données que VolScanner peut partager !

Ainsi, VolScanner peut comparer les prix entre les différentes compagnies qui ont mis en place un vol le 10 décembre pour Lisbonne.

Les API peuvent communiquer :

- d'un logiciel à un logiciel ;
- d'un client à un serveur ;
- ou d'un logiciel à des développeurs.
- 

Je suis certaine que vous avez déjà vu un exemple d'utilisation d'une API pour communiquer entre logiciels et développeurs : sur certains sites, vous pouvez utiliser votre compte Google ou Facebook pour vous identifier sans avoir à créer un identifiant et un mot de passe.

**C'est parce que Google et Facebook ont construit des API que d'autres développeurs peuvent utiliser dans leurs propres sites Internet pour s'occuper de l'inscription et de la connexion des utilisateurs à leur place.**

Et du coup, comment c'est possible techniquement ?

Les API créent des méthodes **standardisées** et **réutilisables** qui permettent aux développeurs d'accéder à des données spécifiques lors de la construction d'applications.

Prenons un exemple. Quand vous sortez manger, le menu du restaurant offre une grande quantité d'options déjà prédéterminées. Cela vous simplifie la tâche car vous savez ce que vous pouvez commander, et donc obtenir plus rapidement votre plat. Cela donne également une meilleure compréhension de ce que vous voulez pour le chef.

Au final, vous n'avez qu'à demander à la serveuse (API) un plat du menu qu'elle transmettra en cuisine, la cuisine prépare votre plat, le remet à la serveuse qui vous le ramène.

## Observez comment utiliser les API en tant que développeur

En tant que développeur, vous serez certainement amené à utiliser diverses API dans votre vie professionnelle ou pour vos projets personnels. Il existe deux types principaux : les API privées et les API publiques. Voyons ensemble de quoi il s'agit !

### Les API privées

Les **API privées** garantissent que les personnes en dehors de votre entreprise ou de votre application n'ont pas accès aux données disponibles de votre base de données. Par exemple, si les développeurs de l'AFPA voulaient construire une application **interne** pour que les RH puissent gérer et analyser des informations de recrutement, il y aurait de nombreuses données que les salariés voudraient voir, créer, et modifier. Pour que les utilisateurs puissent interagir avec les données, les développeurs de l'AFPA pourraient créer une API pour que les RH puissent accéder aux données de recrutement à travers leur application, sans pour autant donner ces accès aux utilisateurs de la plateforme comme vous et moi.

Une API peut être utilisée comme **un tampon ou une couche intermédiaire** entre la base de données et la personne qui veut accéder ou manipuler les données. Une requête directe et non contrôlée sur une base de données pourrait engendrer le chaos !

Et si quelqu'un supprimait accidentellement un cours ou modifiait quelque chose qu'il n'aurait pas dû toucher ? La base de données est la fondation de toutes les données au sein d'une application, donc il ne faut surtout pas qu'elle soit facilement accessible ou manipulable par n'importe qui. Question de sécurité !

Une API permet un niveau de sécurité supplémentaire pour mieux gérer l'accès et les modifications des données, en attribuant ce qu'on appelle des *droits* aux personnes qui en ont besoin. Ainsi, on s'assure de contrôler les utilisateurs qui auront ou non accès à la base de données.

Ainsi, moi seule peux modifier mes informations personnelles sur mon profil **AFPA**.

**Une API privée permet uniquement aux utilisateurs autorisés au sein de votre entreprise ou de votre application d'utiliser l'API qui peut accéder à la base de données.**

### **Les API publiques**

Contrairement aux API privées, les API que l'on appelle *publiques* sont utilisables par d'autres personnes, qu'elles soient sur votre application ou non. Elles permettent aux développeurs de récolter les données d'une autre application pour améliorer ou enrichir leurs propres projets sans autorisation stricte. Il existe de nombreuses manières d'utiliser des données provenant d'API tierces (ou externes), mais en voici quelques-unes :

1. Imaginons que vous vouliez construire un site web qui répertorie les conditions météo des stations de ski. Plutôt que de collecter vos propres données météorologiques, vous pouvez utiliser une [API de météo](#) et y trouver vos données ! ☁
2. Si vous êtes auteur-compositeur-interprète et que vous voulez créer un site web pour que vos fans puissent écouter votre musique, au lieu de construire votre propre lecteur de musique en streaming, vous pouvez utiliser [l'API de Spotify](#) et écouter votre musique directement sur votre site web ! 🎵
3. Vous voulez créer une page de fans pour votre série télé favorite, en réunissant tous les comptes Instagram des différents acteurs sur un seul site web – devinez quoi, il existe une [API Instagram](#) pour vous aider à le faire !

Il existe également certaines API à mi-chemin entre une API publique et privée. Cela peut se produire quand différentes **requêtes** de l'API sont possibles uniquement en fonction du niveau d'**accès** dont vous disposez. Nous y reviendrons plus tard lorsque nous traiterons de l'authentification.

Il existe des milliers d'API publiques que les développeurs peuvent utiliser de différentes façons pour améliorer leurs projets. Vous trouverez ici une liste de ces [API disponibles publiquement](#) que vous pouvez utiliser !

## **Identifiez les avantages d'une API REST**

Bien ! Maintenant que vous savez ce qu'est une API, parlons de ce qui compose une **API REST**. Nous utiliserons REST dans ce cours, car c'est le plus populaire.

C'est l'un des standards de création d'API les plus logiques, efficaces, et utilisés. Et, d'après le [rapport 2017 de l'état d'intégration des API de Cloud Elements](#) (en anglais), 83 % des API sont des API REST.

### **Comprenez tous les avantages de REST**

REST signifie **Representational State Transfer** (ou *transfert d'état de représentation*, en français), et constitue un ensemble de **normes**, ou de lignes directrices **architecturales** qui structurent la façon de communiquer les données entre votre application et le reste du monde, ou entre différents composants de votre application.

Nous utilisons l'adjectif RESTful pour décrire les API REST. Toutes les API REST sont un type d'API – mais toutes les API ne sont pas RESTful !

Les API RESTful se basent sur le protocole **HTTP** pour transférer les informations – le même protocole sur lequel la communication web est fondée ! Donc, lorsque vous voyez **http** au début d'une URL, comme <http://twitter.com> – votre navigateur utilise HTTP pour faire une requête de ce site web au serveur. REST fonctionne de la même façon !

Il y a **six** lignes directrices architecturales clés pour les API REST. Voyons ensemble de quoi il s'agit :

### #1 : Client-serveur séparation

L'une des normes de REST est la **séparation du client et du serveur**. Nous avons un peu abordé la question des clients et des serveurs dans le chapitre précédent, il est temps d'approfondir un peu le sujet !

Un **client** est celui qui va utiliser l'API. Cela peut être une application, un navigateur ou un logiciel. Par exemple : en tant que développeur, vous utiliserez peut-être l'API de Twitter. Comme je l'ai dit précédemment, un client peut aussi être un logiciel ou un navigateur, qu'il s'agisse de Chrome, Safari ou Firefox. Quand un navigateur se rend sur [twitter.com](http://twitter.com), il formule une requête à l'API de Twitter et utilise les données de l'API afin que vous puissiez accéder aux derniers tweets.

Un **serveur** est un ordinateur distant capable de récupérer des données depuis la base de données, de les manipuler si besoin et de les renvoyer à l'API, comme ce gros ordinateur au milieu :

# Conception & Développement Informatique

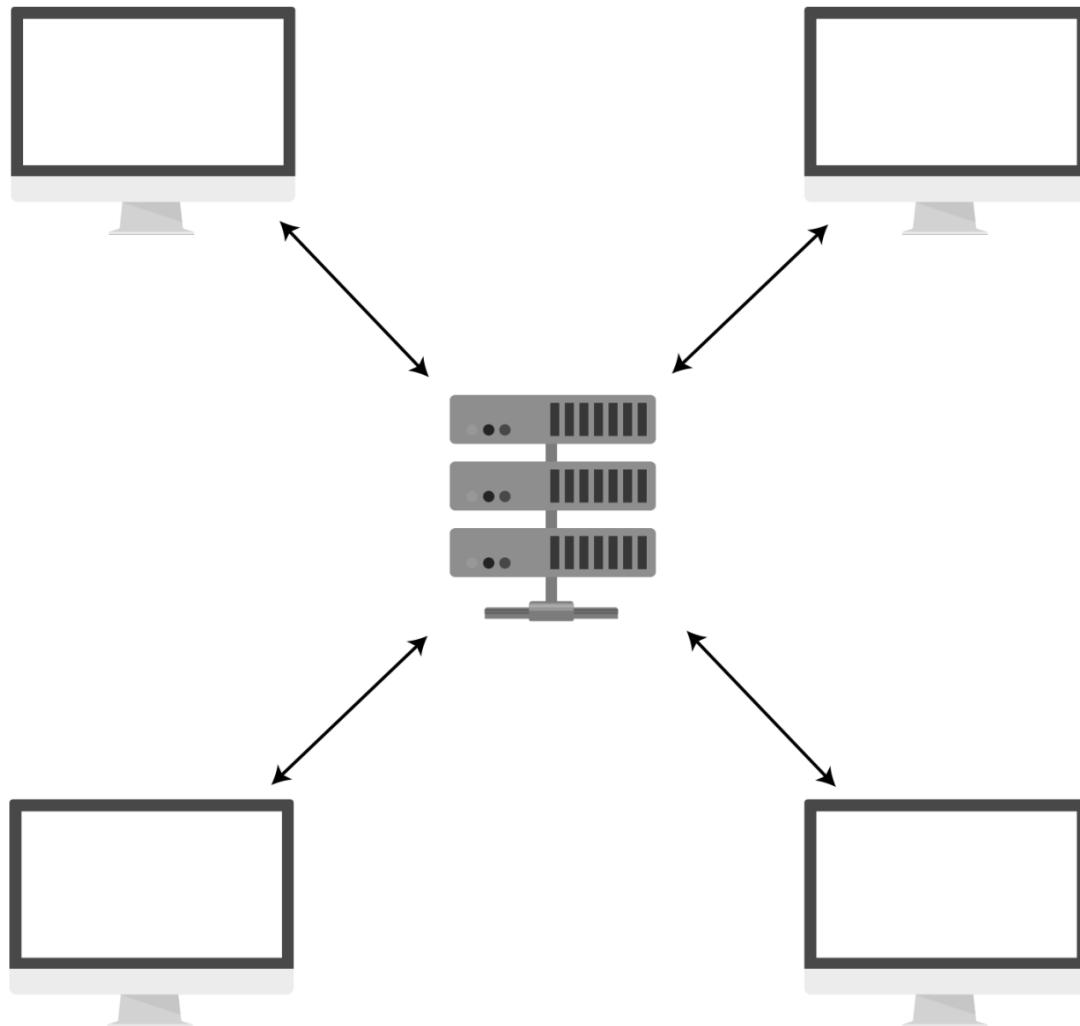
## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle



Une relation client-serveur typique

De façon générale, il existe une séparation entre le client et le serveur. Cette séparation permet au client de s'occuper uniquement de la récupération et de l'affichage de l'information et permet au serveur de se concentrer sur le stockage et la manipulation des données. Chacun son rôle !

Les API REST offrent un **moyen de communication standardisé entre le client et les données**. En gros, peu importe comment le serveur est construit ou comment le client est codé, du moment qu'ils structurent tous les deux leur communication selon les lignes directrices architecturales REST, en utilisant le protocole HTTP, ils pourront communiquer entre eux !

C'est particulièrement utile lorsque de grandes équipes de développeurs travaillent sur une même application. Vous pouvez avoir une équipe qui travaille indépendamment sur le backend tandis que l'autre travaille sur le frontend. Comme l'API REST communique entre les deux, cela permet aux développeurs de scaler plus facilement les applications et aux équipes de travailler de manière plus efficace.

### #2 : Stateless

L'un des autres aspects uniques des API REST est qu'elles sont **stateless** – *sans état*, en français – ce qui signifie que le serveur ne sauvegarde aucune des requêtes ou réponses précédentes.

Mais le rôle du serveur est de stocker et manipuler les données. Comment est-ce que cela pourrait fonctionner si on ne garde pas une trace des requêtes, alors ?

Pour revenir à notre métaphore de l'API en tant que serveuse, imaginons que vous demandiez des frites à votre serveuse. Elle se rend à la cuisine, récupère vos frites, et revient avec votre commande. Parfait !

Houla... mais attendez ! Vous venez de vous souvenir que vous voulez également du ketchup avec vos frites. Vous demandez donc à votre serveuse : « Excusez-moi, je pourrais avoir du ketchup avec ? » « Avec quoi ? » Une serveuse **stateless** n'aurait aucune idée de ce dont vous parlez... car elle ne se souviendrait pas que vous venez de commander des frites ! Elle se charge seulement de transférer les commandes de la cuisine au client.

OK, mais alors concrètement, qu'est-ce que cela signifie pour les API REST ?

Étant donné que chaque message est isolé et indépendant du reste, il vous faudra vous assurer d'envoyer avec la requête que vous formulez toutes les **données nécessaires** pour être sûr d'avoir la réponse la plus précise possible. Cela nous donnerait quelque chose comme : « Est-ce que je pourrais avoir du ketchup sur les **frites** que **j'ai** commandées à **ma** table ? » Avec toutes ces informations, votre serveuse pourra identifier à quelles frites il faut ajouter du ketchup !

**Le fait d'être stateless** rend chaque requête et chaque réponse très **déterminée** et **compréhensible**. Donc, si vous êtes développeur et que vous voyez la requête API de quelqu'un d'autre dans un code déjà existant, vous serez capable de comprendre l'objet de la requête sans contexte !

### #3 : Cacheable (ou sauvegardable, en français)

La réponse doit contenir l'information sur la capacité ou non du client de mettre les données **en cache**, ou de les sauvegarder. Si les données **peuvent être mises en cache**, la réponse doit être accompagnée d'un numéro de version. Ainsi, si votre utilisateur formule deux fois la même requête (c'est-à-dire s'il veut revoir une page) et que les informations n'ont pas changé, alors votre serveur n'a pas besoin de rechercher les informations une deuxième fois. À la place, le client peut simplement mettre en cache les données la première fois, puis charger à nouveau les mêmes données la seconde fois.

Une mise en cache efficace peut réduire le nombre de fois où un client et un serveur doivent interagir, ce qui peut aider à accélérer le temps de chargement pour l'utilisateur !

Vous avez peut-être entendu le terme **cache** en référence à, par exemple, « Rafraîchissez le cache de votre navigateur ». Un cache est un moyen de **sauvegarder** des données pour

pouvoir répondre plus facilement aux prochaines requêtes qui seront **identiques**. Quand vous allez sur de nombreux sites web depuis votre navigateur, celui-ci peut sauvegarder ces requêtes pour pouvoir compléter lui-même le site que vous voulez atteindre ou charger la page plus rapidement la prochaine fois que vous vous y rendez. Pratique !.

### #4 : Uniforme Interface (*interface uniforme*)

Lors de la création d'une API REST, les développeurs acceptent d'utiliser les mêmes normes. Ainsi, chaque API a une **interface uniforme**. L'interface constitue un contrat entre le client et le service, que partagent toutes les API REST. C'est utile, car lorsque les développeurs utilisent des API, cela leur permet d'être sûrs qu'ils se comprennent entre eux.

Une API REST d'une application peut communiquer *de la même façon* avec une autre application entièrement différente.

### #5 : Layered system (*système de couches*)

Chaque composant qui utilise REST n'a pas accès aux composants au-delà du composant précis avec lequel il interagit.

Que... quoi ? C'est-à-dire ?

Cela signifie qu'un client qui se connecte à un composant intermédiaire n'a aucune idée de ce avec quoi ce composant interagit ensuite. Par exemple, si vous faites une requête à l'API Facebook pour récupérer les derniers posts : vous n'avez aucune idée des composants avec lesquels l'API Facebook communique..

Cela encourage les développeurs à créer des composants indépendants, facilitant le remplacement ou la mise à jour de chacun d'entre eux.

### #6 : Code on demand (*code à la demande*)

Le code à la demande signifie que le serveur peut étendre sa fonctionnalité en envoyant le code au client pour téléchargement. C'est facultatif, car tous les clients ne seront pas capables de télécharger et d'exécuter le même code – donc ce n'est pas utilisé habituellement, mais au moins, vous savez que ça existe !

## Découvrez les alternatives aux API REST

REST n'est qu'**un** type d'API. Il existe des alternatives qui vous seront également utiles à connaître, notamment les API **SOAP**.

SOAP est l'acronyme de *Simple Object Access Protocol*, ou *protocole simple d'accès aux objets*, en français. Contrairement à REST, il est considéré comme un protocole, et non comme un style d'architecture.

Les API SOAP étaient les API les plus courantes avant l'arrivée de REST. REST utilise le protocole HTTP pour communiquer, SOAP d'un autre côté peut utiliser de multiples moyens de communication. Le souci, c'est la complexité qui en ressort, car les développeurs doivent se coordonner pour s'assurer qu'ils communiquent de la même manière afin d'éviter les

problèmes. De plus, le SOAP peut demander plus de bande passante, ce qui entraîne des temps de chargement beaucoup plus longs. REST a été créé pour résoudre certains de ces problèmes grâce à sa nature plus légère et plus flexible.

De nos jours, le SOAP est plus fréquemment utilisé dans les applications de grandes entreprises, puisqu'on peut y ajouter des couches de sécurité, de confidentialité des données, et d'intégrité supplémentaires. REST peut être tout aussi sécurisé, mais a besoin d'être implémenté, c'est-à-dire d'être développé au lieu d'être juste intégré comme avec le SOAP.

Maintenant que vous savez que les API REST servent d'intermédiaires et aident les développeurs à manipuler des données, regardons de plus près à quoi ressemblent réellement ces données.

## Appréhendez les données REST via l'utilisation des ressources

Les données REST sont représentées dans ce qu'on appelle des *ressources*.

Une ressource peut être tout type d'objet **nominal** (on lui attribue un nom) que vous pouvez utiliser pour représenter les données dans votre application. Vous savez, une personne, un lieu, ou autre chose ! Pour faire simple, voyez les ressources comme des boîtes dans lesquelles vous rangerez des objets par catégorie et sur lesquelles vous collez une étiquette pour savoir quoi mettre dedans.

Vous trouvez que c'est abstrait ? C'est le but, afin que vous puissiez représenter n'importe quel élément de donnée sous la forme que vous souhaitez.

Chaque **ressource** comporte des informations supplémentaires sur les données contenues. Si on prend l'exemple d'une application qui liste les héros Marvel, une des ressources pourrait être *Superhero* et on pourrait avoir par exemple un nom, une description, etc., comme information supplémentaire.

Les ressources sont regroupées dans un groupe que l'on appelle une **collection**. On s'y réfère avec la forme au **pluriel** du nom de la ressource. Par exemple une ressource superhero donnerait superheroes.

Par convention, tous les champs d'une ressource et le nom d'une collection sont en anglais. Ils sont traduits ici en français pour une meilleure compréhension du cours, mais privilégiez toujours l'anglais !

Imaginons que vous créez une API pour qu'une boutique de skateboards ait un service de livraison en ligne. Ce que vous voulez, c'est que d'un côté vos clients puissent acheter des

# Conception & Développement Informatique

**DEVELOPPEUR WEB ET WEB MOBILE**  
TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT

skateboards sur le site web, et de l'autre que vos salariés puissent ajouter des produits et mettre l'inventaire à jour.

Procédons étape par étape et déterminons ensemble les ressources, leurs informations supplémentaires et les collections.

Pour une **boutique de skateboards**, vos ressources pourraient être :

- Client ;
- Staff ;
- Basket (panier) ;
- Skateboard ;
- Inventory (inventaire).

Une ressource *Skateboard* pourrait comporter comme informations supplémentaires : nom, marque, id, prix.

Vos collections seraient donc :

Ressource	Collection
skateboard	skateboards
client	clients

Cette liste est un exemple et ne contient pas tous les exemples cités précédemment.

Bien ! Nous avons nos collections ainsi que les ressources correspondantes et leurs informations supplémentaires. Continuons avec notre exemple de boutique de skateboards et suivons ensemble le parcours de la requête d'un client via notre API.

Quand un client achète un skateboard en utilisant votre application web, cela donne :

- votre API envoie la requête du navigateur (le client) aux serveurs de l'application pour l'achat d'un skateboard ;
- la requête met à jour l'**inventaire** pour qu'il y ait un skateboard de moins ;
- la requête met à jour l'historique de commandes du **client** pour ajouter le skateboard à son historique d'achats.

Super ! Vous savez maintenant comment et sous quelle forme stocker les données que vous voulez utiliser dans votre API via des ressources et des collections. Mais du coup, comment pouvez-vous y accéder ? Comment savoir où les récupérer, ces données ?

## **URI et Endpoints**

Le **path (ou chemin)** que vous donnez à votre API lui permet de savoir exactement où se trouvent les données que vous voulez récupérer. Vous pouvez imaginer cela comme le fait de parcourir vos propres fichiers sur votre ordinateur. Vous devez aller de dossier en dossier pour trouver vos données, et chaque photo ou document que vous sauvegardez a son propre path,

ou *chemin de fichier*, en français. Par exemple, votre photo de série préférée pourrait se trouver au bout de ce path :

```
MyComputer/Images/Series/gameofthrones.jpg
```

Les API REST stockent également les données de façon similaire, et un URI constitue le chemin pour y arriver.

Si une ressource est l'objet qui stocke vos données, pour les récupérer vous allez avoir besoin d'un identifiant de ressource uniforme, ou **URI** pour *Uniform Resource Identifier*. L'URI est le moyen d'identifier votre ressource, comme une étiquette.

Imaginons que vous créez une API pour un site web qui présenterait toutes les informations de Game of Thrones, que ce soit sur le livre ou la série. L'URI qui listerait tous les personnages pourrait être la suivante :

```
/characters
```

Si vous voulez voir les informations sur *un seul* personnage, qui porte l'ID 123, votre URI serait le suivant :

```
/characters/123
```

Tout comme les paths pour les fichiers, les URI peuvent avoir des ressources **imbriquées**. Si vous voulez obtenir uniquement *le nom* du personnage qui vous intéresse, votre URI pourrait ressembler à ceci :

```
/characters/123/description
```

Wouhou ! Voilà du progrès ! Le souci, c'est que sans l'adresse réelle du site web, l'API ne saura pas du tout où chercher l'URI pour commencer ! C'est là que les **endpoints** (ou *points de terminaison*, en français) interviennent !

Un endpoint est une URL/URI qui fait partie d'une API. Si un URI est comme un chemin de fichier, alors un endpoint est comme l'adresse complète du fichier. Il vous suffit d'ajouter votre **nom de domaine** au début de votre URI, et vous avez un endpoint ! Par exemple, si le nom de domaine de notre app est *gameofthrones-informations.com*, nous aurons :

```
https://gameofthrones-informations.com/characters
```

Houla, attends deux secondes, c'est quoi la différence entre URI et URL ?

Toutes les URL sont des URI, mais toutes les URI ne sont pas des URL. L'URI permet d'identifier une ressource tandis que l'URL permet de la localiser.

On confond souvent les deux. On va tout simplifier avec un exemple ! Reprenons notre site de Game of Thrones. Si le personnage de Jon Snow a pour ID 890, alors l'URI serait /characters/890.

L'URL serait <https://gameofthrones-informations.com/characters/890>

**L'URL de la requête** est l'endpoint complet que vous utilisez pour votre requête. Il associe le nom de domaine + le path de votre ressource. À présent, vous savez comment accéder aux données que vous souhaitez !

## Distinguez XML et JSON

Une fois que vous avez le bon endpoint sur lequel faire votre requête, il est temps pour vous d'obtenir vos données ! C'est là que vous obtenez les informations sur les ressources que vous avez créées.

Le terme *données* est un terme général qui décrit toute information envoyée ou reçue, tandis que le terme *ressource* décrit plus précisément les **éléments** qui sont contenus dans cette information.

Les données des API REST peuvent utiliser deux langages : XML et JSON. Si une API renvoie un set de données en XML ou en JSON, le contenu restera le même, mais la forme change. Le format de données est différent.

### Le XML

En **XML**, chaque élément de donnée a une balise ouvrante et une balise fermante qui peut également avoir des balises imbriquées :

```
<series>

    <serie>    <titre>Game Of Thrones</titre>

        <realisateur>Random</realisateur>

    </serie>

    <serie>    <titre>Peaky Blinders</titre>

        <realisateur>Random</realisateur>  </serie>

    </series>

<series>

    <serie>

        <titre>Game Of Thrones</titre>
```

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

```
<realisateur>Random</realisateur>

</serie>

<serie>

<titre>Peaky Blinders</titre>

<realisateur>Random</realisateur>

</serie>

</series>
```

Vous pouvez voir une balise ouvrante pour la **collection** en haut – series – entre crochets <>. La balise fermante à la fin est identique, sauf qu'on y ajoute une barre oblique / au début : </series> , qui indique que c'est une balise fermante. Chaque **ressource** listée a la balise ouvrante <serie> et la balise fermante </serie> . Au sein de chaque ressource se trouvent davantage d'informations, comme "titre" et "réalisateur".

## Le JSON

Le **JSON** stocke les données sous un format de clé-valeur avec comme clé le type de données, suivi de deux points : , suivi de la valeur de la donnée. Les données JSON sont entourées d'accolades { }, et chaque paire clé-valeur est envoyée comme chaîne de caractères avec des guillemets autour "".

Ce qui nous donne ceci :

```
{"titre" :"Game of Thrones"}
```

Les tableaux, ou listes, en JSON sont entourées de crochets []. L'exemple ci-dessous montre comment une liste complète peut être considérée comme la valeur de la clé "series". Les mêmes données en XML ci-dessus seraient représentées ainsi en JSON :

```
{ "series": [
    {
        "titre": "Game Of Thrones",
        "realisateur": "Alan Taylor" },
        {   "titre": "Peaky Blinders",
        "realisateur": "Otto Bathurst" }
    }
]}
```

Le JSON est généralement considéré comme :

1. Plus facile à analyser avec du code.
2. Plus court.
3. Plus rapide à lire et à écrire pour les machines.
4. Très "léger" et efficace grâce à sa structure en arborescence et sa syntaxe simple.

Voici quelques exemples d'API réelles qui renvoient du JSON et du XML :

- Penguin Random House : [XML](#).
- Potter API : [JSON](#).
- Ghibli API: [JSON](#)

Comme vous pouvez le constater, le JSON est le langage de données le plus utilisé.

## Identifiez les avantages de Postman

Vous savez déjà qu'une API REST implique l'envoi de **requêtes** du client à l'API, qui passe la requête au serveur, l'API récupère la **réponse** et la renvoie enfin au client. Dans ce chapitre, nous allons voir comment formuler ces requêtes grâce à Postman.

Cette interface graphique est utilisée par de nombreux développeurs. Elle facilite la construction de nos requêtes. C'est donc l'outil idéal pour tester des API sans devoir utiliser de code.

C'est également celui que nous allons utiliser dans ce cours !

OK, mais pourquoi utiliser Postman en particulier ?

Parce que cette interface offre beaucoup d'avantages :

- Vous pouvez l'utiliser quel que soit le langage avec lequel vous programmez.
- Son interface utilisateur est simple : vous effectuez vos requêtes facilement.
- Vous n'avez pas besoin de savoir coder, ou d'utiliser une application.

### Téléchargez Postman

L'heure est venue de vous lancer : [téléchargez Postman](#).

Choisissez votre système d'exploitation. Postman est disponible pour Linux, Windows et OS X. Les instructions sont en anglais.

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows the Postman application interface. At the top, there are navigation links: Product, Pricing, Enterprise, Resources and Support, and Explore. On the right, there are buttons for Sign In and Sign Up for Free. The main area displays a collection named "Twitter API v2 / Tweet Lookups". A specific request titled "Single Tweet" is selected, showing a GET method to the URL <https://api.twitter.com/2/tweets/:id>. The "Params" tab is active, showing fields like "id" with the value "140321029621628420". The "Body" tab shows a JSON payload: {"data": {"id": "140321029621628420", "text": "Dinner with Paul George went down after a collision with Paul George toward the end of Game 2. <https://t.co/PV9IA0Q02W>"}}. The "Test Results" tab indicates a successful response with status 200 OK, 466 ms, and 754 B. Below the interface, there is a section titled "What is Postman?" with a brief description and a "Download the desktop app" button with icons for Windows, Mac, and Linux.

## La page d'accueil de Postman

Une fois que vous l'aurez fait, vous allez être redirigé sur une page pour télécharger le logiciel. Cliquez sur le bouton **Download the App**.

## The Postman app

The ever-improving Postman app (a new release every two weeks) gives you a full-featured Postman experience.

Download the App

By downloading and using Postman, I agree to the [Privacy Policy](#) and [Terms](#).

Version 8.11.1 · [Release Notes](#) · [Product Roadmap](#)

Not your OS? Download for Windows ([x32 / x64](#)) or Linux ([x64](#))

Parfait ! Et maintenant, passons à la formulation des requêtes.

## Formulez une requête sur Postman

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



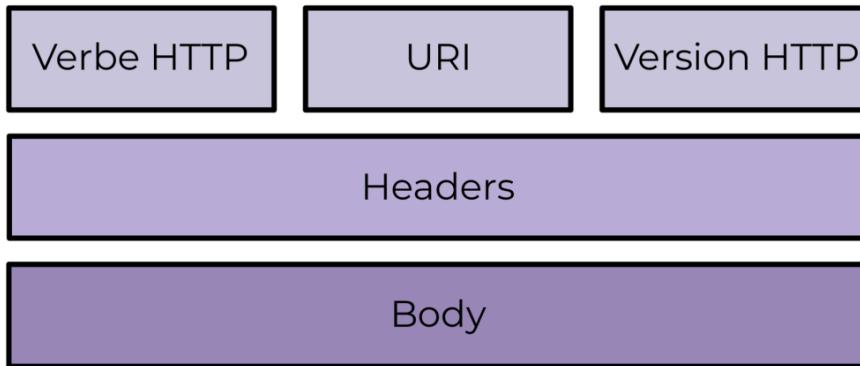
CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

### ***La structure d'une requête***

Chaque requête a une structure spécifique qui a cette forme :

Verbe HTTP + URI + Version HTTP + Headers + Body (facultatif)



Une structure de requête typique

L'aspect peut varier en fonction du logiciel ou du langage utilisé.

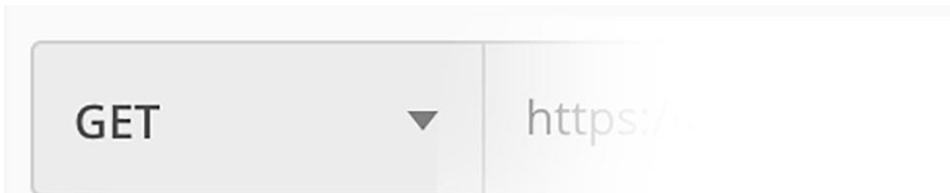
### ***Visualisez une requête sur Postman***

Maintenant que vous avez lancé le logiciel, regardons ensemble à quoi ressemble une requête sur Postman.

Vous devriez avoir cette vue :

Champ utilisé pour réaliser une requête dans Postman

Pas de panique, détaillons ensemble chaque zone une à une en suivant la structure d'une requête :



Exemple de verbe HTTP avec GET

Le verbe

Commençons avec le verbe. Les **verbes HTTP** correspondent à différents types d'actions que vous pouvez accomplir avec votre requête. Ceux que vous rencontrerez le plus couramment sont **GET** (obtenir), **PUT** (mettre), **POST** (publier), et **DELETE** (supprimer). Ne vous y attardez pas trop pour le moment, nous les étudierons tous plus en détail plus tard !



L'URL d'une requête complète comprend le nom de domaine : api.github.com, et l'URI (le chemin de la ressource) : /users/facebook

L'URI

Passons à l'**URI**. Un URI est le moyen d'identifier les ressources. Par exemple, si vous voulez voir tous les utilisateurs sur votre site web, le path serait le suivant :

/users

OK, mais imaginons que vous vouliez obtenir les informations d'un utilisateur spécifique. Dans ce cas-là, il vous faudrait préciser son ID. On obtiendrait quelque chose comme ceci :

users/:user\_id

Pourquoi on utilisait un nombre avant, et tout d'un coup tu nous mets :user\_id ?

On utilise :user\_id pour matérialiser l'ID de l'utilisateur, c'est ce qu'on appelle un placeholder.

En pratique, avec un ID réel, le path ressemblerait plutôt à ça : users/145

The screenshot shows the Postman interface with the 'Headers' tab selected. There are two header entries listed:

KEY	VALUE
Content-Type	application/json
Authorization	token [key]

### Headers dans une requête

Un **header** (ou *en-tête*) vous permet de faire passer des informations supplémentaires sur le message. Par exemple :

- De quel langage s'agit-il ?
- À quelle date l'envoyez-vous ?
- Quel logiciel la personne utilise-t-elle ?
- Quelle est votre clé d'authentification ?

Les headers sont représentés par une paire **clé et valeur**, et il existe de nombreux types d'options différents pour eux. Par exemple :

Date : Mardi 19 Janvier 2019 18:15:41 GMT

Utilisateur-Agent : Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_5)

Vous pouvez consulter la [liste complète](#) des différentes options pour les headers.

The screenshot shows the Postman interface with the 'Body' tab selected. The body content is set to 'raw' and contains the following JSON:

```
1: {"message": "body in JSON here"}
```

Le body (ou corps de message, en français)

Le body

Pour finir, parlons du **body** ! Pour formuler une requête, il n'est utilisé qu'avec **PUT** (mise à jour) ou **POST** (création). Il contient les données réelles de la ressource que vous essayez de créer ou de mettre à jour. Les données sont envoyées sous format JSON.

Petit rappel : le JSON est largement utilisé ; toutefois il se peut que certaines API n'acceptent que le XML. Vous trouverez cette information dans la documentation de l'API que vous utiliserez.

Prenons un exemple ! Vous voulez ajouter ou mettre à jour les informations d'un utilisateur ; vous ajouterez donc les détails de l'utilisateur (prénom, nom , adresse...) dans le body, en JSON.

Notez que le body est facultatif dans ces deux cas. Cela signifie qu'il est tout à fait possible d'envoyer un body vide en fonction des actions de l'API visée.

### Obtenez une réponse avec Postman

Structure des requêtes  ! Passons aux réponses.

Le format du message de réponse est très similaire à celui de la requête :

Version HTTP + Code de réponse HTTP + Headers + Body

Version HTTP

Code HTTP

Headers

Body

Une structure de réponse typique

Dans Postman, vous verrez un message de réponse comme celui-ci :

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows a Postman interface with a successful response. The URL is https://api.github.com/users/kadaaran. The Headers tab is selected, showing a single header 'User-Agent'. The Body tab shows a JSON response object with 20 properties, including login, id, node\_id, avatar\_url, gravatar\_id, url, html\_url, followers\_url, following\_url, gists\_url, starred\_url, subscriptions\_url, organizations\_url, repos\_url, events\_url, received\_events\_url, type, site\_admin, and name. The response status is 200 OK.

```
1 {
2   "login": "Kadaaran",
3   "id": 11220823,
4   "node_id": "MDQ6VXNlcjExMjIwODIz",
5   "avatar_url": "https://avatars.githubusercontent.com/u/11220823?v=4",
6   "gravatar_id": "",
7   "url": "https://api.github.com/users/Kadaaran",
8   "html_url": "https://github.com/Kadaaran",
9   "followers_url": "https://api.github.com/users/Kadaaran/followers",
10  "following_url": "https://api.github.com/users/Kadaaran/following{/other_user}",
11  "gists_url": "https://api.github.com/users/Kadaaran/gists{/gist_id}",
12  "starred_url": "https://api.github.com/users/Kadaaran/starred{/owner}{/repo}",
13  "subscriptions_url": "https://api.github.com/users/Kadaaran/subscriptions",
14  "organizations_url": "https://api.github.com/users/Kadaaran/orgs",
15  "repos_url": "https://api.github.com/users/Kadaaran/repos",
16  "events_url": "https://api.github.com/users/Kadaaran/events{/privacy}",
17  "received_events_url": "https://api.github.com/users/Kadaaran/received_events",
18  "type": "User",
19  "site_admin": false,
20  "name": "Kassandra Pedro",
```

### Un message de réponse typique

Attends deux secondes... Le body ? Mais il n'est pas censé être utilisé seulement pour faire une requête avec POST et PUT ?

Pour formuler une requête, oui ! Mais pour les réponses, le **body** contient l'information que vous avez demandée, et que l'API vous renvoie. Celle-ci est matérialisée au sein du body sous la forme d'un JSON ou en XML. L'image ci-dessus montre une réponse d'une requête réussie, faite à l'API GitHub.

Et si elle échoue ? Que se passe-t-il ? On obtient une information différente ? Exactement ! Si la requête échoue, le **body** peut contenir un message d'erreur :

The screenshot shows a Postman interface with a failed response. The URL is https://api.github.com/users/notfound. The status is 404 Not Found. The Body tab shows a JSON response with a single key 'message' containing the value 'Not Found', and a 'documentation\_url' key pointing to a GitHub developer documentation page.

```
1 {
2   "message": "Not Found",
3   "documentation_url": "https://developer.github.com/v3/users/#get-a-single-user"
4 }
```

### Une requête qui échoue

Mais le message ne suffit pas, et entre nous il peut arriver que des API n'envoient pas de messages du tout – que la requête soit un succès ou non. Cela peut arriver, même si c'est rare ! Dans ce cas, votre meilleur allié sera le **code de réponse HTTP** !

## Analysez le code de réponse HTTP

Le code de réponse HTTP aide le développeur et/ou le client à comprendre le **statut** de la réponse. Jetons un œil sur les exemples obtenus avec Postman :

Status: 200 OK

Un code de statut HTTP pour une requête réussie

Status: 404 Not Found

Un code de statut HTTP pour un échec

Lorsqu'un client vous envoie une requête comme « Salut, pourriez-vous m'envoyer tous les tweets de cet utilisateur ? », vous pouvez vous représenter le code de réponse comme un feu de signalisation  qui vous dit par exemple :

**VERT ✓** : « Cette requête a été traitée avec succès. »

ou

**ROUGE ⚡** : « Nous n'avons rien trouvé pour cette requête ! »

Il existe de nombreux codes de réponse différents – vous connaissez probablement le **404 not found** (ou *introuvable*, en français). Vous l'avez peut-être vu sur quelques sites web. Par exemple, si vous essayez d'aller sur une page qui n'existe pas sur GitHub, vous verrez quelque chose comme ça :



### Page de réponse 404 GitHub

Un autre code de réponse important à connaître est le **200 OK** – qui signifie que votre requête a réussi, et que votre réponse est prête ! En général, les règles de base pour les codes de réponse HTTP sont les suivantes :

- 100+ → Information
- 200+ → Succès
- 300+ → Redirection
- 400+ → Erreur client
- 500+ → Erreur serveur

Les codes HTTP sont très codifiés, et chaque nombre correspond à une réponse particulière. De ce fait, toutes les API renverront une 404 si la ressource est introuvable, et cela permet aux développeurs de comprendre de quel type de réponse il s'agit. C'est une sorte de nomenclature générale respectée par tous.

Si vous avez un doute sur un code HTTP, n'hésitez pas à consulter [cette documentation](#) qui contient de plus amples informations et détails !

Dans le chapitre précédent, nous avons parlé des verbes HTTP et de la façon dont ils permettent de réaliser des **actions** spécifiques lors de la formulation d'une requête API. Rappelez-vous, j'avais mentionné GET, POST, PUT et DELETE. Eh bien, il est temps d'en apprendre un peu plus, et surtout de les utiliser !

### Découvrez le CRUD

Le CRUD est la liste des actions de base que vous pouvez effectuer sur une ressource. C'est un acronyme qui signifie *Create* (créer), *Read* (lire), *Update* (mettre à jour), et *Delete* (supprimer). Bien que le CRUD ne constitue pas vraiment un mécanisme technique en soi, chaque **action CRUD** est associée à un **verbe HTTP**. Voici la cartographie :

Action CRUD	Verbe HTTP associé
Create (Créer)	POST (Publier)
Read (Lire)	GET (Obtenir)
Update (Mettre à jour)	PUT (Mettre)
Delete (Supprimer)	DELETE (Supprimer)

## Obtenez des résultats avec votre première requête GET

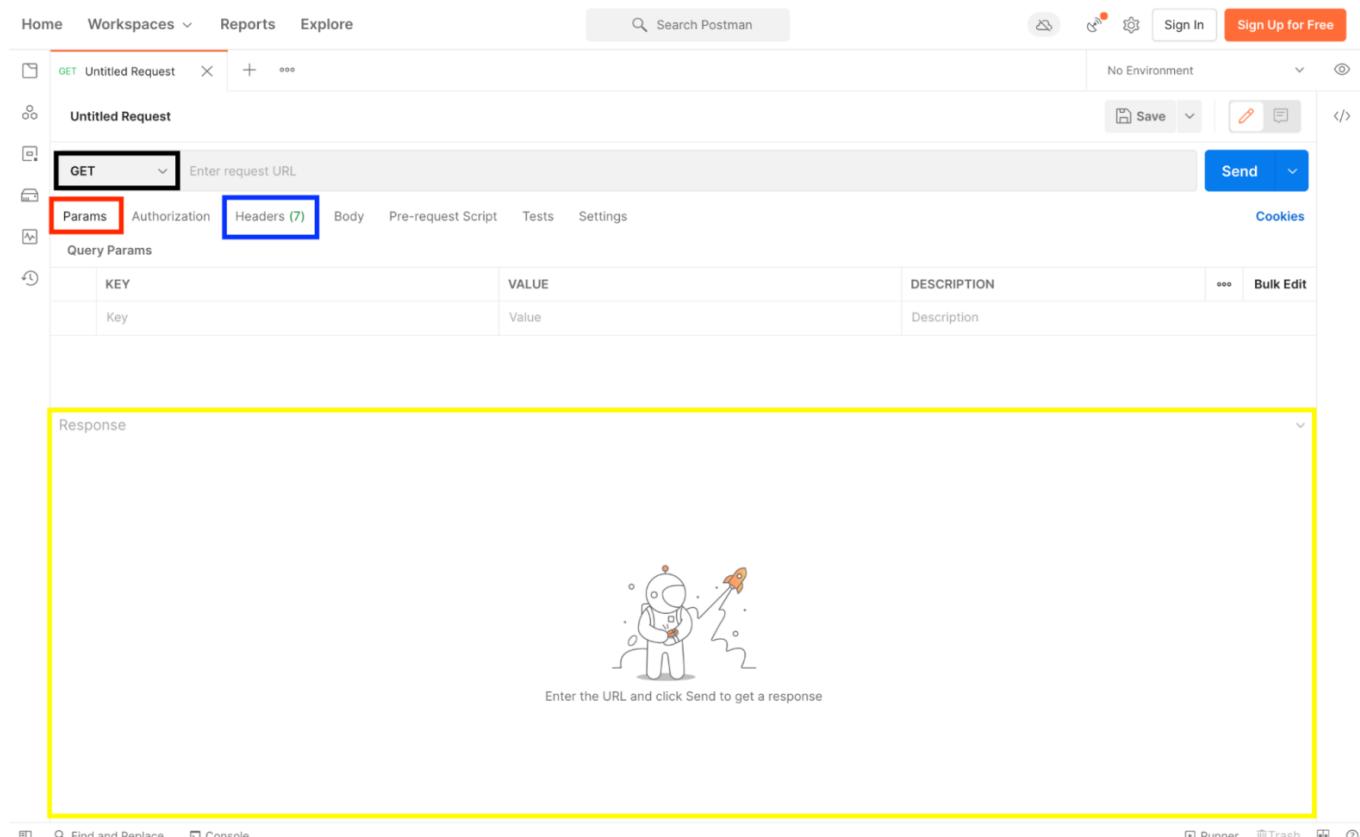
Maintenant que nous avons vu tout le contexte qui se cache derrière une API, il est temps de pratiquer ! Utilisons une API pour obtenir des données. Nous utiliserons le verbe **HTTP GET** et l'[API GitHub](#) (de [GitHub](#)) pour obtenir des données sur un utilisateur GitHub spécifique.

GitHub est une plateforme qu'utilisent les développeurs pour stocker leur code et travailler seul ou en équipe. Cela permet de faciliter la collaboration entre développeurs d'un même projet. En tant que développeur, vous l'utiliserez beaucoup ! Si vous êtes curieux de connaître son fonctionnement, allez voir le cours [Gérez du code avec Git et GitHub](#).

Pour faire une requête sur l'API, utilisons le logiciel [Postman](#) que vous avez téléchargé précédemment.

Commencez par ouvrir le programme.

Nous l'avions survolé lors du dernier chapitre, mais il est toujours bon de revoir ce que nous avons appris. Détaillons un peu ce que nous voyons, de haut en bas !



### Interface utilisateur Postman

- La première ligne (encadrée en noir) vous permet de sélectionner votre type de requête dans le menu déroulant (dans notre cas, ce sera GET).
- À côté, vous pouvez remplir la case avec l'URL complète de votre requête.

- Il y a un petit bouton (encadré en rouge) nommé *Params*. Si vous cliquez dessus, vous aurez un emplacement pour définir les valeurs clés de vos paramètres.
- À droite (encadré en bleu), vous pouvez cliquer sur **Headers**. Cela vous permettra de définir vos headers de requête.
- Et pour finir, en dessous, en jaune , vous pouvez voir l'emplacement du body de votre réponse.

Nous voulons obtenir des informations sur un utilisateur. Mais comment faire ? Quelle URL utiliser ?

Avant de faire une requête sur l'API GitHub et d'obtenir un utilisateur en particulier, vous devez avant tout faire une chose très importante : consulter la [documentation GitHub](#) ! Et plus précisément la section [Users](#), car c'est celle-ci qui nous intéresse. La documentation, c'est le mode d'emploi d'une API. C'est ainsi que vous trouverez les ressources, URI et endpoints que vous pouvez utiliser pour récupérer des données.

Allez sur la section Users de l'API GitHub via cette URL : <https://developer.github.com/v3/users/>

La partie qui nous intéresse ici est celle qui nous permet d'obtenir **un seul utilisateur** (Get a user en anglais). Cliquez dessus !

The screenshot shows the GitHub REST API documentation for the 'Users' endpoint. The left sidebar lists various API endpoints like Gists, Git database, etc. The main content area shows the 'Users' section under 'REST API / Reference / Users'. It includes a brief description of the API, a detailed explanation of the 'Get the authenticated user' endpoint, and a code sample for a GET request to '/user'. To the right, there's a sidebar titled 'In this article' with a list of other endpoints, where 'Get a user' is highlighted with a red box.

Accédez à la documentation GitHub et cliquez sur Get a user

### Get a user

Provides publicly available information about someone with a GitHub account.

GitHub Apps with the `Plan` user permission can use this endpoint to retrieve information about a user's GitHub plan. The GitHub App must be authenticated as a user. See "[Identifying and authorizing users for GitHub Apps](#)" for details about authentication. For an example response, see 'Response with GitHub plan information' below"

The `email` key in the following response is the publicly visible email address from your GitHub [profile page](#). When setting up your profile, you can select a primary email address to be "public" which provides an email entry for this endpoint. If you do not set a public email address for `email`, then it will have a value of `null`. You only see publicly visible email addresses when authenticated with GitHub. For more information, see [Authentication](#).

The Emails API enables you to list all of your email addresses, and toggle a primary email to be visible publicly. For more information, see "[Emails API](#)".

`GET /users/{username}`

#### Parameters

Name	Type	In	Description
<code>accept</code>	string	header	Setting to <code>application/vnd.github.v3+json</code> is recommended.
<code>username</code>	string	path	

Le texte correspondant à la documentation sur l'utilisateur unique  
Voici ce que dit la documentation en français :

### Obtenir un utilisateur unique

Fournit les informations disponibles publiquement sur quelqu'un ayant un compte GitHub.

Les applications GitHub avec la permission utilisateur Plan peuvent utiliser cet endpoint pour récupérer des informations sur le plan GitHub d'un utilisateur. L'application GitHub doit être authentifiée en tant qu'utilisateur. Voir « Identifier et autoriser les utilisateurs pour les applications GitHub » pour plus de détails sur l'authentification. Pour un exemple de réponse, voir « Réponse avec l'information du plan GitHub ».

La clé e-mail dans la réponse ci-dessous correspond à votre adresse e-mail visible publiquement depuis votre page de profil GitHub. Lors de la création de votre profil, vous pouvez sélectionner une adresse e-mail principale comme « publique », ce qui fournit une entrée e-mail pour cet endpoint. Si vous ne choisissez pas d'adresse e-mail publique pour e-mail, alors sa valeur sera nulle. Vous ne voyez que les adresses e-mail visibles publiquement lorsque vous êtes authentifié dans GitHub. Pour plus d'informations, voir [Authentication](#). L'API Emails vous permet de lister toutes vos adresses e-mail, et d'ajouter un toggle à une adresse e-mail principale pour qu'elle soit visible publiquement. Pour plus d'informations, voir « [API Emails](#) ».

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

La documentation nous apprend plusieurs choses :

1. Cette section fournit les informations disponibles publiquement sur quelqu'un ayant un compte GitHub.
2. On peut accéder aux informations d'un utilisateur via **GET /users/:username**.
3. Un exemple de réponse :

The screenshot shows the GitHub REST API documentation for the `/users/:username` endpoint. The URL is `https://api.github.com/users/octocat`. The response status is `200 OK`. The response body is highlighted with a green box and contains the following JSON data:

```
{  
  "login": "octocat",  
  "id": 1,  
  "node_id": "MDQ6VXNlcjE=",  
  "avatar_url": "https://github.com/images/error/octocat_happy.gif",  
  "gravatar_id": "",  
  "url": "https://api.github.com/users/octocat",  
  "html_url": "https://github.com/octocat",  
  "followers_url": "https://api.github.com/users/octocat/followers",  
  "following_url": "https://api.github.com/users/octocat/following{/other_user}",  
  "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",  
  "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",  
  "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",  
  "organizations_url": "https://api.github.com/users/octocat/orgs",  
  "repos_url": "https://api.github.com/users/octocat/repos",  
  "events_url": "https://api.github.com/users/octocat/events{/privacy}",  
  "received_events_url": "https://api.github.com/users/octocat/received_events",  
  "type": "User",  
  "site_admin": false,  
  "name": "monalisa octocat",  
  "company": "GitHub",  
  "blog": "https://github.com/blog",  
  "location": "San Francisco",  
  "email": "octocat@github.com",  
  "hireable": false,  
  "bio": ""}
```

The response is described as "Response with GitHub plan information" and has a status of "200 OK". To the right of the main content, there is a sidebar titled "In this article" with a list of related API endpoints.

Un exemple de réponse

Parfait !

Si on résume, cela signifie que, pour obtenir la donnée user de l'utilisateur "arfpcchatelot" (oui oui c'est un vrai login, vous allez dans Postman et entrez `https://api.github.com/users/arfpchatelet` dans l'URL, puis appuyez sur Send (Envoyer).

The screenshot shows the Postman application interface. At the top, there are tabs for Home, Workspaces, Reports, and Explore, along with a search bar and a note about the version (Postman 2022.11.0). Below the header, a sidebar lists recent requests: 'https://api.github.com/users/tenderlove' (selected) and 'https://api.github.com/users/arfpchatelot'. The main area displays a 'GET' request to 'https://api.github.com/users/tenderlove'. The 'Headers' tab is selected, showing a table with one row: 'Key' (Content-Type) and 'Value' (application/json). Other tabs include Params, Authorization, Body, Pre-request Script, Tests, and Settings. Below the request details, the 'Body' tab is selected, showing a JSON response for the GitHub user 'tenderlove'. The response is a large object with properties like login, id, node\_id, avatar\_url, etc. The JSON is displayed in a pretty-printed format. At the bottom of the body panel, there are buttons for Find and Replace and Console.

### Requête GET

Et voilà ! Vous avez effectué votre première requête GET avec succès !

**GET** est le verbe HTTP le plus utilisé. Il permet de faire une **requête** afin de récupérer un groupe de données mais aussi des données précises. Comme vous l'avez vu avec l'API GitHub, avec une requête GET vous allez obtenir des données précises grâce à un **ID** ; dans notre exemple, votre nom d'utilisateur GitHub. Notez la façon dont votre navigateur (ou client) utilise une API pour afficher les données sur un site web. Ici, la réponse est sous un format JSON.

Rappelez-vous, en JSON, la réponse est affichée sous un format clé-valeur.

Vous pouvez même effectuer des requêtes GET directement depuis votre navigateur, car les endpoints REST utilisent le même protocole HTTP que le web. Essayez d'aller sur <https://api.github.com/users/arfpchatelot> maintenant, et vous verrez !

**Utilisez la documentation pour connaître le mode d'emploi de l'API**

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

Comme vous pouvez le constater, la documentation d'une API ressemble à un manuel d'utilisation très très détaillé. Étant donné que chaque API est différente, vous ne sauriez pas les utiliser sans une documentation claire et précise.

Quelles informations nous donne cette documentation ?

La documentation liste tous les appels API possibles, les requêtes et réponses typiques, mais surtout les verbes à utiliser pour chaque requête.

Return all films

The Films endpoint returns information about all of the Studio Ghibli films.

### PARAMETERS

#### Query Parameters

fields

string

comma-separated list of fields to include in the response

limit

integer <int64>

amount of results (default 50) (maximum 250)

### Responses

200 An array of films

400 Bad request

404 Not found

## Documentation de l'API Ghibli

Dans cet exemple, pour récupérer tous les films Ghibli, vous voyez qu'il vous suffit de faire une requête GET sur <https://ghibliapi.herokuapp.com/films>. Vous avez aussi une indication sur les messages d'erreur que renvoie l'API en cas de mauvais format de la requête (400) ou d'absence de ressources trouvées (404). Vous voyez aussi qu'en cas de succès, vous obtiendrez une réponse sous forme d'un **array** (ou tableau) qui contiendra la liste des films Ghibli. Un exemple de cette donnée se trouve sur la droite dans l'encadré.

Chaque fois que vous voulez utiliser une API, commencez par consulter la documentation.

The screenshot shows a detailed API documentation page for the 'GET /films' endpoint. At the top, it specifies the method (GET), the endpoint (/films), and a dropdown menu. Below this, under 'REQUEST SAMPLES', there are three code snippets: curl, Ruby, and Python. The curl command is:

```
curl -X GET -H "Content-Type: application/json" https://ghibliapi.herokuapp.com/films
```

Under 'RESPONSE SAMPLES', it shows a green box indicating a successful 200 response: '200 An array of films'. Below this, the actual JSON response is displayed:

```
[{"id": "2baef70d1-42bb-4437-b551-e5fed5e", "title": "Castle in the Sky", "description": "The orphan Sheeta inherits the power of the wind spirit to help her find her home and protect it from the mining company that has destroyed her village.", "director": "Hayao Miyazaki", "producer": "Isao Takahata", "release_date": "1986", "rt_score": "95"}, {"id": "12cfb892-aac0-4c5b-94af-521852e", "title": "Grave of the Fireflies", "description": "In the latter part of World War II, two young orphans, Seita and Setsuko, struggle to survive after their family is killed by American bombers.", "director": "Isao Takahata", "producer": "Toru Hara", "release_date": "1988"}, {"id": "33f2a20a-1a20-433b-8a20-1a201a201a20", "title": "My Neighbor Totoro", "description": "A girl and her younger brother move to the countryside to live with their parents and encounter magical forest spirits.", "director": "Hayao Miyazaki", "producer": "Isao Takahata", "release_date": "1988"}, {"id": "44444444-4444-4444-4444-444444444444", "title": "Porco Rosso", "description": "A former WWI ace becomes a pig-like warlord in the sky who is trying to regain his humanity.", "director": "Hayao Miyazaki", "producer": "Isao Takahata", "release_date": "1992"}, {"id": "55555555-5555-5555-5555-555555555555", "title": "Spirited Away", "description": "A girl is transported to a magical world where she must work at a spa to earn her way back home.", "director": "Hayao Miyazaki", "producer": "Isao Takahata", "release_date": "2001"}]
```

## COMPRENEZ L'IMPORTANCE DE L'AUTHENTIFICATION POUR UNE API

Avant d'aborder d'autres verbes HTTP, il est important que nous parlions davantage de l'**authentification**. L'authentification constitue simplement un moyen pour les API de garantir que le client a les **autorisations** nécessaires pour accéder aux données et les manipuler.

Si je crée une API, je ne veux pas que n'importe qui puisse changer le mot de passe de n'importe quel utilisateur, n'est-ce pas ?

Lorsque nous avions fait notre requête **GET** avec l'API GitHub, nous n'avions pas besoin d'authentification pour obtenir des données utilisateurs. Pourquoi ? Rappelez-vous : il existe des API privées... et... des API publiques !

Ah, GitHub est une API publique !

Oui et non. GitHub est une API qui possède **une partie publique** qui vous permet de faire des requêtes sans autorisation, comme nous l'avons fait dans le chapitre précédent. Cependant, **toute l'application n'est pas publique**. Une partie des endpoints nécessite une authentification, afin d'avoir les autorisations nécessaires pour mettre à jour des données.

Si vous voulez modifier, ajouter ou supprimer des données, GitHub doit vous donner l'autorisation de le faire. Vous ne voulez pas que n'importe qui puisse remplacer votre photo de profil par celle de Homer Simpson, duh ! Donc, pour que ce soit possible, GitHub doit disposer d'un processus d'**authentification**.

Une des méthodes d'authentification les plus utilisées consiste à exiger qu'un développeur s'inscrive par le site web de l'API pour obtenir **un token ou clé**. Une fois le token obtenu, le développeur l'utilise dans sa requête pour s'identifier, et voilà.

C'est quoi un token, au juste ? Une sorte de code ?

Eh bien, en quelque sorte ! Un token est généralement une chaîne longue et unique de lettres et de chiffres aléatoires que l'on assigne à un utilisateur. Le token est un peu comme un numéro de passeport : il est unique et permet de vous identifier. L'API peut donc l'utiliser pour savoir **qui** effectue la requête, et surtout de quel niveau d'**autorisation** cette personne dispose.

Les autorisations peuvent décrire des accès spécifiques à certaines fonctionnalités, comme le nombre de requêtes que l'on peut envoyer mais aussi quelles actions on peut effectuer (si on

est un administrateur ou non, par exemple)... La documentation de l'API se doit de fournir toutes les informations sur les fonctionnalités accessibles à travers un token d'authentification.

### Comment ajouter le token à la requête ?

On l'envoie soit dans les **paramètres du header**, soit dans l'endpoint lui-même.

Pour illustrer ce qu'on vient de voir, nous allons regarder ensemble comment obtenir un token GitHub pas à pas !

Pour commencer, allez sur ce lien : <https://github.com/settings/tokens>.

Vous devriez obtenir cette page :

The screenshot shows the GitHub developer settings page under 'Personal access tokens'. On the left, there's a sidebar with 'GitHub Apps', 'OAuth Apps', and 'Personal access tokens' (which is selected). The main area displays a table for personal access tokens. One token is listed: 'hub for kassandre@Ukulele.local — repo', which was last used within the last 6 months. There are 'Generate new token' and 'Revoke all' buttons at the top right, and a 'Delete' button next to the token entry.

Page GitHub qui vous permet d'obtenir un token

Vous êtes à présent sur la partie qui vous permet de demander à GitHub de vous donner un token afin de pouvoir effectuer des opérations via l'API GitHub.

Cliquez sur **Generate new token**.

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

Settings / Developer settings

GitHub Apps
OAuth Apps
Personal access tokens

### New personal access token

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API](#) over Basic Authentication.

#### Note

OpenClassrooms

What's this token for?

#### Expiration \*

30 days

The token will expire on Sat, Oct 23 2021

#### Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input checked="" type="checkbox"/> <b>repo</b>	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events
<input type="checkbox"/> workflow	Update GitHub Action workflows
<input type="checkbox"/> write:packages	Upload packages to GitHub Package Registry
<input type="checkbox"/> read:packages	Download packages from GitHub Package Registry
<input type="checkbox"/> delete:packages	Delete packages from GitHub Package Registry
<input type="checkbox"/> admin:org	Full control of orgs and teams, read and write org projects
<input type="checkbox"/> write:org	Read and write org and team membership, read and write org projects
<input type="checkbox"/> read:org	Read org and team membership, read org projects
<input type="checkbox"/> admin:public_key	Full control of user public keys
<input type="checkbox"/> write:public_key	Write user public keys
<input type="checkbox"/> read:public_key	Read user public keys
<input type="checkbox"/> admin:repo_hook	Full control of repository hooks
<input type="checkbox"/> write:repo_hook	Write repository hooks
<input type="checkbox"/> read:repo_hook	Read repository hooks
<input type="checkbox"/> admin:org_hook	Full control of organization hooks
<input type="checkbox"/> gist	Create gists
<input type="checkbox"/> notifications	Access notifications
<input type="checkbox"/> user	Update ALL user data
<input type="checkbox"/> read:user	Read ALL user profile data
<input type="checkbox"/> user:email	Access user email addresses (read-only)
<input type="checkbox"/> user:follow	Follow and unfollow users
<input checked="" type="checkbox"/> delete_repo	Delete repositories
<input type="checkbox"/> write:discussion	Read and write team discussions
<input type="checkbox"/> read:discussion	Read team discussions
<input type="checkbox"/> admin:enterprise	Full control of enterprises
<input type="checkbox"/> manage_billing:enterprise	Read and write enterprise billing data
<input type="checkbox"/> read:enterprise	Read enterprise profile data
<input type="checkbox"/> admin:gpg_key	Full control of public user GPG keys (Developer Preview)
<input type="checkbox"/> write:gpg_key	Write public user GPG keys
<input type="checkbox"/> read:gpg_key	Read public user GPG keys

Generate token

Cancel

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

### Génération d'un token

Dans notre cas, nous voulons que GitHub nous donne l'autorisation pour deux choses :

1. Effectuer des actions sur les repositories GitHub.
2. Supprimer des repositories GitHub

Un repo (**repository**, ou *répertoire*) est un espace de stockage pour le code dans GitHub. Suivez l'exemple ci-dessus et n'oubliez pas d'inscrire dans la section **Note** une information qui vous permettra plus tard de vous souvenir à quoi servait ce token. Cela peut être le nom d'une application, de votre API, etc. Dans mon cas, j'ai indiqué testCNAME. Cliquez ensuite sur **Generate token** (générer un token).

The screenshot shows the GitHub 'Personal access tokens' page. At the top, there are two buttons: 'Generate new token' (in blue) and 'Revoke all' (in red). Below this, a heading says 'Personal access tokens' and 'Tokens you have generated that can be used to access the GitHub API.' A note below the heading says 'Make sure to copy your personal access token now. You won't be able to see it again!' A single token entry is listed, which has been redacted except for the first character. To the right of the token is a blue copy icon, which is highlighted with a green box. To the right of the copy icon are 'Delete' and 'Edit' buttons. The token details are: 'Ruby client repository — repo', 'Last used within the last week', and 'Expires on Mon, Nov 15 2021.'

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

### Génération de token

Votre token sera visible à l'emplacement du marqueur noir. J'ai choisi de masquer le mien.

Un token est personnel et privé autant que votre code de carte bleue, alors gardez-le pour vous !

L'encadré bleu explique qu'il faut copier votre token personnel car vous ne serez plus en mesure de le voir.

Gardez-le précieusement et **copiez-le**, ou faites une capture d'écran afin de ne pas le perdre. Nous allons l'utiliser dans les chapitres suivants !

Après avoir copié soigneusement votre token, rechargez la page. Vous devriez voir apparaître comme ci-dessus votre nouveau token par le nom que vous lui avez attribué ; dans mon cas : Open Classrooms. Vous verrez aussi les actions qu'il vous permet de faire sur l'API GitHub ; dans notre cas, les deux que nous avons sélectionnés : *delete\_repo* et *repo*.

## Personal access tokens

[Generate new token](#)

[Revoke all](#)

Tokens you have generated that can be used to access the [GitHub API](#).

[OpenClassrooms](#) — `delete_repo, repo`

Never used

[Delete](#)

Expires on *Sat, Oct 23 2021*.

[Ruby client repository](#) — `repo`

Last used within the last week

[Delete](#)

Expires on *Mon, Nov 15 2021*.

Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

## Actions possibles avec le token

Nous avons vu ensemble comment utiliser des API. Sauf que les API peuvent être créées par des entreprises, des services, ou des développeurs indépendants. Comment être certain qu'elles sont fiables ?

## **Privilégiez la sécurité : choisissez vos API avec discernement**

Comme vous l'avez appris précédemment, il existe des milliers d'API différentes que vous pouvez utiliser dans vos projets. Comme toujours, il est important de garder en tête la sécurité de vos données et de votre application. En tant que développeur, vous êtes responsable de la sécurité des données de vos utilisateurs ! Vous devez vous assurer que vos API proviennent d'une source fiable.

Mais comment faire ? Comment puis-je savoir si une API est fiable ou non ?

Il existe quelques méthodes simples et rapides pour vérifier si une API est fiable ou non. Les API de qualité auront plusieurs mesures de sécurité comme **l'authentification, l'autorisation et le cryptage**. Elles auront aussi été mises à jour récemment ; vous saurez donc qu'elles sont mises à jour en fonction des derniers standards de sécurité.

Prenons un exemple ! Sur GitHub, vous trouverez l'API Pokémon nommée [PokeApi](#) qui a été mise à jour récemment.

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows the GitHub repository for PokeAPI. The repository is public and has 35 issues, 2 pull requests, and 642 forks. The master branch is selected. The commit history shows several recent updates:

- docs: update graphql internal link [ci skip] (Naramsim, 14 days ago)
- Create SECURITY.md (Naramsim, 14 days ago)
- Merge pull request #649 from technicity1/master (Naramsim, 16 days ago)
- Fixed typo on species 186 (technicity1, 17 days ago)
- Merge pull request #648 from stormonster/flavor\_typo\_species\_236 (Naramsim, 17 days ago)
- Actually removed soft hyphen properly this time (stormonster, 17 days ago)
- Fixed typo on species 236. Fixes issue #646 (stormonster, 17 days ago)

Un exemple d'une API mise à jour récemment #PokeApi

Vous pouvez constater sur leur GitHub que la date de leur dernier commit est assez récente ; vous savez donc que cette API est maintenue et mise à jour.

Avant chaque utilisation d'une API externe, vérifiez la date de la dernière mise à jour sur GitHub ou bien sur son site Internet, lisez la documentation et si vous avez des doutes (ou non), regardez en ligne des avis ou posez simplement la question à un autre développeur.

## MANIPULEZ DES DONNEES AVEC L'API GITHUB

Maintenant que vous avez un token d'authentification, vous pouvez utiliser l'API pour mettre à jour votre profil GitHub !

Dans ce chapitre, nous allons pratiquer et utiliser le reste des opérations CRUD – Create (créer), Update (mettre à jour) et Delete (supprimer) – et leurs verbes HTTP équivalents – POST (publier), PUT (mettre) et DELETE (supprimer).

Je vous invite vivement à suivre pas à pas les étapes suivantes et à pratiquer afin de maximiser votre apprentissage.

## Créez un repo GitHub avec la méthode POST

Pour **créer** quelque chose de nouveau, ou une nouvelle ressource, on utilise le verbe **HTTP POST** (publier). Qu'il s'agisse d'un nouveau tweet, d'une nouvelle photo ou d'une nouvelle *publication* (vous comprenez ?). Par exemple, dès que vous remplissez un formulaire en ligne ou que vous en utilisez un pour vous inscrire et vous créer un nouveau compte, le verbe associé par défaut est POST.

Comment l'API sait-elle ce qu'on essaie de **créer** quelque chose ?

C'est là qu'intervient le **body** ! Vous vous souvenez ? On l'a vu dans les chapitres précédents. Le body accompagne les requêtes POST et PUT pour contenir des informations supplémentaires. Vous pouvez intégrer les données que vous voulez créer dans le body de votre requête en utilisant du **JSON**.

Mettons tout ça en pratique ; je vous assure que ça fera sens !

On veut créer un nouveau repo sur GitHub pour notre API. Super, mais comment faire ? 1re étape : la documentation !

Regardons ce que nous dit la documentation GitHub au sujet des repositories : <https://developer.github.com/v3/repos>

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows the GitHub REST API documentation for the 'Repositories' endpoint. The left sidebar contains a navigation menu with links like Gists, Git database, Gitignore, Interactions, Issues, Licenses, Markdown, Meta, Migrations, Organizations, Packages, Projects, Pulls, Rate limit, Reactions, and a section for **Repositories** which includes SCIM, Search, Secret scanning, Teams, Users, and GitHub App permissions. Below this is a 'GUIDES' section with a dropdown arrow. The main content area has a breadcrumb trail: REST API / Reference / Repositories. The title 'Repositories' is bolded. A text block states: 'The Repos API allows to create, manage and control the workflow of public and private GitHub repositories.' Below this is a section titled 'List organization repositories' with a brief description: 'Lists repositories for the specified organization.' It shows a 'GET' request method and the URL '/orgs/{org}/repos'. A table titled 'Parameters' lists three columns: Name, Type, and Description. The first row has 'accept' as the name, 'string' as the type, and a header note: 'Setting to application/vnd.github.v3+json is recommended. See preview notices'. The second row has 'org' as the name, 'string' as the type, and 'path' as the description. The third row has 'type' as the name, 'string' as the type, and a detailed description: 'Specifies the types of repositories you want returned. Can be one of all, public, private, forks, sources, member, internal. Note: For GitHub AE, can be one of all, private, forks, sources, member, internal. Default: all. If your organization is associated with an enterprise account using GitHub Enterprise Cloud or GitHub Enterprise Server 2.20+, type can also be internal. However, the internal value is not yet supported when a GitHub App calls this API with an installation'. On the right side, there is a sidebar titled 'In this article' with a long list of GitHub API endpoints, and a specific link 'Create a repository for the authenticated user' is highlighted with a green border.

Documentation GitHub concernant les repositories

Houlà, on a beaucoup de choix ! Nous, ce qu'on veut, c'est créer un nouveau repo. Cliquez en bas à droite sur [Create a repository for the authenticated user](#).

### Create a repository for the authenticated user

Creates a new repository for the authenticated user.

#### OAuth scope requirements

When using [OAuth](#), authorizations must include:

- `public_repo` scope or `repo` scope to create a public repository. Note: For GitHub AE, use `repo` scope to create an internal repository.
- `repo` scope to create a private repository.

**POST** /user/repos

#### Parameters

Name	Type	In	Description
<code>accept</code>	string	header	Setting to <code>application/vnd.github.v3+json</code> is recommended. <a href="#">See preview notices</a>
<code>name</code>	string	body	<b>Required.</b> The name of the repository.
<code>description</code>	string	body	A short description of the repository.
<code>homepage</code>	string	body	A URL with more information about the repository.
<code>private</code>	boolean	body	Whether the repository is private. Default: <code>false</code>
<code>has_issues</code>	boolean	body	Whether issues are enabled. Default: <code>true</code>
<code>has_projects</code>	boolean	body	Whether projects are enabled. Default: <code>true</code>

Documentation pour créer un nouveau repo

Dans le premier encadré en vert, on peut lire “*Crée un nouveau repository pour un utilisateur authentifié*”.

Pour créer un nouveau repository, il faut faire une requête **POST** vers `/user/repos`.

Parfait, on avance ! Maintenant, scrollez un peu et vous tomberez sur les paramètres.

L'encadré vert vous montre que le paramètre **name** est obligatoire.

Mais ce n'est pas tout ! La documentation nous donne un exemple de requête et de réponse réussies.

### Response

Status: 201 Created

```
{
  "id": 1296269,
  "node_id": "MDEwOlJlcG9zaXRvcnkxMjk2MjY5",
  "name": "Hello-World",
  "full_name": "octocat/Hello-World",
  "owner": {
    "login": "octocat",
    "id": 1,
    "node_id": "MDQ6VXNlcjE=",
    "avatar_url": "https://github.com/images/error/octocat_happy.gif",
    "gravatar_id": "",
    "url": "https://api.github.com/users/octocat",
    "html_url": "https://github.com/octocat",
    "followers_url": "https://api.github.com/users/octocat/followers",
    "following_url": "https://api.github.com/users/octocat/following{/other_user}",
    "gists_url": "https://api.github.com/users/octocat/gists{/gist_id}",
    "starred_url": "https://api.github.com/users/octocat/starred{/owner}{/repo}",
    "subscriptions_url": "https://api.github.com/users/octocat/subscriptions",
    "organizations_url": "https://api.github.com/users/octocat/orgs",
    "repos_url": "https://api.github.com/users/octocat/repos",
    "events_url": "https://api.github.com/users/octocat/events{/privacy}",
    "received_events_url": "https://api.github.com/users/octocat/received_events",
    "type": "User",
    "site_admin": false
},  
  "private": false
```

Une réponse réussie

Nous avons notre endpoint, notre verbe http, nos paramètres et même un exemple ! Allons faire cette requête.  
Lancez Postman !

Sélectionnez POST dans le menu déroulant puis tapez l'endpoint pour créer un nouveau repo via l'API GitHub: <https://api.github.com/user/repos> ; et appuyez sur Send.

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows a POSTMAN interface. At the top, it says "GET https://api.github.com/user/repos". Below that, there's a "Headers (7)" tab selected. A table shows one header entry: "Key" (Authorization) and "Value" (Token). The "Body" tab is also visible. At the bottom, the "Test Results" section shows a red error message: "Status: 401 Unauthorized Time: 162 ms Size: 1.14 KB". The JSON response body is highlighted with a green box and contains the following text:

```
1 {  
2   "message": "Requires authentication",  
3   "documentation_url": "https://docs.github.com/rest/reference/repos#list-repositories-for-the-authenticated-user"  
4 }
```

### Message d'erreur

Arf, mais on a une erreur. Qu'est-ce que c'est ?

Le message nous indique qu'une authentification est nécessaire. Le code de réponse HTTP est 401 : Unauthorized (401 Non autorisé). Eh oui, notre token !

Reprenez votre token d'authentification API GitHub, celui que vous avez précieusement gardé. Ajoutez-le, comme sur l'image suivante, dans l'onglet Headers sous forme de clé-valeur où la clé sera Authorization et la valeur sera Token, votre token. Si mon token est *abcde*, alors la valeur sera : *token abcde*.

Ensuite, appuyez sur Send et regardez la réponse.

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows a POST request to <https://api.github.com/user/repos>. The Headers tab is selected, showing Content-Type: application/json and Authorization: Bearer [REDACTED]. The response status is 400 Bad Request.

```
1 {  
2   "message": "Body should be a JSON object",  
3   "documentation_url": "https://docs.github.com/rest/reference/repos#create-a-repository-for-the-authenticated-user"  
4 }
```

Statut de la requête avec le token

Encore une erreur ?! Mmmh... on a le bon endpoint, le token. Cette fois-ci, le code de réponse HTTP est : 400 Bad Request. Mais oui ! Il nous manque les paramètres . Eh oui, on demande à GitHub de nous créer un repo, mais on ne lui donne pas les informations.

Cliquez sur la section Body, puis sur Raw, binary et sélectionnez JSON.

The screenshot shows a GET request to <https://api.github.com/users/repo>. The Body tab is selected with JSON selected. The response shows a new repository creation.

```
1 {  
2   "name": "OpenClassrooms API",  
3   "description": "Nouveau repo !"  
4 }
```

Sélectionnez JSON

Maintenant, on peut ajouter nos paramètres sous forme de clé-valeur. Il nous faut un nom de repo qui est obligatoire, et ajoutons une description.

{

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

```
"name": "OpenClassrooms API",
"description": "Nouveau repo !"

}
```

Appuyez sur Send et observez la réponse.

The screenshot shows a POST request to `https://api.github.com/user/repos`. The request body is a JSON object:

```
1 {
2   ... "name": "OpenClassrooms API",
3   ... "description": "Nouveau repo !"
4 }
```

The response tab shows the following details:

- Status: 201 Created
- Time: 116 ms
- Pretty JSON view of the created repository object:

```
1 {
2   "id": 409670912,
3   "node_id": "R_kgDOGGsVAA",
4   "name": "OpenClassrooms-API",
5   "full_name": "Kadaaran/OpenClassrooms-API",
6   "private": false,
7   "owner": {
8     "login": "Kadaaran",
9     "id": 11220823,
10    "node_id": "MDQ6VXNlcjExMjIwODIz",
11    "avatar_url": "https://avatars.githubusercontent.com/u/11220823?v=4",
12    "gravatar_id": "",
13    "url": "https://api.github.com/users/Kadaaran",
14    "html_url": "https://github.com/Kadaaran",
15    "followers_url": "https://api.github.com/users/Kadaaran/followers",
16    "following_url": "https://api.github.com/users/Kadaaran/following{/other_user}",
17    "gists_url": "https://api.github.com/users/Kadaaran/gists{/gist_id}",
18    "starred_url": "https://api.github.com/users/Kadaaran/starred{/owner}{/repo}"}
```

Le repo a été créé

Notre repository a été créé. Le code de réponse HTTP est bien 201 Created. L'API nous renvoie les informations du nouveau repo créé. On peut aussi vérifier sur GitHub si le repo apparaît bien dans notre liste, et c'est le cas !

### Vérification de la création du repo

Pour créer un repo GitHub en utilisant l'API GitHub, nous avons :

- consulté la [documentation](#) GitHub pour trouver l'endpoint adapté : POST user/repos ;
- ajouté notre token d'authentification à nos paramètres dans le header avec Postman ;
- ajouté les détails de notre repo à notre body en JSON avec Postman ;
- vu qu'une fois que nous avons effectué une requête réussie avec notre API, le repo est apparu sur l'UI de notre profil GitHub !

Consultez la vidéo pour revoir comment faire :

Et voilà ! Vous avez un nouveau repo ! Et si maintenant on essayait de le modifier avec l'API GitHub ?

### **Mettez à jour un repo GitHub avec les méthodes PUT/PATCH**

Vous avez créé votre repo GitHub, mais la description ne vous plaît plus trop, vous changez d'avis et vous voulez la modifier ! Vous pouvez utiliser PUT pour **mettre à jour** une ressource déjà existante dans votre API.

Sur Internet, on croise souvent PUT avec PATCH ? Les deux signifient la même chose ou est-ce qu'ils sont différents ?

- PUT : met à jour la ressource complète (c'est-à-dire, remplace tout).
- PATCH : met à jour uniquement la partie de la ressource qui a été envoyée.

On ne va pas s'attarder ici sur la différence entre les deux. Mais si vous ne savez pas lequel utiliser et pour quelle situation : consultez la documentation de l'API !

Et c'est ce que nous allons faire ! Reprenons la documentation des repos GitHub mais cette fois-ci cliquons sur [Update a repository](#).

Pour mettre à jour un repo, il nous faut utiliser la méthode PATCH sur l'endpoint /repos/:Owner/:repo.

Les deux points (:) devant *owner* et *repo* signifient qu'il nous faut l'identifiant unique du repository que vous voulez modifier, et celui du owner (propriétaire) du repo : vous, en l'occurrence.

Changez le verbe par PATCH. Puis entrez l'endpoint correspondant. Mon username est Kadaaran et mon repo se nomme OpenClassrooms-API. Changez Kadaaran par votre username et OpenClassrooms-API par votre nom de repo s'il est différent. Dans la section Body, tapez une nouvelle description. Dans mon cas, j'ai mis : Paragraphe ci-dessus à revoir.

```
{  
  "description": "Ceci est une nouvelle description meilleure que la précédente !!"  
}
```

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows a Postman interface with a PATCH request to <https://api.github.com/repos/Kadaaran/OpenClassrooms-API>. The Body tab is selected, showing the following JSON payload:

```
1 {  
2   ... "description": "Ceci est une nouvelle description plus cool que la précédente."  
3 }
```

Changement de la description

Appuyez sur Send et regardez la réponse.

The screenshot shows the Postman interface after sending the PATCH request. The status bar indicates a 200 OK response. The Body tab displays the updated repository details, including the new description.

```
1 {  
2   "id": 409670912,  
3   "node_id": "R_kgDOGGSVAA",  
4   "name": "OpenClassrooms-API",  
5   "full_name": "Kadaaran/OpenClassrooms-API",  
6   "private": false,  
7   "owner": {  
8     "login": "Kadaaran",  
9     "id": 11220823,  
10    "node_id": "MDQ6VXNlcjExMjIwODIz",  
11    "avatar_url": "https://avatars.githubusercontent.com/u/11220823?v=4",  
12    "gravatar_id": "",  
13    "url": "https://api.github.com/users/Kadaaran",  
14    "html_url": "https://github.com/Kadaaran",  
15    "followers_url": "https://api.github.com/users/Kadaaran/followers",  
16    "following_url": "https://api.github.com/users/Kadaaran/following{/other_user}",  
17    "gists_url": "https://api.github.com/users/Kadaaran/gists{/gist_id}",  
18    "starred_url": "https://api.github.com/users/Kadaaran/starred{/owner}{/repo}"  
}
```

Description mise à jour

Notre repo a été mis à jour !

Nous obtenons la réponse avec le body : le code de réponse HTTP est bien 200 OK.

Vous pouvez vérifier sur GitHub et vous devriez voir votre nouvelle description apparaître.

La nouvelle description

Pour **mettre à jour** un repo GitHub, nous avons :

- vérifié la documentation GitHub pour l'URI approprié : PATCH /repos/wner/:repo ;
- utilisé Postman, changé notre verbe HTTP en PATCH, et saisi le propriétaire et le nom de repository que nous voulons modifier, ainsi que changé la description dans le body ;
- vu la nouvelle description de notre repository dans notre UI GitHub

Bravo ! Vous avez créé un repo, puis vous l'avez modifié. Essayons à présent de le supprimer !

## SUPPRIMEZ VOTRE REPO GITHUB AVEC LA METHODE DELETE

Vous décidez à présent que vous n'avez pas réellement besoin de votre repo GitHub – alors allez-y et supprimez-le avec DELETE !

Allez hop ! On reprend la même gymnastique que pour les requêtes précédentes. Tout d'abord : la documentation ! Comment supprime-t-on un repository GitHub ?

Dans la liste des méthodes disponibles, nous trouvons : [Delete a repository](#). Cliquez dessus !

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows the GitHub Docs page for deleting a repository. It includes a sidebar with navigation links like Actions, Activity, Apps, etc. The main content area has sections for 'Delete a repository', 'Parameters', 'Code samples' (Shell and JavaScript), and a 'In this article' sidebar with various API endpoints.

**Delete a repository**

Deleting a repository requires admin access. If OAuth is used, the `delete_repo` scope is required.

If an organization owner has configured the organization to prevent members from deleting organization-owned repositories, you will get a `403 Forbidden` response.

**DELETE** `/repos/{owner}/{repo}`

**Parameters**

Name	Type	In	Description
<code>accept</code>	string	header	Setting to <code>application/vnd.github.v3+json</code> is recommended.
<code>owner</code>	string	path	
<code>repo</code>	string	path	

**Code samples**

**Shell**

```
curl \
-X DELETE \
-H "Accept: application/vnd.github.v3+json" \
https://api.github.com/repos/octocat/hello-world
```

**JavaScript (@octokit/core.js)**

```
await octokit.request('DELETE /repos/{owner}/{repo}', {
  owner: 'octocat',
  repo: 'hello-world'
})
```

**In this article**

- List organization repositories
- Create an organization repository
- Get a repository
- Update a repository
- Delete a repository**
- Enable automated security fixes
- Disable automated security fixes
- List repository contributors
- Create a repository dispatch event
- List repository languages
- List repository tags
- List repository teams
- Get all repository topics
- Replace all repository topics
- Transfer a repository
- Check if vulnerability alerts are enabled for a repository
- Enable vulnerability alerts
- Disable vulnerability alerts
- Create a repository using a template
- List public repositories
- List repositories for the authenticated user
- Create a repository for the authenticated user
- List repositories for a user
- Autolinks
- List all autolinks of a repository
- Create an autolink reference for a repository
- Get an autolink reference of a repository
- Delete an autolink reference from a repository

## Documentation pour supprimer un repo

Selon la documentation, pour supprimer un repository GitHub, il nous faut utiliser le verbe `DELETE` sur l'endpoint `/repos/Owner/:repo`. Si la requête est un succès, alors nous devrions avoir `204 No Content` (204 Aucun contenu) comme réponse. Essayons avec Postman.

The screenshot shows a Postman request configuration. The method is set to `DELETE`, the URL is `https://api.github.com/repos/Kadaaran/OpenClassrooms-API`, and the body is left empty as indicated by the message "This request does not have a body".

Request details:

- Method: `DELETE`
- URL: `https://api.github.com/repos/Kadaaran/OpenClassrooms-API`
- Body: This request does not have a body

## Suppression d'un repo

Tapez les informations nécessaires : votre username et le nom du repo que vous souhaitez supprimer. Pas besoin de body pour `DELETE`, cliquez sur `none`. Puis cliquez sur `Send` et regardez la réponse obtenue.

# Conception & Développement Informatique

## DEVELOPPEUR WEB ET WEB MOBILE

TYPE: APPRENTISSAGE / COURS THÉORIQUE - Franck CHATELOT



CENTRE DE RÉADAPTATION  
MULHOUSE

Rééducation et Formation Professionnelle

The screenshot shows a Postman request for a DELETE operation on the GitHub API endpoint `https://api.github.com/repos/Kadaaran/OpenClassrooms-API`. The 'Body' tab is selected, showing the message "This request does not have a body". The 'Headers' tab shows 21 entries. The response section indicates a status of 204 No Content, with a green border around the status code. Other details include a time of 387 ms and a size of 1.07 KB.

Le repo est supprimé

Et voilà ! Votre repository est supprimé à tout jamais !

Nous obtenons un body vide et un statut 204 No Content comme l'a indiqué la documentation.

Vérifiez sur GitHub que votre repo a bien été supprimé.

Vérification de la suppression

Vous avez vu comment fonctionnent POST, PUT et DELETE et vous les avez vous-même utilisés avec l'API GitHub et Postman.

Vous les avez utilisés dans le contexte d'un repo GitHub, mais vous pouvez utiliser ces verbes HTTP pour tout type de fonctionnalité, en fonction de l'API.

- Vous avez vu dans la documentation GitHub que l'URI correct est le même que pour Update, mais avec un nouveau verbe HTTP : DELETE /repos/:Owner/:repo.
- Vous avez vu que notre repo est supprimé et ne se trouve plus dans notre UI GitHub !