REST API

REST signifie ***Re****presentational****S****tate****T****ransfer* (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](https://twitter.com/) – votre navigateur utilise HTTP pour faire une requête de ce site web au serveur. REST fonctionne de la même façon  !

Si vous ne vous souvenez plus de ce qu’est le protocole HTTP, jetez un œil à ce chapitre du cours [**Comprendre le web**](https://openclassrooms.com/fr/courses/1946386-comprendre-le-web/6874807-decouvrez-les-protocoles).

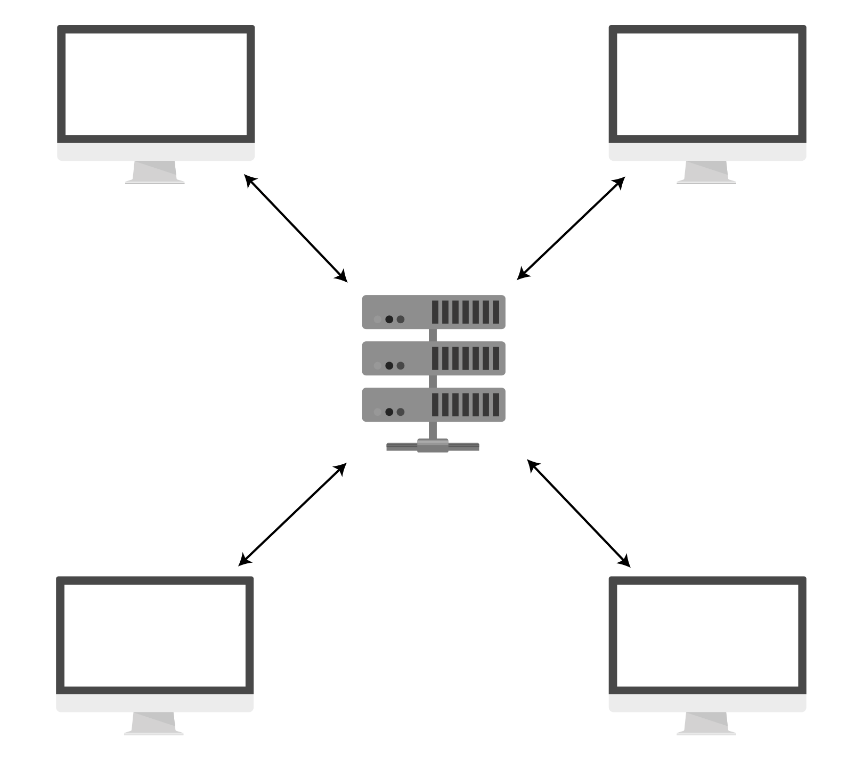
Il y a **six** lignes directrices architecturales clés pour les API REST. Voyons ensemble de quoi il s’agit :

**#1 : Client-serveur separation**

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, 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 :



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 ca 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.

**En résumé**

* Toutes les API ne sont pas RESTful et les API REST ont des lignes directrices architecturales spécifiques.
* Les avantages clés des API REST sont les suivants :
  + la séparation du client et du serveur, qui aide à scaler plus facilement les applications ;
  + le fait d’être stateless, ce qui rend les requêtes API très spécifiques et orientées vers le détail ;
  + la possibilité de mise en cache, qui permet aux clients de sauvegarder les données, et donc de ne pas devoir constamment faire des requêtes aux serveurs.
* SOAP est un autre type d’API, mais est plus utilisé dans les grandes entreprises.

*Vous venez de voir la structure d’une API REST et ses avantages ; il est temps de voir ce qui constitue une API REST : les ressources. Suivez-moi dans le prochain chapitre, et attaquons-nous aux ressources !*

##### Méthodes HTTP

* GET : utilisée pour récupérer des informations en rapport avec l'URI ; il ne faut en aucun cas modifier ces données au cours de cette requête. Cette méthode est dite safe (sécuritaire), puisqu'elle n'affecte pas les données du serveur. Elle est aussi dite idempotent, c'est-à-dire qu'une requête faite en GET doit toujours faire la même chose (comme, renvoyer une liste d'utilisateurs à chaque fois que la requête est faite - d'une requête à l'autre, on ne renverra pas des produits si le client s'attend à une liste d'utilisateurs !).
* POST : utilisée pour créer une ressource. Les informations (texte, fichier…) pour créer la ressource sont envoyées dans le contenu de la requête. Cette méthode n'est ni safe, ni idempotent.
* PUT : utilisée pour remplacer les informations d'une resource avec ce qui est envoyé dans le contenu de la requête. Cette méthode n'est ni safe, ni idempotent.
* PATCH : utilisée pour modifier une ressource. La différence avec une requête avec la méthode PUT est que l'action à effectuer sur la ressource est indiquée dans le contenu de la requête. Prenons un exemple : nous souhaitons rattacher un utilisateur à une organisation, dans le contenu de la requête, il sera indiqué qu'il s'agit d'un rattachement à une organisation en plus des informations à mettre à jour.

La différence fondamentale entre les méthodes  PUT  et  PATCH  est la manière dont les modifications sont demandées. Pour une requête  PUT , l'ensemble des données sont fournies, il suffit simplement de récupérer la nouvelle version de la ressource pour la persister par exemple. Pour une requête  PATCH , l'action est fournie parce qu'il peut y avoir différentes manières de mettre à jour une ressource.

* DELETE : utilisée pour supprimer une ou plusieurs ressources. Les ressources à supprimer sont indiquées dans l'URI.
* OPTIONS : utilisée pour obtenir la liste des actions possibles pour une ressource donnée (suppression, ajout…).
* CONNECT : utilisée pour établir une première connexion avec le serveur pour une URI donnée.
* HEAD : même principe que pour la méthode GET, mais seules les entêtes devront être renvoyées en réponse.
* TRACE  : utilisée pour connaître le chemin parcouru par la requête à travers plusieurs serveurs. En réponse, une entêteviasera présente pour décrire tous les serveurs par lesquels la requête est passée.

En général, dans le cadre d'une API RESTFul, seules les méthodes  GET ,  POST ,  PUT ,  DELETE  sont utilisées.

**Réponse HTTP**

Une réponse HTTP émane d'un serveur (tout logiciel dans la capacité de forger une réponse HTTP). Une réponse est constituée des éléments suivants :

1. La première ligne (*status line*) doit contenir :
   * la **version du protocole** utilisée
   * le **code status**
   * l'**équivalent textuel du code status**
2. Les entêtes (*headers*), une entête par ligne, chaque ligne finie par le caractère spécial "retour à la ligne" (CRLF)
3. Le contenu de la réponse (*body*), doit être séparé de deux caractères spéciaux "retour à la ligne (CRLFCRLF) - optionnel.

##### Code status

Il est important d'accorder une attention toute particulière aux codes status qui doivent être choisis avec sagesse lorsqu'il vous faudra rendre une réponse. Il existe cinq catégories de code status, les voici :

|  |  |
| --- | --- |
| Catégorie | Description |
| 1xx : les informations | Une réponse doit contenir ce type de code status lorsqu'il s'agit d'informer le client de l'état de la demande. C'est utile pour indiquer que, par exemple, la requête a bien été reçue et que le traitement vient de commencer, dans le cas de traitements asynchrones par exemple. |
| 2xx : les succès | Tout s'est bien passé sur le serveur. |
| 3xx : les redirections | Une redirection est sur le point d'être effectuée.  Petite exception avec le code 304 qui signifie que le contenu n'a pas changé, dans un contexte de cache. |
| 4xx : les erreurs client | La requête contient une erreur et ne peut pas être traitée. |
| 5xx : les erreurs serveur | Le serveur vient de rencontrer un problème empêchant le traitement de la requête |

Voyons maintenant comment une application REST devient RESTFul, c'est à dire pleinement REST.

**La sérialisation**

ll s'agit de créer un objet php et de le transformer en format linéarisé pour l'envoyer à un client. En général, nous sommes dans ce cas d'utilisation lorsqu'il faut présenter des données. Ici, nous allons présenter les données d'un article. Pour l'instant, nous n'avons pas d'article en base de données, du coup, nous allons tout simplement créer un objet factice dans notre *controller*, le sérialiser, et le mettre en contenu de notre objet réponse.

N'oublions pas de respecter les contraintes de REST concernant :

* l'URI qui doit être unique pour chaque ressource (ici, ce sera  /articles/{id} , même si nous n'utilisons pas encore l'ID pour aller chercher notre objet en base de données) ;
* le contenu de notre réponse doit être auto-décrit, il nous faut donc indiquer dans la réponse quel type de données nous envoyons (nous indiquerons qu'il s'agit de JSON).

#### la désérialisation

Il s'agit maintenant de recevoir du JSON pour le transformer en objet. Complétons notre classe de controller avec une nouvelle action. Dans cette action, nous prenons soin de n'accepter que les requêtes en POST, car le cas d'utilisation est de créer un nouvel article. Nous allons donc désérialiser le JSON reçu, en obtenir un objet article, puis le persister en base de données

# **Guzzle Documentation**

Guzzle is a PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services.

* Simple interface for building query strings, POST requests, streaming large uploads, streaming large downloads, using HTTP cookies, uploading JSON data, etc...
* Can send both synchronous and asynchronous requests using the same interface.
* Uses PSR-7 interfaces for requests, responses, and streams. This allows you to utilize other PSR-7 compatible libraries with Guzzle.
* Abstracts away the underlying HTTP transport, allowing you to write environment and transport agnostic code; i.e., no hard dependency on cURL, PHP streams, sockets, or non-blocking event loops.
* Middleware system allows you to augment and compose client behavior.