Engineer

REST

🏛 Style architectural du web organisé autour des ressources

Ce que c’est

REST, abréviation de Representational State Transfer, est le style architectural exposé par Roy Fielding au chapitre 5 de sa thèse en 2000. Ce n’est ni un protocole ni une norme. C’est un ensemble de contraintes sur les interactions possibles entre clients, serveurs et intermédiaires. Fielding précise ce qu’apporte chaque contrainte et, pour la plupart, son coût. RESTful est l’adjectif : au sens strict, il décrit un système respectant toutes ces contraintes. Dans l’usage courant, les deux termes sont étendus bien au-delà, à presque toute API HTTP échangeant du JSON, qu’elle respecte ou non les contraintes ci-dessous.

L’idée centrale est la ressource. Selon Fielding, « toute information pouvant être nommée peut être une ressource » : document, image, collection d’autres ressources, personne, voire météo actuelle d’une ville. Ce qui traverse le réseau n’est jamais la ressource elle-même, mais une représentation : une suite d’octets accompagnée de métadonnées qui décrivent ces octets, comme un document HTML ou une image JPEG. Un client agit sur une ressource en envoyant ou en recevant une représentation de son état actuel ou souhaité.

Fielding construit ce style à partir d’un ensemble vide de contraintes, en les ajoutant une à une. Client-serveur sépare l’interface utilisateur du stockage des données, pour leur permettre d’évoluer indépendamment. La contrainte d’absence d’état signifie que chaque requête contient tout ce qui est nécessaire pour la comprendre, sans s’appuyer sur un contexte conservé sur le serveur : l’état de session réside donc entièrement chez le client. Le cache exige que chaque réponse soit marquée, explicitement ou implicitement, comme pouvant ou non être mise en cache, afin qu’un client puisse réutiliser une réponse pouvant être mise en cache pour une requête ultérieure équivalente. L’interface uniforme, décrite plus bas, donne à chaque composant la même interface générale. Un système en couches signifie que chaque composant ne voit que la couche avec laquelle il communique directement. Cela laisse de la place à des

proxys, passerelles et pare-feu intermédiaires sans modifier les interfaces de part et d'autre. Le code à la demande permet au serveur d'étendre un client en lui envoyant des scripts à exécuter. Fielding en fait la seule contrainte facultative, car elle réduit la visibilité.

Il appelle l'interface uniforme la caractéristique centrale qui distingue REST des autres styles en réseau et la divise en quatre contraintes : les ressources sont identifiées, sur le web par des URL ; elles sont manipulées par leurs représentations ; les messages sont autodescriptifs, rendant leur sens visible à chaque intermédiaire ; et l'hypermédia est le moteur de l'état de l'application. Ce dernier point signifie que le client choisit sa prochaine transition à partir des représentations reçues, comme une personne suit des liens dans un navigateur.

Utilisez le concept pour prendre une décision

Utilisez « Qu'est-ce que REST ? » pour éclairer une décision réelle. Nous pouvons vous aider à relier ce concept au système que vous construisez ou achetez. La partie invisible pour les utilisateurs : où résident les données, quelles règles s'appliquent et comment les autres systèmes peuvent y accéder.

Découvrir nos prestations Contactez-nous →

Vérifiez les hypothèses derrière le terme

Appuyez-vous sur l'explication et les réalisations liées pour formuler les questions de votre projet. Nous pouvons examiner les hypothèses avec vous avant le choix d'un outil. 6 pipelines ETL/ELT conçus et exploités — 6 pipelines ETL sur trois moteurs de bases de données

6 pipelines ETL/ELT conçus et exploités Contactez-nous →

Pourquoi c'est important

Chaque contrainte apporte quelque chose de précis. L'absence d'état permet à un système de supervision de comprendre une requête sans regarder au-delà, facilite la récupération après une défaillance partielle et permet au serveur de libérer des ressources dès sa réponse. Un cache peut supprimer complètement certaines interactions. Les couches permettent aux intermédiaires de répartir la charge entre réseaux et machines, et à un cache à la périphérie d'une organisation de servir tout le monde à l'intérieur. Comme l'interface uniforme dissocie les implémentations des services rendus, client et serveur peuvent chacun évoluer sans obliger l'autre à changer au même rythme.

Fielding est tout aussi explicite sur les coûts. L'absence d'état répète dans chaque requête des données qu'un contexte partagé sur le serveur aurait évité de transmettre. Un cache peut distribuer des données périmées. Chaque couche ajoute du travail et de la latence. L'interface uniforme dégrade l'efficacité, car les informations voyagent sous une forme standardisée plutôt que façonnée pour les besoins d'une application précise. Fielding l'a conçue pour les transferts de

données hypermédia à gros grain, cas courant sur le web, et précise qu'elle n'est pas optimale pour les autres interactions. C'est sur ce terrain que se placent les alternatives. Un serveur GraphQL reçoit chaque requête sur un point d'entrée unique, identifie ses entités autrement que par des URL et laisse le client choisir exactement les champs renvoyés, tandis que gRPC appelle des méthodes déclarées dans une définition de service. Chacun renonce à une partie de l'interface uniforme pour un autre type de trafic.

Pour une API HTTP, une bonne part de l'interface uniforme existe déjà. HTTP définit des méthodes au sens convenu, des codes de statut et des champs d'en-tête pour le cache et les requêtes conditionnelles. Ce sens étant partagé, un proxy ou un cache intermédiaire peut agir sur un message sans rien connaître de l'API concernée. Une API utilisant ces éléments comme ils sont définis bénéficie gratuitement de cette mécanique. Une API qui donne à ses clients des modèles d'URL à remplir plutôt que des liens à suivre abandonne la contrainte d'hypermédia. Ses clients sont liés à l'organisation des URL du serveur : quand celle-ci change, ils cessent de fonctionner. C'est ce couplage que l'interface uniforme doit supprimer.

À quoi cela ressemble en pratique

L'échange ci-dessous crée une commande. La réponse contient l'adresse de la nouvelle ressource dans `Location` et une étiquette d'entité dans `ETag`. La RFC 9110 prend elle-même pour exemple un `ETag` dans une réponse 201 : cette étiquette peut servir de validateur dans les requêtes conditionnelles ultérieures, pour empêcher deux clients d'écraser silencieusement leurs modifications respectives.

```
POST /orders HTTP/1.1
Host: api.example.com
Content-Type: application/json

{"item": "notebook", "quantity": 2}

HTTP/1.1 201 Created
Location: /orders/7421
ETag: "v1"
Content-Type: application/json

{"id": 7421, "item": "notebook", "quantity": 2, "status": "open"}
```

Chaque méthode porte un sens sur lequel tout intermédiaire peut s'appuyer. La RFC 9110 définit GET, HEAD, OPTIONS et TRACE comme sûres : le client ne demande ni n'attend aucun changement d'état sur le serveur. PUT et DELETE, avec les méthodes sûres, sont idempotentes : plusieurs requêtes identiques ont le même effet prévu qu'une seule. POST n'a aucune de ces deux propriétés, et la RFC 5789 dit la même chose de PATCH. Ces deux propriétés permettent à un robot d'exploration de suivre des liens sans dommage, à un cache d'anticiper des lectures et à un client de répéter un PUT dont la connexion a été coupée avant l'arrivée de la réponse.

Une API REST peut être pilotée avec un simple client HTTP générique, et ce site montre le gain comme le coût. Lorsqu'un second serveur a été provisionné à partir du code, les pare-feu des deux fournisseurs cloud ont été configurés directement par leurs API REST avec une tâche HTTP générique plutôt qu'avec une collection propre au fournisseur. Cela supprime une dépendance et sa dérive de versions, au prix d'écrire à la main la structure de chaque requête. La même page nomme un second coût : la tâche HTTP générique ne déclare pas de prise en charge du mode de vérification, si bien qu'une simulation du play de création de la machine en ignore toutes les tâches.

Où les problèmes surviennent

Un GET modifie quelque chose. Une adresse comme `/orders/7421/cancel`, qui annule une commande lorsqu'elle est consultée, rompt la promesse de sûreté de GET. La RFC 9110 explique cette promesse : les robots et les caches qui anticipent les lectures doivent pouvoir travailler sans crainte de provoquer des dommages. Le premier robot ou cache d'anticipation qui rencontre le lien annule la commande. Placez plutôt l'action dans la méthode, par un POST vers la commande ou un PATCH de son statut.

Chaque réponse est 200 OK. Certaines API renvoient 200 pour tout et placent le véritable résultat, erreur ou non, dans le corps. Les messages cessent d'être autodescriptifs : caches, proxys, supervision et bibliothèques clientes lisent tous le code de statut et voient tous un succès. La RFC 9457 définit un corps d'erreur standard, `application/problem+json`, avec des membres comme `type`, `title`, `status` et `detail`. Son membre `status` n'est qu'indicatif : le serveur doit envoyer le même code de statut dans la réponse HTTP elle-même, pour que les logiciels HTTP génériques ignorant ce format se comportent correctement.

Un POST répété crée deux commandes. Si la connexion est coupée après le traitement d'un POST par le serveur mais avant que le client ait lu la réponse, celui-ci ne peut pas savoir si la commande existe. La RFC 9110 indique qu'un client ne devrait pas répéter automatiquement une requête non idempotente, sauf s'il sait qu'elle est en réalité idempotente ou peut déterminer que la première tentative n'a jamais été appliquée. Un proxy ne doit pas le faire du tout. Elle relève aussi que certains clients prennent le chemin plus risqué et répètent malgré tout un POST lorsque la connexion se ferme avant l'arrivée du moindre morceau de réponse, en particulier sur une connexion persistante inactive. Une solution consiste à laisser le client choisir l'identifiant et envoyer un PUT de la commande à cette adresse, puisque PUT est idempotent : la première requête crée la commande avec 201 et une répétition ne fait que la remplacer par le même état.

Deux clients écrasent leurs modifications respectives. Tous deux lisent la commande, modifient chacun un champ différent puis renvoient le tout par PUT : la seconde écriture annule silencieusement la première. La RFC 9110 appelle cela le problème de mise à jour perdue. La correction est l'étiquette d'entité reçue par chaque client lors de la lecture, renvoyée dans `If-Match` . Le serveur n'applique alors le changement que si la ressource est encore dans l'état vu pour la dernière fois par ce client. Sinon, il ne doit pas l'appliquer et peut l'indiquer par 412 Precondition Failed. Le client relit alors la commande, réapplique son changement et réessaie.

```
PUT /orders/7421 HTTP/1.1
Host: api.example.com
If-Match: "v1"
Content-Type: application/json

{"item": "notebook", "quantity": 3, "status": "open"}

HTTP/1.1 412 Precondition Failed
```

Le serveur se souvient de la conversation. Un serveur qui conserve entre les requêtes un état par client, comme la page courante d'une liste ou un formulaire à moitié rempli gardé en mémoire, a abandonné la contrainte d'absence d'état. Une requête ne se comprend plus seule, le répartiteur de charge doit renvoyer chaque client vers la même machine et un redémarrage fait perdre cet état. Les raisons de Fielding pour l'absence d'état sont précisément ce qui est perdu : visibilité, récupération plus facile après une défaillance partielle et serveur pouvant libérer ses ressources dès sa réponse. Mettez l'état dans la requête,

sous forme de jeton de page dans l'URL, ou faites de l'élément inachevé une ressource propre, comme une commande brouillon lisible par n'importe quel serveur.

Pour aller plus loin

- Fielding, chapitre 5 : transfert d'état de représentation, REST — la définition : chaque contrainte, les propriétés qu'elle induit et ce qu'elle sacrifie en échange.
- RFC 9110 : sémantique HTTP — méthodes, sûreté et idempotence, codes de statut et requêtes conditionnelles avec `ETag` et `If-Match` .
- RFC 5789 : méthode PATCH pour HTTP — modification partielle d'une ressource et raison pour laquelle PATCH n'est ni sûre ni idempotente.
- RFC 9457 : détails des problèmes pour les API HTTP — corps JSON standard pour les réponses d'erreur, qui préserve le sens du code de statut.

Glossaire

Choisissez selon vos contraintes réelles

Présentez le processus de travail, les interfaces existantes et les utilisateurs qui ont besoin du changement. Nous pouvons définir une livraison testable et une transmission permettant à votre équipe d'assurer la maintenance des éléments livrés.

Consulter les tarifs Contactez-nous →

Rendez la terminologie concrète

Demandez comment « Qu'est-ce que REST ? » s'applique à votre organisation. Un exemple concret tiré de votre système actuel suffit pour engager la discussion.

Découvrir nos prestations

welcome@engineer.company +45 50 10 61 84 Copenhague, Danemark

Emportez cette page avec vous

Fiche de contact

Créé avec soin et amour par D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Droit d'auteur © 2025 – aujourd'hui · Engineer ApS · Tous droits réservés.

Imprimé depuis https://engineer.company/fr/glossary/rest/