Engineer

REST

🏛 the web's architectural style, built around resources

What it is

REST, short for Representational State Transfer, is the architectural style Roy Fielding set out in chapter 5 of his dissertation in 2000. It is not a protocol and not a standard. It is a set of constraints on how clients, servers and the intermediaries between them may interact, and Fielding states what each constraint buys and, for most of them, what it costs. RESTful is the adjective: strictly, it describes a system that keeps all of the constraints. In everyday use both words stretch much further, to almost any HTTP API that exchanges JSON, whether or not it keeps the constraints below.

The central idea is the resource. In Fielding's words, "Any information that can be named can be a resource": a document, an image, a collection of other resources, a person, even the current weather in one city. What crosses the network is never the resource itself but a representation, a sequence of bytes plus metadata that describes those bytes, such as an HTML document or a JPEG image, and a client acts on a resource by sending or receiving a representation of its current or intended state.

Fielding builds the style up from an empty set of constraints, adding one at a time. Client-server separates the user interface from data storage, so the two can evolve independently. Stateless means that every request carries everything needed to understand it and leans on no context stored on the server, so session state lives entirely with the client. Cache requires each response to be labelled, explicitly or implicitly, as cacheable or not, so that a client can reuse a cacheable one for a later, equivalent request. Uniform interface, described below, gives every component the same general interface. Layered system means each component sees only the layer it talks to directly, which leaves room for proxies, gateways and firewalls in between without changing the interfaces on either side. Code on demand lets a server extend a client by sending it scripts to run, and Fielding makes it the one optional constraint because it reduces visibility.

He calls the uniform interface the central feature that sets REST apart from other network-based styles, and splits it into four constraints: resources are identified, on the web by URLs; resources are manipulated through representations; messages are self-descriptive, so their meaning is visible to every intermediary; and hypermedia is the engine of application state, which means the client chooses its next transition from the representations it has received, the way a person follows links in a browser.

Use the concept in a decision

Put “What is REST?” to work in a real decision. We can help relate the concept to the system you are building or buying. The part users never see: where the data lives, what the rules are, and how other systems are allowed to talk to it.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Designed and ran 6 ETL/ELT pipelines — 6 ETL pipelines across three database engines

Designed and ran 6 ETL/ELT pipelines Get in touch →

Why it matters

Each constraint pays for something specific. Statelessness lets a monitoring system understand a request without looking beyond it, makes it easier to recover from partial failures, and lets a server free resources as soon as it has answered. A cache can remove some interactions altogether. Layers let intermediaries spread load across networks and machines, and let one cache at the edge of an organisation serve everyone inside it. And because the uniform interface decouples implementations from the services they provide, a client and a server can each change without the other having to change in step.

Fielding is just as plain about the costs. Statelessness repeats data in every request that a shared context on the server would have saved. A cache can hand out stale data. Every layer adds overhead and latency. The uniform interface degrades efficiency, because information travels in a standardised form rather than one shaped to the needs of a particular application. Fielding designed it for large-grain hypermedia data transfer, the common case on the web, and says plainly that it is not optimal for other kinds of interaction. That is the ground the alternatives stand on. A GraphQL server takes every request at a single endpoint, identifies its entities by something other than URLs and lets the client choose exactly which fields come back, while gRPC calls methods declared in a service definition. Each gives up part of the uniform interface to suit a different kind of traffic.

For an HTTP API, much of the uniform interface comes ready-made. HTTP already defines methods with agreed meanings, status codes, and header fields for caching and for conditional requests. Because that meaning is shared, a proxy or cache on the way can act on a message without knowing anything about the particular API, and an API that uses these parts as defined gets that machinery for free. An API that hands its clients URL templates to fill in, rather than links to follow, gives up the hypermedia constraint, and its clients are tied to the server's URL layout: when that layout changes, they break, which is the coupling the uniform interface exists to remove.

What it looks like in practice

The exchange below creates an order and is answered with the address of the new resource in `Location` and an entity tag in `ETag`. RFC 9110 uses an `ETag` in a 201 response as its own example: the tag can serve as a validator in later conditional requests, so that two clients cannot silently overwrite each other's changes.

```
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"}
```

Every method carries a meaning that any intermediary may rely on. RFC 9110 defines GET, HEAD, OPTIONS and TRACE as safe: the client neither asks for nor expects any change of state on the server. PUT and DELETE, together with the safe methods, are idempotent, so several identical requests have the same intended effect as one. POST is neither, and RFC 5789 says the same of PATCH. Those two properties are what let a crawler follow links without doing harm, let a cache fetch ahead, and let a client repeat a PUT whose connection dropped before the answer arrived.

A REST API can be driven with nothing more than a generic HTTP client, and this site shows both what that saves and what it costs. When a second server was provisioned from code, the firewalls at both cloud providers were configured straight through their REST APIs with a generic HTTP task instead of a vendor collection. That removes a dependency and its version drift, and the price is writing the shape of every request by hand. The same page names a second cost: the generic HTTP task declares no check-mode support, so a dry run of the play that creates the machine skips every task in it.

Where it goes wrong

A GET changes something. An address such as `/orders/7421/cancel` that cancels an order when it is fetched breaks the promise that GET is safe. RFC 9110 explains why that promise exists: so that crawlers and caches that fetch ahead can work without fear of causing harm. The first crawler or prefetching cache to come across the link cancels the order. Put the action in the method instead, as a POST to the order or a PATCH of its status.

Every answer is 200 OK. Some APIs return 200 for everything and put the real outcome, error or not, in the body. The messages stop being self-descriptive: caches, proxies, monitoring and client libraries all read the status code, and all of them see success. RFC 9457 defines a standard body for errors, `application/problem+json`, with members such as `type`, `title`, `status` and `detail`. Its `status` member is only advisory: the server must send the same status code in the HTTP response itself, so that generic HTTP software which knows nothing of the format behaves correctly.

A retried POST creates two orders. When a connection drops after the server has processed a POST but before the client has read the answer, the client cannot tell whether the order exists. RFC 9110 says a client should not retry a non-idempotent request automatically unless it knows that the request is idempotent after all, or can tell that the first attempt was never applied, and that a proxy must not do so at all. It also notes that some clients take the riskier path and retry a POST anyway when the connection closes before any of the answer arrives, particularly on an idle persistent connection. One way out is to let the client choose the identifier and PUT the order to it, because PUT is idempotent: the first request creates the order with 201, and a repeat only replaces it with the same state.

Two clients overwrite each other. Both read the order, both change a different field, and both PUT the whole thing back: the second write silently undoes the first. RFC 9110 calls this the lost update problem. The fix is the entity tag each client got when it read the order, sent back in `If-Match`. The server then applies the change only if the resource is still in the state that client last saw; if it is not, the server must not apply it and can say so with 412 Precondition Failed. The client then reads the order again, reapplies its change and tries once more.

```
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
```

The server remembers the conversation. A server that keeps per-client state between requests, such as the current page of a listing or a half-finished form held in memory, has dropped the stateless constraint. A request no longer makes sense on its own, the load balancer has to send each client back to the same machine, and a restart throws the state away. Fielding's reasons for statelessness are exactly the ones lost: visibility, easier recovery from partial failure, and a server that can free resources once it has answered. Put the state in the request, as a page token in the URL, or turn the half-finished thing into a resource of its own, such as a draft order, that any server can read.

Further reading

- Fielding, chapter 5: Representational State Transfer (REST) — the definition: each constraint, the properties it induces and what it trades away for them.
- RFC 9110: HTTP Semantics — methods, safety and idempotency, status codes, and conditional requests with `ETag` and `If-Match`.
- RFC 5789: PATCH Method for HTTP — partial modification of a resource, and why PATCH is neither safe nor idempotent.
- RFC 9457: Problem Details for HTTP APIs — a standard JSON body for error responses that keeps the status code meaningful.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is REST?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

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

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/rest/