Engineer

Idempotent

safe to run twice, because the second time changes nothing

What it is

An operation is idempotent when performing it several times has the same effect as performing it once, and the property is called idempotence or idempotency. The word comes from mathematics, where a function is idempotent when applying it to its own result changes nothing, so that f(f(x)) equals f(x): rounding down a number that has already been rounded down gives the same number.

In software the useful test is a second run. Setting a value is idempotent, because the second run writes the value that is already there. Adding to a value is not, because the second run adds again:

```
-- idempotent: a second run sets the same value
UPDATE accounts SET plan = 'pro' WHERE id = 42;

-- not idempotent: a second run adds ten more
UPDATE accounts SET credits = credits + 10 WHERE id = 42;
```

HTTP gives the word its most precise definition. RFC 9110 calls a request method idempotent when several identical requests are intended to have the same effect on the server as a single one, and it lists PUT, DELETE and the safe methods GET, HEAD, OPTIONS and TRACE. Two details in that definition matter. The property covers the effect the client asked for, so a server is free to log every request separately. And the response may differ: a second DELETE of the same resource can answer 404 where the first answered 200, and the method is still idempotent, because the resource is gone either way. PATCH sits outside the list. RFC 5789 says it is neither safe nor idempotent, although a client can send a

particular PATCH in an idempotent way by making it conditional, with a strong ETag in an If-Match header, so that a repeated copy is refused once the first has changed the resource.

Safe is the stricter of the two promises. A safe method in RFC 9110's sense is essentially read-only, so every safe method is idempotent, while DELETE is idempotent without being safe.

Use the concept in a decision

Put "What does idempotent mean?" to work in a real decision. We can help relate the concept to the system you are building or buying. Numbers arriving where they are needed, on a schedule, without anyone copying a spreadsheet at the end of the month.

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. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

Networks lose answers. When a client sends a request and the connection drops before the response arrives, it cannot tell whether the server never saw the request, saw it and stopped, or carried it out and lost the reply. Repeating the request is the obvious recovery, and idempotence is what makes the repetition harmless. RFC 9110 builds its retry rules on exactly that. An idempotent request can be repeated automatically after a communication failure. A client should not retry a non-idempotent request automatically unless it knows the request is idempotent in practice or can tell that the original was never applied, and a proxy must never retry one on its own.

Queues raise the same question at scale. Many message systems promise delivery at least once: a consumer that crashes after doing its work and before acknowledging the message is handed the same message again. Delivery exactly once across a network is a far harder promise, so the practical answer is a consumer for which a second delivery changes nothing. Configuration management depends on the property too. A tool that brings servers to a declared state runs again and again, after every change and often on a schedule, and each run has to leave a correct server exactly as it found it.

The real decision is how to get idempotence where an operation lacks it by nature, and there are three common answers. The first is design: express the change as a target state rather than a step, so “make sure this line is in the file” instead of “append this line”. The second is a key: the client attaches a unique identifier to the request, the server records it under a unique constraint, and a repeat finds the record and gets the original result back instead of a second action. The third is to check before acting, which is how most Ansible modules work: they compare the current state with the desired one and exit without acting when the two already match. Each has a cost. Target-state design needs a clear model of the end state, keys need storage and a rule for how long they are kept, and checking first needs a reliable way to read the current state.

What it looks like in practice

The company’s own servers are configured with Ansible, and on that project idempotence is a contract: a converged host reports zero changes. Provisioning with no arguments converges every host to its declared state, and a play that cannot report zero changes on a host that is already correct counts as unfinished. That makes idempotence something every run measures rather than something one hopes for.

The same idea, applied to data, is behind a test that runs the nineteen-step build pipeline twice. The page names the kind of bug it exists for: a generator that works the first time and corrupts the second, through steps that insert without checking or depend on the order of an earlier step’s output. The test builds the database from an empty file, takes a snapshot, runs the whole pipeline again over the result and requires the second pass to change nothing, down to row counts, contents and identifiers. The test provides repeat-run evidence for the inputs and database state it exercises.

For selling limited stock, the note on selling the last ticket exactly once combines a key with conditional writes. Every checkout carries an idempotency key, and the holds table carries `UNIQUE (tenant_id, idempotency_key)`, so a retried click gets back the hold it already has instead of a second unit. A release that runs twice, from a timeout and a cancel button for instance, gives the unit back once, because it credits the stock only when its own update changed exactly one row. Further along, every message consumer is idempotent, keyed by the order: messages arrive at least once, and idempotency makes that effectively once.

Where it goes wrong

A retried request charges twice. A POST that creates a payment, an order or an email is not idempotent by default, and a client or proxy that retries it after a timeout creates a second one. Give these requests an idempotency key that the server stores under a unique constraint, and return the stored result when the same key comes back.

An empty result is read as an error. In PostgreSQL, `INSERT ... ON CONFLICT DO NOTHING` skips a row that would violate a unique constraint, and `RETURNING` gives back only rows that were actually inserted or updated. A repeated insert of the same row therefore returns nothing, which is the correct outcome of a second run. Code that treats the empty result as a failure turns a harmless retry into an error page; it should look up the row that is already there.

A command task changes the host on every run. Ansible's documentation says most modules check whether the desired state has already been reached, and also that "not all playbooks and not all modules behave this way". A task that runs a command is the usual exception, because the command is arbitrary and Ansible cannot know what it does. The command module's `creates` option skips the step when a given file already exists, `removes` runs it only while the file is still there, and `changed_when` defines which outcome counts as a change. A command task with none of them runs again on every converge, whatever state the host is in.

A counter counts the retry. Under delivery at least once, a consumer that increments a counter or sends a notification for each message does it twice for a message delivered twice. Store the message's business key alongside the effect, in the same transaction, and skip any key that is already stored.

The second run corrupts what the first one built. A step can be correct on an empty database and wrong on a full one, inserting a duplicate, renumbering identifiers or relying on an order the first run left behind. A test that starts from nothing and runs once will pass all the same. Run the whole job twice and require the second run to change nothing, as the pipeline test above does.

Further reading

- RFC 9110, section 9.2.2: Idempotent Methods — HTTP's definition, the idempotent methods and the rules for automatic retries.

- RFC 5789: PATCH Method for HTTP — why PATCH is neither safe nor idempotent, and how a conditional request makes one safe to repeat.
- Ansible: Introduction to playbooks — desired state, idempotency and check mode in Ansible's own words.
- Ansible: command module — the `creates` and `removes` options that make a command safe to rerun.
- PostgreSQL: INSERT — `ON CONFLICT` and what `RETURNING` gives back when a row already exists.

Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how "What does idempotent mean?" 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/idempotent/