Engineer

Backend

the half of a system the reader never loads

What it is

A backend is the part of a software system that runs on machines its operator controls: it stores the data, applies the rules about who may do what, and answers the requests that arrive from browsers, apps and other services. It is also written back end or back-end, and the same territory is often called the server side.

In HTTP terms, a backend is what RFC 9110 calls an origin server, “a program that can originate authoritative responses for a given target resource”. The W3C’s Architecture of the World Wide Web adds the reader’s view: a URI identifies a resource, and what travels to the browser is a representation of it, data that encodes information about the resource’s state. A customer sees an order page; the backend holds the order, the stock and the payment, and decides what goes into the reply.

A backend usually exposes an API, a set of HTTP endpoints that a frontend, a mobile app or another company’s system can call. The API is a contract: which paths exist, what each one accepts, what it returns and which errors it can produce. The OpenAPI Specification is the common way to write that contract down, as a programming language-agnostic description from which tools can generate clients, documentation and tests. Behind the API, the work is usually split into layers: request handling, business logic and storage.

Use the concept in a decision

Put “What is a backend?” 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

The first question is whether a system needs a backend running at request time at all. A page that is the same for every visitor can be generated once and served as files. A running backend earns its place when a request has to change something or depends on who sent it: an order, a sign-in, a search over private data. It also carries a standing cost, since software that answers requests from the internet has to be secured, patched and paid for.

The second is trust. Because the backend runs on machines its operator controls, it is the one place a rule can be enforced. Anything the frontend checks can be skipped by someone who sends requests directly, so every rule that matters, from a price to a permission, is checked on the server against the signed-in user and the specific record. The OWASP Application Security Verification Standard turns this into requirements a team can build to and test against.

The third is where the logic lives. A rule spread across a handler, a service class and a database trigger has three places to change and three copies to drift apart. Some teams keep the rules in application code and use the database as storage; others push rules and the shaping of data into the database and keep the API thin. Either can work, as long as each rule has one home.

The fourth is the contract with clients. HTTP gives every response a status code with a defined meaning, and clients, proxies and monitoring tools act on it. A 422 Unprocessable Content says the request was syntactically correct and in a supported format, but semantically wrong, such as a booking dated in the past. RFC 9457 adds a standard body for errors, called problem details: a small JSON object with a type, a title, a status, a detail and an instance, sent as `application/problem+json`, so a client can tell one failure from another without parsing prose.

What it looks like in practice

A relay race of transformations is the usual route from database to browser: the API unmarshals rows into structs, reshapes them and serialises them back to JSON. On one platform, the backend lets the database build each response and

passes it on untouched. PostgreSQL functions assemble the whole response as JSON, where the data already is, and the Go handler takes it as `json.RawMessage` and forwards it untouched. A small `QueryJSON` helper made that the path of least resistance. The DTOs, the mappers and the response structs fell away, handler code got dramatically shorter, and a change to what a function returns flows straight through to the client without anyone editing handler code.

Requests coming in get the same one-source treatment, through a contract that starts at the database and runs outward. On the backend side, the schema and its functions define the shapes and the Go types define the API. Huma emits the OpenAPI description from those types, and the frontend's client, 44,076 lines of typed TypeScript, is generated from that description. The limits the user interface enforces on a form come from the same place instead of being retyped into a validator. Every hop has a guard, so a field cannot change on one side only: the mismatch fails in a build, minutes after the change. Regenerating is the chore this costs, since every change to a shape the backend defines has to be carried down the chain by generating each link after it again.

At the recorded design stage, this site made the opposite choice. It was generated entirely at build time and needed no request-time application backend. Its four requirements, including working in three languages and avoiding a separately billed application service, led to that choice: a runtime application backend would need securing, patching and explanation on a privacy page. Beyond the recorded scoped JavaScript exception for a portfolio filter, interactive features had to be solved at build time or omitted. This describes the recorded design stage, not the current site's inventory or implementation; it does not mean hosting had no cost.

Where it goes wrong

Backend faults tend to stay quiet until a request arrives that nobody planned for.

A user changes a record that belongs to someone else. The endpoint checks that the caller is signed in and then loads whatever ID the URL names, so editing `/invoices/1041` to `/invoices/1042` in a request is all it takes. Check ownership or permission on the server for every object a request touches, in one place every handler passes through, and keep a test that asks for another account's record and expects a 403 or a 404.

An action happens twice. A link that cancels something through a GET request is followed by a crawler or a browser that prefetches pages, and the action runs without anyone clicking. RFC 9110 defines GET, HEAD, OPTIONS and TRACE as safe so that spiders and pre-fetching can work without fear of causing harm. The other route is a retry: a payment request times out, the client sends it again, and the customer pays twice. Put every change of state behind POST, PUT, PATCH or DELETE, and give writes that may be retried a key the server remembers, so the second copy returns the result of the first.

Every error comes back as a 500, or as a 200 with an error message inside. One catch-all handler reports every failure the same way, so monitoring counts typing mistakes as outages and retry logic repeats requests that will fail every time. Map each failure to the status code that describes it: 400 or 422 for bad input, 401 and 403 for identity and permission, 404 for a missing resource, 409 for a conflict, and 5xx only for the server's own faults. Send the details as problem details:

```
{
    "type": "https://example.com/problems/booking-in-past",
    "title": "Booking date is in the past",
    "status": 422,
    "detail": "The date 2026-10-01 is before today.",
    "instance": "/bookings/8812"
}
```

RFC 9457 says problem details "are not a debugging tool for the underlying implementation", so stack traces and SQL stay in the server's logs.

Pages slow down as the data grows. A handler loads a list of orders and then queries the customer for each one, so a page of 50 orders makes 51 round trips to the database. It feels quick in development with ten rows and drags in production with ten thousand. Fetch related data in one query with a join, or let the database assemble the whole response, and watch the number of queries per request as closely as the response time.

Further reading

- Architecture of the World Wide Web, Volume One — the W3C Recommendation on resources, representations, and safe and unsafe interactions.

- RFC 9110: HTTP Semantics — the IETF standard for methods, status codes and what makes a request safe or idempotent.
- OpenAPI Specification — the language-agnostic format for describing an HTTP API, read by people and by code generators.
- RFC 9457: Problem Details for HTTP APIs — the standard JSON shape for machine-readable error responses.
- OWASP Application Security Verification Standard — a list of security requirements for building and testing web applications.

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 a backend?” 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/backend/