**Engineer**

# gRPC

Framework für entfernte Prozeduraufrufe über HTTP/2

## Was es ist

gRPC ist ein Framework für entfernte Prozeduraufrufe. Ein Programm ruft eine Methode eines Servers auf einem anderen Rechner auf, als wäre der Server ein lokales Objekt. gRPC überträgt den Aufruf in einem eigenen Protokoll über das Netzwerk, aufbauend auf HTTP/2, und liefert die Antwort als Rückgabewert zurück.

Alles beginnt mit einer Dienstdefinition: einer Liste entfernter Operationen und der Anfrage- und Antworttypen, die jede davon annimmt. Standardmäßig wird sie in Protocol Buffers geschrieben, Googles Open-Source-Format zur sprach- und plattformunabhängigen Serialisierung von Daten, in einer Datei mit der Endung `.proto`. Andere Formate lassen sich einbinden; ohne eine entsprechende Vorgabe verwendet gRPC jedoch Protocol Buffers. Der Compiler `protoc` liest die Datei zusammen mit einem gRPC-Plugin und erzeugt Code für beide Seiten. Der Client erhält einen Stub, ein lokales Objekt mit denselben Methoden wie der Dienst; manche Sprachen nennen es einfach den Client. Der Server erhält eine Schnittstelle, die er implementieren muss. Weil beide Seiten aus einer Datei entstehen, kann ein in Java geschriebener Server Anfragen von Clients in Go, Python oder Ruby beantworten.

Eine Methode hat eine von vier Formen. Eine unäre Methode nimmt eine Anfrage entgegen und gibt eine Antwort zurück, wie ein gewöhnlicher Funktionsaufruf. Eine Server-Streaming-Methode nimmt eine Anfrage entgegen und sendet einen Strom von Antworten zurück; eine Client-Streaming-Methode macht es umgekehrt. Bei einer bidirektionalen Streaming-Methode können beide Seiten ihre Datenströme unabhängig voneinander senden. Innerhalb eines einzelnen Aufrufs garantiert gRPC, dass Nachrichten in der Reihenfolge eintreffen, in der sie gesendet wurden.

Bei der Übertragung läuft gRPC über HTTP/2. Jeder Aufruf ist ein `POST` an einen Pfad aus dem vollständig qualifizierten Dienstnamen und dem Methodennamen, mit dem Inhaltstyp `application/grpc`. Vor jeder Nachricht steht ein Präfix von fünf Bytes: ein Byte gibt an, ob die Nachricht komprimiert ist, vier weitere ihre Länge. Das Ergebnis des Aufrufs ist nicht der HTTP-Status. Eine von gRPC erzeugte Antwort trägt auch dann den Status 200, wenn der Aufruf scheitert. Das tatsächliche Ergebnis steht in Trailern, also nach dem Nachrichtentext gesendeten Headerfeldern: `grpc-status` mit einer Zahl und optional `grpc-message` mit einer Fehlerbeschreibung.

Damit unterscheidet es sich grundlegend von REST. Eine REST-API benennt Ressourcen und bearbeitet sie mit den Methoden und Statuscodes von HTTP, meist mit JSON-Nachrichtentexten, die jeder lesen kann. Eine gRPC-API ist eine Liste von Methoden mit typisierten binären Nachrichten; HTTP/2 ist lediglich die darunterliegende Transportschicht. Die Dokumentation von Protocol Buffers vergleicht das Format mit JSON und hebt kleinere Nachrichten und schnellere Verarbeitung hervor.

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist gRPC?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Der Teil, den Nutzer nie sehen: wo die Daten liegen, welche Regeln gelten und wie andere Systeme darauf zugreifen dürfen.

**Unsere Leistungen entdecken** Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. 6 ETL/ELT-Pipelines entworfen und betrieben — 6 ETL-Pipelines über drei Datenbankengines hinweg

**6 ETL/ELT-Pipelines entworfen und betrieben** Kontakt aufnehmen →

## Warum es wichtig ist

Der Vertrag kommt zuerst, der Code folgt daraus. Bei vielen HTTP-APIs sind die Beschreibung und der Code, der sie bedient, getrennte Dinge, die auseinanderlaufen können. Ein Client wird häufig anhand eines Dokuments von Hand geschrieben. Bei gRPC ist die `.proto`-Datei die einzige Stelle, an der eine Methode oder ein Feld definiert wird, und jeder Client und Server wird daraus generiert. Eine Änderung an einer Nachricht erreicht jede Sprache über denselben Generator, nicht dadurch, dass jemand erneut ein Dokument liest.

Die Kodierung ist kompakt, weil jedes Feld mit seiner Nummer und seinem Wert statt mit seinem Namen übertragen wird. Der Preis dafür: Ohne die zugehörige `.proto`-Datei lässt sich eine Nachricht nicht vollständig interpretieren, denn

Protocol Buffers beschreiben sich nicht selbst. Außerdem setzt das Format voraus, dass eine ganze Nachricht auf einmal in den Arbeitsspeicher passt. Seine eigene Dokumentation empfiehlt für Daten oberhalb einiger Megabytes eine andere Lösung.

Streaming gehört zum Entwurf und ist keine nachträgliche Ergänzung. Aufrufe laufen als Streams auf HTTP/2-Verbindungen. Eine Server-Streaming-Methode kann daher über dieselben Mechanismen fortlaufend Ergebnisse senden wie ein Aufruf, der nur einmal antwortet.

Fristen werden mit dem Aufruf weitergegeben. Ein Client setzt eine Frist, gRPC überträgt die verbleibende Zeit im Header `grpc-timeout`, und wenn sie abläuft, erhält der Client `DEADLINE_EXCEEDED`; die Serverseite des Aufrufs wird abgebrochen. Ruft ein Server seinerseits einen weiteren Server auf, geben manche Implementierungen die Frist weiter: Java und Go tun dies standardmäßig, in C++ muss es eingeschaltet werden. Weil die Frist bei jedem Zwischenschritt als verbleibende Dauer statt als Uhrzeit übertragen wird, sind sich auch Rechner mit voneinander abweichenden Uhren einig, wann sie aufgeben müssen.

Für Fehler gilt ein gemeinsames Vokabular. gRPC definiert 17 Statuscodes, von 0 `OK` bis 16 `UNAUTHENTICATED`. Sie gehören zum Protokoll selbst, sodass ein Go-Client und ein Java-Server dasselbe unter `NOT_FOUND` oder `UNAVAILABLE` verstehen.

All das eignet sich für den Verkehr zwischen Diensten, die ein Team oder ein Unternehmen kontrolliert und deren beide Seiten die `.proto`-Dateien gemeinsam nutzen und daraus bauen können. Für den Browser eignet es sich weniger gut. Die eigenen Protokollhinweise des gRPC-Projekts erklären, dass Browser Skripten keinen Zugriff auf das HTTP/2-Framing geben und dies auch niemals tun werden. Browsercode spricht deshalb ein gesondertes Protokoll, gRPC-Web. Im üblichen Aufbau übersetzt ein Proxy es in natives gRPC.

## Wie es in der Praxis aussieht

Hier ist ein kleiner Abrechnungsdienst. Eine unäre Methode ruft eine Rechnung ab; eine Server-Streaming-Methode sendet die Rechnungen eines Kunden einzeln:

```
syntax = "proto3";

package billing.v1;

service InvoiceService {
  rpc GetInvoice(GetInvoiceRequest) returns (Invoice);
  rpc WatchInvoices(WatchInvoicesRequest) returns (stream Invoice);
}

message GetInvoiceRequest {
  string invoice_id = 1;
}

message WatchInvoicesRequest {
  string customer_id = 1;
}

message Invoice {
  reserved 4;
  reserved "discount_code";
  string invoice_id = 1;
  string customer_id = 2;
  int64 total_cents = 3;
  string currency = 5;
}
```

Die Zahlen sind keine Standardwerte. Es sind Feldnummern, die bei der Übertragung jedes Feld anstelle seines Namens kennzeichnen. Nummern von 1 bis 15 benötigen zur Kodierung ein Byte und sollten deshalb den am häufigsten gesetzten Feldern gehören. Die beiden `reserved` -Zeilen halten ein früher vorhandenes Feld fest, Nummer 4 mit dem Namen `discount_code` . So kann weder die Nummer noch der Name für ein anderes Feld wiederverwendet werden.

Für Python erzeugt ein Befehl `billing_pb2.py` mit den Nachrichtenklassen und `billing_pb2_grpc.py` mit den Client- und Serverklassen:

```
python -m grpc_tools.protoc -Iprotos \
  --python_out=. --grpc_python_out=. protos/billing.proto
```

Ein Aufruf von `GetInvoice` sieht als HTTP/2-Frames dann so aus:

```
HEADERS (flags = END_HEADERS)
:method = POST
:scheme = https
:path = /billing.v1.InvoiceService/GetInvoice
:authority = billing.example.com
te = trailers
content-type = application/grpc+proto
grpc-timeout = 2S

DATA (flags = END_STREAM)
<Length-Prefixed Message>

HEADERS (flags = END_HEADERS)
:status = 200
content-type = application/grpc+proto

DATA
<Length-Prefixed Message>

HEADERS (flags = END_STREAM, END_HEADERS)
grpc-status = 0
```

`2S` bedeutet zwei Sekunden. `te = trailers` dient dazu, Proxys zu erkennen, die mit gRPC nicht kompatibel sind. Existiert die Rechnung nicht, beginnt die Antwort trotzdem mit `:status = 200`. Der letzte Frame enthält dann `grpc-status = 5`, also `NOT_FOUND`, zusammen mit einer `grpc-message` wie `invoice not found`. Ein Server, der sofort scheitert, kann sogar nur einen einzelnen `HEADERS`-Frame mit dem Status und ganz ohne Nachricht senden.

Beide Seiten aus einem Vertrag zu erzeugen, ist keine Besonderheit von gRPC. Ein Projekt im Portfolio dieser Website generierte den API-Vertrag von der Datenbank nach außen für eine mit OpenAPI beschriebene API. Die Kette beginnt in der Datenbank und führt nach außen: Go-Typen definieren die API, Huma erzeugt daraus die OpenAPI-Beschreibung, aus dieser werden ein typisierter TypeScript-Client und Mock-Handler generiert. Eine Vertragsprüfung in CI lässt den Build scheitern, sobald das, was das Frontend sendet, nicht mehr zu dem passt, was die API erwartet. gRPC baut dieselbe Disziplin in seine Werkzeugkette ein, mit der `.proto`-Datei als einziger Quelle.

## Wo es schiefgeht

**Ein Aufruf ohne Frist kann ewig warten.** gRPC setzt standardmäßig keine Frist. Ein Client, der einen hängenden Server aufruft, kann deshalb praktisch unbegrenzt auf eine Antwort warten. Geben Sie jedem Aufruf eine realistische

Frist. In Python ist das das Argument `timeout`, in Sekunden:

```
import grpc

import billing_pb2
import billing_pb2_grpc

credentials = grpc.ssl_channel_credentials()
with grpc.secure_channel("billing.example.com:443", credentials) as channel:
    stub = billing_pb2_grpc.InvoiceServiceStub(channel)
    request = billing_pb2.GetInvoiceRequest(invoice_id="INV-1001")
    invoice = stub.GetInvoice(request, timeout=2.0)
```

Nach Ablauf der Zeit löst der Aufruf `grpc.RpcError` aus. Dessen `code()` gibt `grpc.StatusCode.DEADLINE_EXCEEDED` zurück. Auf dem Server bricht gRPC den Aufruf nach Ablauf der Frist mit `CANCELLED` ab. Bereits vom Handler begonnene Arbeit anzuhalten, ist jedoch Sache des eigenen Dienstcodes: Ein lange laufender Handler muss selbst auf den Abbruch prüfen und aufhören.

**Ein Timeout bedeutet nicht, dass nichts geschehen ist.** Client und Server entscheiden jeweils selbst, ob ein Aufruf erfolgreich war. Er kann daher auf dem Server abgeschlossen sein und beim Client trotzdem mit `DEADLINE_EXCEEDED` scheitern. Ein Abbruch macht bereits erledigte Arbeit nicht rückgängig. Ein Client, der einfach einen neuen Versuch startet, kann dieselbe Rechnung zweimal anlegen. Das Protokoll geht nur dann davon aus, dass ein Aufruf idempotent ist, wenn er entsprechend gekennzeichnet ist. `UNAVAILABLE` ist der Code, bei dem ein vorübergehender Fehler und damit ein erneuter Versuch mit Backoff am ehesten infrage kommen. Die Dokumentation der Statuscodes warnt jedoch, dass Wiederholungen bei nicht idempotenten Operationen nicht immer sicher sind. Dasselbe Dokument überlässt dem aufrufenden Code die Entscheidung, bei welchen Codes er es erneut versucht.

**Eine wiederverwendete Feldnummer verfälscht Daten.** Sobald ein Nachrichtentyp verwendet wird, darf sich eine Feldnummer nicht mehr ändern. Wird ein Feld gelöscht und seine Nummer später einem neuen Feld gegeben, wird bei älteren Clients und gespeicherten Nachrichten das neue Feld aus Bytes dekodiert, die zuvor etwas anderes bedeuteten. Der Leitfaden zu Protocol Buffers nennt mögliche Folgen: Parse-Fehler, offengelegte personenbezogene Daten und beschädigte Daten. Nummerieren Sie niemals ein Feld um und reservieren Sie jede gelöschte Nummer und jeden gelöschten Namen wie im Beispiel oben.

**Jeder Fehler sieht für HTTP wie ein Erfolg aus.** Weil gRPC auch bei gescheiterten Aufrufen mit HTTP-Status 200 antwortet, zählt alles, was Verkehr nur anhand des HTTP-Status beurteilt, Fehler als Erfolge: etwa ein Zugriffsprotokoll oder eine aus Statuscodes berechnete Fehlerrate. Genau deshalb lässt die Spezifikation Server auf Anfragen, deren Inhaltstyp nicht `application/grpc` ist, mit 415 antworten. Andere HTTP/2-Clients sollen einen gRPC-Fehler nicht als Erfolg lesen. Zählen Sie `grpc-status`, nicht `:status`.

**Lange Streams binden ihre Last an einen Server.** Aufrufe laufen als Streams auf einer gemeinsam genutzten HTTP/2-Verbindung. Sobald ein Stream begonnen hat, kann er nicht auf einen anderen Server verschoben werden. Ein stundenlang laufender Stream behält seine Last dort, wo er begann, unabhängig davon, wie viele Server später hinzukommen. Jede Verbindung hat außerdem eine Obergrenze für gleichzeitige Streams. Ist sie erreicht, warten weitere Aufrufe im Client, bis ein Stream endet. Der Leistungsleitfaden von gRPC empfiehlt Streaming nur dort, wo es Leistung oder Einfachheit erheblich verbessert, und einen eigenen Channel für jeden Bereich mit hoher Last.

**Große Nutzdaten passen nicht in eine Nachricht.** gRPC erzwingt eine maximale Nachrichtengröße. Eine Nachricht oberhalb des konfigurierten Limits scheitert auf der jeweiligen Sender- oder Empfängerseite mit `RESOURCE_EXHAUSTED`. Das Limit anzuheben verschiebt das Problem nur: Protocol Buffers erwarten die ganze Nachricht im Arbeitsspeicher, und bei großen Daten können dort mehrere serialisierte Kopien gleichzeitig liegen. Teilen Sie große Daten in einen Strom kleinerer Nachrichten auf oder übertragen Sie sie außerhalb von gRPC.

## Weiterführende Lektüre

- Grundkonzepte, Architektur und Lebenszyklus — Dienstdefinitionen, die vier Methodenarten, Stubs, Fristen und Abbruch.
- gRPC über HTTP2 — das Übertragungsprotokoll: Pfade, Header, Nachrichtenpräfix und Status in Trailern.
- Statuscodes — alle 17 Codes, ihre Bedeutung und die Codes, die die Bibliothek selbst nie erzeugt.
- Fristen — warum es keine Standardfrist gibt, wie Fristen den Server erreichen und wie sie zwischen Diensten weitergegeben werden.
- Sprachleitfaden (proto 3) — Feldnummern, reservierte Felder und Regeln zum sicheren Ändern eines Nachrichtentyps.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

**Preise ansehen** Kontakt aufnehmen →

## Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist gRPC?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

**Unsere Leistungen entdecken**

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

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

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/grpc/