PROGRAMMING

gRPC

Protocol Buffers & Code Generation

The Four Streaming Modes · Interceptors & Security

Microservices & Load Balancing · Capstone Project

Philip Osztromok

Generated with Claude

Table of Contents

gRPC — 9 Chapters

🔌 What Is gRPC? RPC Meets HTTP/2

`api-design1-6` introduced the RPC model and JSON-RPC, then named gRPC as where that road leads next without going deep. This course picks up exactly there — starting with what gRPC actually is, why Google built it, and where it genuinely fits against REST and GraphQL.

The RPC Model, Recap

`api-design1-6` drew the core distinction: REST thinks in terms of **resources** ("fetch this thing"), while RPC thinks in terms of **function calls** ("run this specific operation"). A JSON-RPC call like `{"method": "getUser", "params": {"id": 104}}` is calling a named function remotely, not fetching a URL representing a resource. gRPC is exactly that model, made concrete with a real schema, real code generation, and a purpose-built transport.

Why Google Built gRPC

gRPC grew out of an internal RPC system Google had used for years to connect its own services at enormous scale, open-sourced in 2015. It was designed first and foremost for **service-to-service** communication inside a company's own infrastructure — the traffic between a user service and an order service, for instance — not primarily for public-facing APIs a browser calls directly. That origin explains nearly every design choice covered in this chapter.

HTTP/2 as the Transport

Most REST APIs run over HTTP/1.1 with JSON bodies. gRPC runs over **HTTP/2**, and leans on two of its capabilities directly:

- **Multiplexing** — many requests and responses can flow concurrently over a *single* connection, without one slow request blocking others behind it (the "head-of-line blocking" problem HTTP/1.1 suffers from when reusing a connection).
- **Binary framing** — messages are encoded as compact **Protocol Buffers** binary (Chapter 2), not human-readable JSON — smaller payloads, faster to parse, at the cost of no longer being readable by just glancing at network traffic.

REST's Typical Stack vs. gRPC's Stack

> **REST**
>
> HTTP/1.1 (often), JSON bodies — human-readable, universally supported, easy to debug with `curl` or a browser.

> **gRPC**
>
> HTTP/2, Protocol Buffers binary — smaller and faster, but requires generated code and dedicated tooling to inspect.

gRPC vs. REST vs. GraphQL Positioning

`api-design1-8` built a four-question decision framework across REST, SOAP, RPC-style, and webhooks — gRPC and GraphQL are the two remaining pieces of the full picture.

	REST	GraphQL	gRPC
Model	Resources	Client-specified query shape	Function calls
Typical transport	HTTP/1.1, JSON	HTTP/1.1, JSON	HTTP/2, Protobuf binary
Contract	Often implicit/documented separately	A typed schema	A typed schema (`.proto`), code-generated
Best fit	Public APIs, broad client compatibility	Flexible client data needs	Internal service-to-service, high performance

A First Glimpse: A .proto File

Chapter 2 covers the syntax properly — this is just a preview of what a gRPC schema-first contract looks like:

```
service UserService {
  rpc GetUser(GetUserRequest) returns (User);
}

message GetUserRequest {
  int32 id = 1;
}

message User {
  int32 id = 1;
  string name = 2;
}
```

gRPC Tends to Fit

Internal microservice-to-microservice calls where both ends are code you control, high-throughput or low-latency requirements, and genuine streaming needs (Chapter 4).

REST/GraphQL Still Win

Public-facing APIs consumed by browsers or arbitrary third parties, situations needing easy debugging with just `curl`, and clients that can't run generated code.

💻 Coding Challenges

Challenge 1: RPC vs. REST, Revisited

Using `api-design1-6`'s RPC-vs-REST distinction, explain in your own words why "call GetUser" and "GET /users/104" represent two different mental models, even though both might return the same data.

Goal: Practice restating the function-call vs. resource distinction in your own words before layering gRPC specifics on top.

→ Solution

Challenge 2: Explain Multiplexing's Benefit

Explain why HTTP/2 multiplexing matters for a gRPC client making many concurrent calls to the same service, compared to HTTP/1.1's limitations.

Goal: Practice connecting a specific HTTP/2 feature to a concrete performance benefit.

→ Solution

Challenge 3: Pick the Right Style

For each, recommend REST, GraphQL, or gRPC and justify: (a) a public API for third-party developers to integrate with, (b) high-frequency internal calls between an order service and an inventory service, (c) a mobile app that needs to fetch a customizable set of fields per screen.

Goal: Practice applying this chapter's positioning table to concrete, realistic scenarios.

→ Solution

⚠️ Gotcha: gRPC Doesn't Run Natively in Browsers

A browser can't perform the low-level HTTP/2 framing and trailer handling gRPC needs directly — calling a gRPC service straight from browser JavaScript isn't possible without an intermediary (gRPC-Web, requiring a proxy that translates browser-compatible requests into real gRPC calls). This is a direct consequence of gRPC's service-to-service origins: it was never designed with "a browser calls this directly" as a primary use case, which is exactly why Chapter 8's public-vs-internal framing matters — reaching for gRPC for a public API a browser calls directly adds real friction REST or GraphQL simply don't have.

🎯 What's Next

The next chapter gets concrete: **Protocol Buffers: Defining Your Schema** — `.proto` file syntax, message types, field numbering rules, and why a schema-first contract beats REST's typically implicit ones.

📐 Protocol Buffers: Defining Your Schema

Chapter 1 showed a `.proto` file glimpse without explaining it. This chapter covers the real syntax — messages, fields, numbering rules — and why this schema-first contract is a genuine structural advantage over how most REST APIs document themselves.

The .proto File

Every gRPC schema starts with a syntax declaration and, optionally, a package name to avoid naming collisions across files.

```
syntax = "proto3";
package shop;
```

Message Types & Scalar Fields

A `message` defines a structured data type — each field has a **type**, a **name**, and a **number**.

```
message Product {
  int32 id = 1;
  string name = 2;
  double price = 3;
  bool in_stock = 4;
}
```

proto3 Scalar Type	Roughly Equivalent To
`int32` / `int64`	Integer (32-bit / 64-bit)
`string`	UTF-8 text
`bool`	Boolean
`float` / `double`	Floating-point number
`bytes`	Raw binary data

Field Numbering Rules

The number after each field's `=` is not a default value — it's the field's **unique identifier in the binary wire format**. The field's *name* only exists for humans reading the `.proto` file; the actual bytes sent over the network reference fields **by number**, never by name.

- Numbers **1–15** encode in a single byte — reserve these for a message's most frequently set fields.
- Numbers **16 and above** take two bytes to encode — fine for less common fields.

- A field number, once used in any deployed version of a schema, should **never be reused** for a different field later — old serialized data (or old clients) may still reference it.

Repeated Fields

The `repeated` keyword marks a field as a list — the closest proto3 equivalent to a JSON array.

```
message Product {
  int32 id = 1;
  string name = 2;
  repeated string tags = 5;
}
```

Nested Messages

A message can use another message as a field's type, building the same kind of hierarchical structure a nested JSON object would.

```
message Address {
  string city = 1;
  string postcode = 2;
}

message Customer {
  int32 id = 1;
  string name = 2;
  Address shipping_address = 3;
}
```

Why Schema-First Beats REST's Often-Implicit Contract

A REST API's contract is frequently documented **separately** from the code that implements it — an OpenAPI/Swagger spec written by hand, generated after the fact, or simply a wiki page — and any of these can silently drift out of sync with what the API actually does, exactly the kind of documentation-vs-reality gap `api-design1-4` touched on. A `.proto` file has no such gap: it's not documentation *about* the contract, it **is** the contract — the exact same file that generates both the client and server code (Chapter 3), so client and server can never silently disagree about a message's shape.

REST's Contract vs. gRPC's Contract

REST (typically)

Documented separately (OpenAPI/Swagger, a wiki) — can drift from the real implementation over time if not diligently kept in sync.

gRPC

The `.proto` file *generates* both sides of the contract directly — client and server code both come from the same source, so they can't quietly diverge.

💻 Coding Challenges

Challenge 1: Write a Message Definition

Write a `.proto` message called `Order` with fields: `id` (int32), `customer_name` (string), and a `repeated` list of `item_names` (string). Assign field numbers deliberately.

Goal: Practice combining scalar fields and a repeated field in one message definition.

→ Solution

Challenge 2: Add a Nested Message

Extend Challenge 1's `Order` message with a nested `ShippingInfo` message (fields: `address` string, `expedited` bool) as a field on `Order`.

Goal: Practice defining and referencing a nested message type.

→ Solution

Challenge 3: Diagnose a Field Numbering Mistake

A team removes a deprecated field numbered `3` from a message, then later adds a completely different new field and assigns it number `3` to "reuse" it. Explain what could go wrong.

Goal: Practice applying this chapter's field-numbering rule to a realistic schema-evolution mistake.

→ Solution

⚠️ Gotcha: Changing a Field's Number Breaks Compatibility — Renaming It Doesn't

Because the wire format identifies fields by **number**, not name, renaming a field (e.g. `name` → `full_name`, keeping the same number) is completely safe — old and new code still agree on what number `2` means. But changing a field's *number*, or reusing a retired number for something unrelated, is genuinely dangerous: any client or stored data still using the old schema will misinterpret the field, potentially reading a string where it expects an integer, with no error raised at all. When a field is truly no longer needed, mark its number `reserved` rather than letting it be reassigned, so no one accidentally reuses it later.

🎯 What's Next

The next chapter is **Generating Code & Your First gRPC Service** — the `protoc` compiler, generated client/server stubs, service definition syntax, and a basic unary call end to end.

⚙️ Generating Code & Your First gRPC Service

Chapter 2 defined messages — the data shapes. This chapter defines the **service** — the actual RPC methods — and turns the whole `.proto` file into real, runnable client and server code.

Service Definition Syntax

A `service` groups related RPC methods together, the way a class groups methods — a structurally different grouping principle from REST's "group by resource URL."

```
service ProductService {
  rpc GetProduct(GetProductRequest) returns (Product);
  rpc CreateProduct(CreateProductRequest) returns (Product);
}

message GetProductRequest {
  int32 id = 1;
}
```

`rpc MethodName(RequestType) returns (ResponseType);` is the whole shape of a basic call — a named method, one request message type, one response message type.

The protoc Compiler

`protoc` reads a `.proto` file and, via language-specific plugins, generates real source code — client stubs and server base code — in whichever language each service needs.

```
# Generating Node.js code from product.proto
protoc --js_out=import_style=commonjs,binary:. \
    --grpc-web_out=import_style=commonjs,mode=grpcwebtext:. \
    product.proto
```

What Gets Generated

Two things come out of this process, both from the *same* `.proto` source — the concrete payoff of Chapter 2's "can't drift apart" claim:

- A **client stub** — a class with a method matching each `rpc` definition, handling all the connection, serialization, and network plumbing internally. Calling it looks like calling a normal local function.
- A **server base/interface** — generated routing and deserialization code, leaving exactly one thing for a developer to write: the actual method body implementing the business logic.

A Basic Unary Call End-to-End

A **unary** RPC — one request, one response — is the simplest of Chapter 4's four streaming modes, and the right starting point.

```
// Server: implementing the generated method body
function getProduct(call, callback) {
  const id = call.request.id;
  const product = database.findProduct(id);
  callback(null, product);
}

server.addService(ProductService.service, { GetProduct: getProduct });
// Client: calling it via the generated stub
const client = new ProductServiceClient('localhost:50051');

client.getProduct({ id: 42 }, (err, response) => {
  console.log(response); // the Product message, already deserialized
});
```

Writing a REST Handler vs. Implementing a gRPC Method

> **REST Route Handler**
>
> Manually parse the request body, validate it, call business logic, manually serialize the response — all written by hand, every time.

> **gRPC Method Implementation**
>
> Parsing and serialization are already handled by generated code — the method body is *just* the business logic itself.

Term	Meaning
Service definition	A named group of RPC methods in a `.proto` file
Unary RPC	One request, one response — the simplest call shape
`protoc`	The Protocol Buffers compiler that generates language-specific code
Client stub	Generated code a client calls — handles serialization/networking internally
Server base/interface	Generated routing/deserialization code a developer implements method bodies against

💻 Coding Challenges

Challenge 1: Write a Service Definition

Using Challenge 1's `Order` message from Chapter 2, write a `service` called `OrderService` with a unary `rpc` method `GetOrder` taking an `id` (int32) and returning an `Order`.

Goal: Practice writing the `rpc MethodName(Request) returns (Response)` syntax against a real message type.

→ Solution

Challenge 2: Identify What's Generated

For a `.proto` file defining `UserService` with one `rpc` method, list what `protoc` would generate on both the client and server side, and what a developer still has to write by hand.

Goal: Practice distinguishing generated code from the one piece that's never auto-generated: business logic.

→ Solution

Challenge 3: REST Handler vs. gRPC Method

Explain, using this chapter's comparison, why implementing a gRPC method body is typically shorter than an equivalent REST route handler, even for the exact same underlying business logic.

Goal: Practice articulating specifically which work moved from "written by hand" to "generated" and why.

→ Solution

⚠️ Gotcha: Forgetting to Regenerate Stubs After a .proto Change

Editing a `.proto` file doesn't automatically update the generated client/server code sitting in the project — `protoc` has to be re-run explicitly every time the schema changes. A common, easy-to-hit workflow mistake: adding a new field or method to the `.proto` file, then testing against stale generated stubs that still reflect the old schema, producing confusing "field doesn't exist" errors that have nothing to do with the actual code being written. Most real projects wire stub regeneration into a build script or CI step specifically to prevent stale generated code from ever being an option.

🎯 What's Next

The next chapter covers **The Four Streaming Modes** — unary (just covered), server-streaming, client-streaming, and bidirectional streaming, and when each genuinely fits.

🔀 The Four Streaming Modes

Chapter 3 covered unary calls — one request, one response. HTTP/2's native support for long-lived, multiplexed streams (Chapter 1) lets gRPC go further, with three more call shapes REST has no built-in equivalent for at all (REST needs WebSockets or SSE bolted on, already covered in the WebSockets course).

Unary — Recap

One request in, one response out. The baseline every other mode is described relative to.

```
rpc GetProduct(GetProductRequest) returns (Product);
```

Server Streaming

One request, **many** responses streamed back over time. The `stream` keyword on the response type marks it.

```
rpc WatchPrice(WatchPriceRequest) returns (stream PriceUpdate);
```

Right for **live updates** — subscribing to stock price changes, streaming search results as they're found, watching a long-running job's progress. One request establishes the subscription; the server pushes as many responses as it needs to, whenever it has something new.

Client Streaming

Many requests streamed from the client, **one** response at the end. The `stream` keyword moves to the request type instead.

```
rpc UploadReadings(stream SensorReading) returns (UploadSummary);
```

Right for **batch uploads** — sending a large file in chunks, or submitting many sensor readings and getting back a single confirmation/summary once all of them have arrived, rather than paying the overhead of a separate unary call per chunk.

Bidirectional Streaming

Both sides stream **independently and simultaneously** — neither side waits for the other to finish before sending more.

```
rpc Chat(stream ChatMessage) returns (stream ChatMessage);
```

Right for genuinely **chat-like exchanges** — a live chat, real-time collaborative editing — the closest gRPC gets to the full-duplex model `web-sockets1-2` defined for raw WebSockets.

Mode	Syntax Shape	Use Case
Unary	`rpc M(Req) returns (Res);`	A single request/response — the default choice
Server streaming	`rpc M(Req) returns (stream Res);`	Live updates, subscriptions
Client streaming	`rpc M(stream Req) returns (Res);`	Batch uploads, chunked submission
Bidirectional streaming	`rpc M(stream Req) returns (stream Res);`	Chat, real-time collaboration

Implementing Server Streaming

```
function watchPrice(call) {
  const interval = setInterval(() => {
    call.write({ price: getCurrentPrice() });
  }, 1000);

  call.on('cancelled', () => clearInterval(interval));
}
```

Unlike a unary handler's single `callback`, a server-streaming handler calls `write()` as many times as it needs to, and must explicitly handle the client disconnecting mid-stream — a lifecycle unary calls never have to think about.

gRPC Bidi Streaming vs. Raw WebSockets

Both are full-duplex, but gRPC's bidirectional streams are still schema-constrained by Protobuf messages and shaped around an RPC method — raw WebSockets (the WebSockets course) are a free-form byte/message channel with no inherent request/response or schema structure at all.

When to Reach for Which

gRPC bidi streaming fits when both ends are internal services that benefit from a typed schema; raw WebSockets fit better when a browser is one of the endpoints, or when the message shape needs to stay flexible.

💻 Coding Challenges

Challenge 1: Pick the Right Streaming Mode

For each, name the correct mode and write its `rpc` signature: (a) a client uploading a large log file in 1MB chunks, getting one confirmation back, (b) a dashboard subscribing to live order-count updates.

Goal: Practice matching a use case to the correct streaming direction and syntax.

→ Solution

Challenge 2: Explain Why Unary Doesn't Fit

Explain why implementing a live stock-price subscription as a unary call (the client polling repeatedly) is a worse fit than server streaming, in terms of both efficiency and design.

Goal: Practice articulating what server streaming specifically buys over repeated unary polling.

→ Solution

Challenge 3: gRPC Bidi or WebSockets?

A team is building a real-time multiplayer game where the browser is one of the two endpoints. Should they use gRPC bidirectional streaming or raw WebSockets? Justify your answer.

Goal: Practice applying this chapter's gRPC-vs-WebSockets distinction to a concrete architectural decision.

→ Solution

⚠️ Gotcha: Streaming Adds Real Complexity — Don't Reach for It by Default

A streaming handler has to manage a genuine lifecycle a unary call never does: knowing when the stream should end, handling a client disconnecting mid-stream, and dealing with errors that occur partway through rather than all-or-nothing. This complexity is worth paying when a use case genuinely needs ongoing or many-part communication — but it's a real cost, not a free upgrade. **Unary should be the default** for any call that's fundamentally "ask a question, get an answer"; reach for one of the three streaming modes only when the shape of the actual problem — live updates, chunked uploads, a genuinely two-way exchange — calls for it.

🎯 What's Next

The next chapter is **Error Handling & Metadata** — gRPC status codes vs. HTTP status codes, rich error details, metadata (gRPC's header equivalent), and deadlines/timeouts.

🚦 Error Handling & Metadata

Chapters 3–4 built calls that succeed. This chapter covers what happens when they don't, how to attach auxiliary data alongside a call without cramming it into the message itself, and how to stop a call from waiting forever.

gRPC Status Codes vs. HTTP Status Codes

gRPC has its own fixed enum of status codes, purpose-built for RPC semantics — distinct from HTTP's status codes, even though gRPC calls physically ride on top of HTTP/2.

HTTP Status Codes vs. gRPC Status Codes

HTTP

~40 general-purpose codes designed for web resources and REST conventions (404, 401, 500...).

gRPC

A fixed set of ~17 codes designed specifically for RPC outcomes — several (like `DEADLINE_EXCEEDED` or `ALREADY_EXISTS`) have no clean HTTP equivalent at all.

gRPC Status Code	Meaning
`OK`	The call succeeded
`INVALID_ARGUMENT`	The request itself was malformed
`NOT_FOUND`	The requested resource doesn't exist
`ALREADY_EXISTS`	An attempt to create something that already exists
`PERMISSION_DENIED`	Authenticated, but not authorized for this action
`UNAUTHENTICATED`	No valid credentials were provided at all
`DEADLINE_EXCEEDED`	The call didn't complete before its deadline
`UNAVAILABLE`	The service is temporarily unreachable — often safe to retry
`INTERNAL`	An unexpected server-side error

Rich Error Details

Beyond a status code and a message, gRPC supports attaching **structured error details** via the `google.rpc.Status` extension mechanism — field-level validation errors, retry-after hints, or any custom structured data, all still strongly typed via Protobuf rather than the ad-hoc, differently-shaped error JSON bodies common across different REST APIs.

Metadata

Metadata is gRPC's equivalent of HTTP headers — key-value pairs sent alongside a call, outside the message body itself. Common uses: auth tokens (Chapter 7 covers this properly), request tracing IDs, and custom routing hints.

```
// Client: attaching metadata (an auth token, preview of Chapter 7)
const metadata = new grpc.Metadata();
metadata.add('authorization', `Bearer ${token}`);

client.getProduct({ id: 42 }, metadata, (err, response) => { ... });
```

Deadlines & Timeouts

Every gRPC call should carry a **deadline** — an absolute point in time by which the call must complete, after which it's automatically cancelled with `DEADLINE_EXCEEDED`. Deadlines are more powerful than a plain client-side timeout: they **propagate** across a chain of calls. If Service A calls Service B, which calls Service C, the same deadline can flow through all three — a slow Service C doesn't leave Service A waiting indefinitely, because the whole chain shares one clock, directly relevant to Chapter 8's microservices coverage.

```
// Client: a call with a 5-second deadline
const deadline = Date.now() + 5000;
client.getProduct({ id: 42 }, { deadline }, (err, response) => {
  if (err && err.code === grpc.status.DEADLINE_EXCEEDED) {
    // the call was cancelled — it never completed in time
  }
});
```

💻 Coding Challenges

Challenge 1: Pick the Right Status Code

For each, name the correct gRPC status code: (a) a client requests a product ID that doesn't exist, (b) a client sends a request with no auth token at all, (c) a downstream service is temporarily down and the call couldn't be completed.

Goal: Practice matching a concrete failure to its specific gRPC status code, not just a generic "error."

→ Solution

Challenge 2: Explain Deadline Propagation

Service A calls Service B, which calls Service C. Explain, using this chapter's material, what deadline propagation buys this chain that a plain timeout set independently at each hop wouldn't.

Goal: Practice explaining why a shared deadline across a call chain is a genuine architectural advantage, not just a convenience.

→ Solution

Challenge 3: Metadata or Message Field?

For each, say whether it belongs in gRPC metadata or as a field in the request message: (a) an auth token, (b) the product ID being requested, (c) a request-tracing ID used for logging across services.

Goal: Practice distinguishing data that's genuinely part of the call's payload from auxiliary data that travels alongside it.

→ Solution

⚠️ Gotcha: An HTTP 200 Does Not Mean the gRPC Call Succeeded

Because gRPC rides on top of HTTP/2, a call's underlying HTTP status is very often `200` **regardless of whether the gRPC call itself succeeded** — the real outcome is communicated separately, via gRPC's own status field sent in an HTTP/2 trailer. Code that checks only the HTTP-level response (natural habit coming from REST) can easily miss a genuine `NOT_FOUND` or `INTERNAL` failure entirely. Always check the gRPC status specifically — generated client stubs surface this through the language's own error/exception mechanism (as in this chapter's `err.code` example), not through an HTTP status code. Relatedly, deadline propagation is typically **opt-in per language/library**, not automatic just because a deadline was set on the outermost call — check the specific gRPC library's documentation for how it actually threads a deadline through nested calls.

🎯 What's Next

The next chapter is **Interceptors & Middleware** — client/server interceptors for logging, auth, and retries, the gRPC equivalent of Express middleware.

🧩 Interceptors & Middleware

Chapter 5's auth-metadata example checked a token by hand, once, in one method. This chapter shows how to avoid repeating that in *every* method — **interceptors**, gRPC's direct structural equivalent to the Express middleware chain (`express1-3`).

The Problem: Repeating Logic in Every Method

Without interceptors, every single RPC method implementation would need to manually check auth metadata, log the call, and handle retries — repetitive, error-prone, and one new method away from someone forgetting the auth check entirely. This is exactly why Express middleware exists in the first place (`express1-3`): cross-cutting concerns don't belong duplicated inside every route handler.

Server Interceptors

A **server interceptor** wraps around *every* incoming call before it reaches the actual method handler — the same role `app.use()` middleware plays before Express route handlers run.

```
// A server interceptor checking auth ONCE, for every method
function authInterceptor(call, next) {
 const token = call.metadata.get('authorization')[0];
 if (!isValidToken(token)) {
  throw new Error(grpc.status.UNAUTHENTICATED);
 }
 return next(call);
}
```

Every method handler defined in Chapter 3 now runs *behind* this check — none of them need to repeat the token-parsing logic Chapter 5's example wrote inline.

Client Interceptors

A **client interceptor** wraps around every *outgoing* call before it's actually sent — common uses include automatically attaching auth metadata to every call, retries with backoff, and logging.

```
// A client interceptor auto-attaching auth to every outgoing call
function authClientInterceptor(options, nextCall) {
 return new InterceptingCall(nextCall(options), {
  start(metadata, listener, next) {
   metadata.add('authorization', `Bearer ${getToken()}`);
   next(metadata, listener);
  }
 });
}
```

Client code calling `getProduct()` no longer needs to manually build and pass a `Metadata` object with a token on every call site the way Chapter 5's raw example did — the interceptor attaches it automatically, exactly once, in one place.

Express Middleware vs. gRPC Interceptors

Express Middleware

`app.use()` registers a chain that runs before route handlers, each calling `next()` to pass control along — one direction, server-side only.

gRPC Interceptors

The same wrap-before-the-real-handler concept, but on *both* sides — server interceptors wrap incoming calls, client interceptors wrap outgoing ones.

Use Case	Server or Client?	Example
Central auth check	Server	Reject unauthenticated calls before any handler runs
Auto-attaching auth metadata	Client	Every outgoing call gets a token automatically
Request/response logging	Either	Log every call attempt, regardless of outcome
Automatic retries	Client	Retry a call on `UNAVAILABLE` with backoff

Server Interceptor Responsibilities

Anything that should apply uniformly to every incoming call before business logic runs — auth, rate limiting, request logging.

Client Interceptor Responsibilities

Anything that should apply to every outgoing call automatically — attaching credentials, retry logic, response-time logging.

💻 Coding Challenges

Challenge 1: Refactor Chapter 5's Auth Check

Chapter 5's `getProduct` handler manually checked auth metadata inline. Rewrite this as a server interceptor instead, so the handler itself no longer needs any auth-checking code.

Goal: Practice moving repeated per-method logic into a centralized interceptor.

→ Solution

Challenge 2: Server or Client Interceptor?

For each, say whether it belongs as a server interceptor or a client interceptor: (a) logging how long every call took to complete, (b) rejecting calls that exceed a rate limit, (c) automatically retrying a failed call up to 3 times.

Goal: Practice distinguishing responsibilities that belong on the receiving side from ones that belong on the calling side.

→ Solution

Challenge 3: Explain Interceptor Ordering

A team places their auth interceptor before their logging interceptor. Explain what this means for calls that get rejected by the auth interceptor, and propose a reordering if they want to log every attempt, including rejected ones.

Goal: Practice reasoning about interceptor order the same way you'd reason about Express middleware order.

→ Solution

⚠️ Gotcha: Interceptor Order Matters, Just Like Express Middleware Order

Interceptors run in a chain, and — exactly like Express middleware — the **order** they're registered in determines what each one sees. If an auth interceptor runs first and rejects a call, any interceptor registered *after* it (a logging interceptor, say) never runs at all for that call, since the chain short-circuits at the rejection. This means a logging interceptor placed after auth silently misses every rejected/unauthenticated attempt — often exactly the calls worth logging most. If visibility into every attempt (including rejected ones) matters, logging needs to run *before* auth in the chain, not after — the same ordering discipline Express developers already apply to their own middleware stacks.

🎯 What's Next

The next chapter is **Authentication & Security in gRPC** — TLS by default, token-based auth via metadata in practice, and mutual TLS for service-to-service calls.

🔐 Authentication & Security in gRPC

Chapters 5–6 built the metadata and interceptor mechanics. This chapter puts real security behind them — reusing the `https1` course's TLS mechanics and `bc1`'s token-based auth wholesale, plus one technique especially suited to gRPC's service-to-service origins: mutual TLS.

TLS by Default

Where a REST API often starts as plain HTTP and has HTTPS bolted on later, gRPC channels are typically created **with TLS from the very start** — the same handshake and certificate mechanics `https1-4`/`https1-6` already covered in depth, just carrying gRPC's HTTP/2 traffic instead of a browser's.

```
// Production: a secure channel using TLS
const client = new ProductServiceClient(
  'products.internal:443',
  grpc.credentials.createSsl()
);

// Local development ONLY — never in production
const devClient = new ProductServiceClient(
  'localhost:50051',
  grpc.credentials.createInsecure()
);
```

Token-Based Auth via Metadata

Chapters 5–6 already built this pattern — a bearer token passed as metadata, validated once by a server interceptor. This chapter names it formally: it's gRPC's version of `bc1-7`'s token-based authentication, and it's a genuinely **separate layer** from the channel's TLS. **Channel credentials** (TLS) secure the transport itself; **call credentials** (the token in metadata) establish who's making a specific call — the same "one secret, defended end to end" layering `bc1`'s quartet framing established for the site's Auth course generally.

Mutual TLS (mTLS) for Service-to-Service Calls

Standard TLS only proves the *server's* identity to the client, via the server's certificate. **Mutual TLS** goes further: the *client* also presents its own certificate, and the server verifies it — both sides authenticate each other. This fits gRPC's original service-to-service use case (Chapter 1) especially well: there's no human typing a password on either end, so a certificate-based identity is the natural fit for proving "this call is genuinely coming from the order service, not an impostor."

```
// Server requiring client certificates (mTLS)
const serverCredentials = grpc.ServerCredentials.createSsl(
 rootCert,
 [{ private_key: serverKey, cert_chain: serverCert }],
 true // checkClientCertificate — enforce mTLS
);
```

Standard TLS vs. Mutual TLS

> **Standard TLS**
>
> Only the server proves its identity — right for calls from a human or external client that doesn't have its own certificate.

> **Mutual TLS**
>
> Both sides present certificates — right for internal service-to-service calls where each service has a well-defined identity worth verifying.

Security Layer	Mechanism	Protects Against
Channel credentials (TLS)	Server certificate, encrypted transport	Eavesdropping, tampering in transit
Call credentials (bearer token)	Metadata, validated per-call	Unauthenticated/unauthorized calls
Mutual TLS	Client *and* server certificates	An impostor service pretending to be a trusted caller

Bearer Tokens Fit When

The caller is a human user or an external client authenticating via credentials rather than a certificate — the same territory `bc1` covers generally.

mTLS Fits When

Both endpoints are internal services within a trusted network, where each service having its own verifiable certificate identity is practical (Chapter 8's microservices territory).

💻 Coding Challenges

Challenge 1: Channel vs. Call Credentials

Explain the difference between channel credentials and call credentials in gRPC, using this chapter's TLS-and-bearer-token example.

Goal: Practice distinguishing transport-level security from per-call identity as two separate, layered concerns.

→ Solution

Challenge 2: Bearer Token or mTLS?

For each, recommend bearer-token auth or mTLS: (a) a mobile app calling a public-facing gRPC gateway, (b) an internal payments service calling an internal ledger service, both within the same company's private network.

Goal: Practice applying the human/external-vs-internal-service distinction to concrete scenarios.

→ Solution

Challenge 3: Spot the Security Mistake

A developer ships a gRPC service to production using `grpc.credentials.createInsecure()` "because it worked fine in local testing." Explain what's wrong with this.

Goal: Practice recognizing a real, concrete security regression rather than a purely theoretical one.

→ Solution

⚠️ Gotcha: createInsecure() Belongs Only in Local Development

An insecure channel sends every call — including any bearer token attached via metadata — in **plaintext**, exactly the same risk the site's HTTPS/TLS and Database Security courses have flagged repeatedly for other protocols: an unencrypted channel means anyone positioned to observe the traffic reads everything, credentials included. It's easy to reach for `createInsecure()` during local development (no certificates to manage) and simply forget to switch to `createSsl()` before deploying — treat this exactly like the site's recurring "never expose unencrypted" theme (database connections, TLS certificate verification): a deliberate, checked step before anything reaches production, not an assumption.

🎯 What's Next

The next chapter is **gRPC in Microservices & Load Balancing** — service discovery, client-side vs. proxy load balancing, and when gRPC is the right internal-service choice vs. REST/GraphQL for public-facing APIs.

🌐 gRPC in Microservices & Load Balancing

Chapter 7 secured calls between services. This chapter covers the operational reality of running those services as many instances at once — how a client finds a healthy instance at all, and how load actually gets spread across them.

Service Discovery

In a real microservices deployment, a service like `order-service` rarely runs as one fixed instance — it runs as **many**, scaling up and down, restarting, and being redeployed constantly. A client needs a way to find *currently healthy* instances rather than hardcoding one IP address that might not even exist anymore by the time a call is made. **Service discovery** solves this — a DNS record resolving to multiple IPs, or a dedicated registry (Consul, etcd, or Kubernetes' own built-in service discovery) that gRPC's resolver plugins can query directly.

Client-Side Load Balancing

With **client-side load balancing**, the gRPC client itself receives a list of available addresses (via service discovery) and decides which one to send each call to — round-robin is the simple default. This works especially well for gRPC specifically because of Chapter 1's HTTP/2 multiplexing: a client picks a backend connection *once* and reuses it for many calls, rather than a fresh choice being needed for every individual request the way a simpler, connectionless client model might behave.

```
// Conceptual: a DNS-based resolver feeding round-robin client-side LB
const client = new OrderServiceClient(
  'dns:///order-service.internal:50051',
  { loadBalancingPolicy: 'round_robin' }
);
```

Proxy Load Balancing

The alternative: a dedicated **proxy** (an L7 proxy like Envoy, common in a service mesh) sits between client and servers. The client just talks to the proxy — it never needs to know about individual instances, service discovery, or balancing logic at all; the proxy handles all of it.

Client-Side vs. Proxy Load Balancing

Client-Side

The client is balancing-aware — it runs resolver logic and picks a backend itself. Simpler infrastructure, more logic embedded in every client.

Proxy

The client stays simple — a proxy (often part of a service mesh like Istio+Envoy) does all the routing and balancing work centrally.

Approach	Who Decides Routing	Common Use
Client-side LB	The client itself, via a resolver	Simpler deployments, fewer moving infrastructure pieces
Proxy LB	A dedicated proxy/service mesh	Larger microservice fleets, centralized traffic policy

Client-Side LB Fits When

The deployment is simpler, and adding resolver/balancing logic to clients directly is an acceptable trade for not running extra infrastructure.

Proxy/Service Mesh Fits When

Many services need consistent routing, retry, and observability policy applied centrally, without embedding that logic separately into every client.

When gRPC Is the Right Internal-Service Choice vs. REST/GraphQL for Public APIs

This closes the loop back to Chapter 1's positioning: gRPC's performance and schema-first benefits pay off most **within a trust boundary you control** — internal services, secured with mTLS (Chapter 7), discovered and balanced across instances (this chapter). The moment an API needs to be called by a browser or an arbitrary third-party developer, Chapter 1's browser-support gotcha becomes decisive again: REST or GraphQL's universal compatibility wins for anything public-facing, while gRPC remains the stronger choice for the internal service-to-service traffic it was originally built for.

💻 Coding Challenges

Challenge 1: Explain Why Hardcoding an IP Fails

Explain why a gRPC client hardcoding one server IP address is a poor fit for a real microservices deployment, using this chapter's material.

Goal: Practice connecting the dynamic nature of scaled deployments to the need for service discovery.

→ Solution

Challenge 2: Client-Side or Proxy LB?

A small team runs three internal services with simple routing needs and wants to avoid operating extra infrastructure. A larger organization runs 200 microservices and wants centralized traffic policy and observability. Recommend an approach for each.

Goal: Practice matching organizational scale to the right load-balancing approach.

→ Solution

Challenge 3: Synthesize the Course's gRPC-vs-REST Thread

Using material from Chapters 1, 7, and this chapter, explain in a few sentences why gRPC is well suited to internal microservices but not to a public API a mobile app calls directly.

Goal: Practice pulling together multiple chapters' worth of reasoning into one coherent argument.

→ Solution

⚠️ Gotcha: Long-Lived Connections Can Leave New Instances Starved of Traffic

Chapter 1's HTTP/2 multiplexing is a real strength — but it has an operational cost here: because gRPC clients reuse one long-lived connection for many calls, a client that already picked a backend connection before a *new* instance was added to the pool has no reason to reconnect and discover it. In a naive setup, a freshly scaled-up instance can sit there receiving **zero** traffic while existing clients keep happily reusing their old connections to the original instances — the opposite of what adding capacity was supposed to achieve. Real deployments address this with periodic connection refresh/rebalancing (client-side) or by routing all traffic through a proxy (this chapter's proxy approach) that's aware of new instances immediately, rather than assuming clients will naturally redistribute themselves.

🎯 What's Next

The final chapter is the **Capstone: Building a Small gRPC Service** — a real two-service example (a user service plus an order service calling it) combining schema design, streaming, interceptors, and auth.

🏁 Capstone: Building a Small gRPC Service

This capstone builds a small, real two-service system: **UserService** and **OrderService**, where OrderService calls UserService internally — combining schema design, streaming, interceptors, and auth from every prior chapter.

The System: UserService + OrderService

Requirements: an end user (with a bearer token) calls `OrderService.CreateOrder`; OrderService needs to look up the caller's details from UserService — a genuine internal service-to-service call, secured with mTLS; a client can then subscribe to live order status updates via streaming; every call on both services is logged.

Step 1: Schema Design (Chapter 2)

```
// user.proto
service UserService {
  rpc GetUser(GetUserRequest) returns (User);
}
message GetUserRequest { int32 id = 1; }
message User {
  int32 id = 1;
  string name = 2;
}

// order.proto
service OrderService {
  rpc CreateOrder(CreateOrderRequest) returns (Order);
  rpc WatchOrderStatus(WatchOrderStatusRequest) returns (stream OrderStatusUpdate);
}
message CreateOrderRequest {
  int32 user_id = 1;
  repeated string item_names = 2;
}
message Order {
  int32 id = 1;
  string customer_name = 2;
  string status = 3;
}
```

Step 2: The Service-to-Service Call (Chapters 1, 3, 7)

Creating an order needs the customer's name — OrderService fetches it by calling UserService directly, exactly the internal service-to-service pattern Chapter 1 named as gRPC's original purpose, secured with the mutual TLS Chapter 7 covered.

```
async function createOrder(call, callback) {
 const { user_id, item_names } = call.request;

 // OrderService calling UserService internally, over mTLS
 userServiceClient.getUser({ id: user_id }, (err, user) => {
  if (err) return callback(err);

  const order = database.createOrder({ customer_name: user.name, item_names });
  callback(null, order);
 });
}
```

Step 3: Streaming Order Status (Chapter 4)

A server-streaming RPC lets a client subscribe to live updates as an order moves through processing.

```
function watchOrderStatus(call) {
 const unsubscribe = orderEvents.on(call.request.order_id, (status) => {
  call.write({ status });
 });
 call.on('cancelled', unsubscribe);
}
```

Step 4: Interceptors Tying It Together (Chapter 6)

A shared logging interceptor applies to both services; an auth interceptor on OrderService validates the end user's bearer token before `createOrder` ever runs.

```
orderServer.use(loggingInterceptor);   // registered first — logs every attempt
orderServer.use(authInterceptor);      // validates the bearer token

userServer.use(loggingInterceptor);
userServer.use(mtlsCheckInterceptor);  // only accepts calls with a valid client cert
```

Step 5: Putting It All Together

1. A client (holding a bearer token) calls `OrderService.CreateOrder`.
2. OrderService's logging interceptor records the attempt; its auth interceptor validates the token, rejecting with `UNAUTHENTICATED` if invalid.

3. The `createOrder` handler calls `UserService.GetUser` internally, over an mTLS-secured channel — UserService's own interceptors verify the client certificate before responding.
4. The order is created and returned to the original client.
5. The client calls `WatchOrderStatus` to receive live status updates as the order is processed.

Course Concept	Where It's Used in This App
Service definitions & messages (Ch.2)	`user.proto` / `order.proto`
Service-to-service calls (Ch.1, 3)	OrderService calling UserService's `GetUser`
Server streaming (Ch.4)	`WatchOrderStatus`
Interceptors (Ch.6)	Shared logging + auth on OrderService, mTLS check on UserService
Bearer tokens & mTLS (Ch.7)	End-user auth on OrderService; service identity on the OrderService→UserService call

💻 Coding Challenges

Challenge 1: Add Error Handling to the Internal Call

Extend Step 2's `createOrder` handler so that if `UserService.GetUser` returns `NOT_FOUND`, the whole `CreateOrder` call fails with a clear, matching error rather than a generic one.

Goal: Practice propagating a meaningful gRPC status code (Chapter 5) across a service-to-service call.

→ Solution

Challenge 2: Design a Client-Streaming Addition

Design a new client-streaming RPC, `ImportOrders`, that lets a client submit many orders in one call and receive a single summary response with the total count created.

Goal: Practice applying Chapter 4's client-streaming mode to a new scenario within this capstone's schema.

→ Solution

Challenge 3: Justify the Interceptor Order

Explain why Step 4 registers the logging interceptor before the auth interceptor on OrderService, referencing Chapter 6's material.

Goal: Practice applying the interceptor-ordering lesson to this capstone's specific setup.

→ Solution

⚠️ Gotcha: Deadlines Need to Be Threaded Through the Internal Call Too

Chapter 5 established that deadlines propagate across a call chain — but only if the code actually threads them through. If `createOrder`'s call to `UserService.GetUser` doesn't derive its own deadline from the incoming client call's remaining time budget, the two calls end up with **independent** timeouts: the original client's deadline could expire while OrderService is still waiting on a UserService call that has no idea a time budget even exists. Every internal call made from within a handler needs to explicitly forward (or derive from) the deadline it received, not start a fresh, unrelated one — exactly the kind of detail that's easy to get right in a single-service example and easy to silently miss the moment a real call chain like this one exists.

🎉 Course Complete

That's the full **gRPC** course — from the RPC model and HTTP/2 transport through Protocol Buffers, code generation, all four streaming modes, error handling and metadata, interceptors, authentication and security, microservices/load balancing, and finally a real two-service system combining all of it. Together with API Types & Design and API Testing & Tooling, this completes the site's "APIs & Integration" area.