Gorgo Go for Java Programmers

June 11, 2026

Contents

Chapter 16

gRPC

The previous chapter built services that speak JSON over HTTP — the lingua franca of the public web. That works, but it leaves a lot on the table for service-to-service traffic inside your own systems. JSON is text, so every request pays to serialize numbers into strings and parse them back. The contract lives in documentation (or in your head), so a typo in a field name fails silently at runtime rather than at compile time. And every endpoint reinvents pagination, streaming, deadlines, and error codes by hand.

gRPC fixes all three. You describe your service once in a `.proto` file — the messages and the methods — and a code generator produces strongly typed Go (and Java, and Python, and...) for both sides. Messages travel as compact binary over HTTP/2, which also gives you multiplexed streams for free. If you have used Java's gRPC stack (the `protobuf-maven-plugin`, `StreamObserver`, blocking and async stubs), the concepts here are identical — the contract is the same `.proto` — but Go's generated code is far less ceremonious. Think of it as the typed, binary, streaming counterpart to the REST services from Chapter 15: same network, very different ergonomics.

RPC stands for Remote Procedure Call. As programmers we are very comfortable with function calls — also known as procedure calls. They are one of the early concepts that we learn in programming, so RPC endeavors to make calling a remote service as easy as calling a local function. Unfortunately, network failures, object serialization, remote failures, and delays are all challenges that crop up with RPCs that aren't present with local functions. gRPC — Google's version of RPC — is still a useful tool to use, but it is far from simple.

This chapter gives a brief introduction to Protocol Buffers and the `.proto` language, the `protoc` toolchain that generates Go code, implementing and running a gRPC server, calling it from a client, all four kinds of RPC (unary and the three streaming flavors), propagating deadlines and metadata, returning typed errors with status codes, interceptors (gRPC's middleware), and securing the wire with TLS. We encourage you to learn more from the official gRPC documentation (The gRPC Authors 2025).

Protocol Buffers

A gRPC service starts with a schema written in **Protocol Buffers** (protobuf), Google's language-neutral interface definition language. We can only use types that gRPC knows about or that we have defined with its definition language. A `.proto` file declares the *messages* — gRPC lingo for *types* — and the *services* — the methods that use those types. Code for both the client and server is generated from it, so they can consistently use field names and types when communicating — even using differing programming languages.

Here is the schema we will use for the rest of the chapter — a small song catalog:

```
syntax = "proto3";                      // always proto3 for modern gRPC

package music.v1;                       // protobuf namespace
```

```
option go_package = "example/musicpb";   // Go import path for generated code

message Song {
  string id     = 1;                     // field numbers, not values
  string title  = 2;
  string artist = 3;
  int32  bpm    = 4;
}

message GetSongRequest {
  string id = 1;
}

service Jukebox {
  rpc GetSong(GetSongRequest) returns (Song);  // one request, one response
}
```

Notice that if you change **message** to **class** the message definitions look almost like Java classes, except for the numbers after each field. The numbers after each field (`= 1`, `= 2`) are **field tags**, not default values. They are how protobuf identifies a field on the wire. The *name* is used to generate methods and fields, and the *field tags* are used to encode data to send over the network — the field *name* never travels, only its tag. This is why protobuf payloads are small, and why you can rename a field freely but must never reuse or change a tag number.

Just like Go, Protocol Buffers have built-in types. The scalar types map onto Go types predictably:

proto type	Go type	Java type
`string`	`string`	`String`
`bool`	`bool`	`boolean`
`int32`, `sint32`	`int32`	`int`
`int64`, `sint64`	`int64`	`long`
`double`	`float64`	`double`
`float`	`float32`	`float`
`bytes`	`[]byte`	`ByteString`
`repeated T`	`[]T`	`List<T>`
`map<K,V>`	`map[K]V`	`Map<K,V>`

Wut: In proto3 every scalar field has a zero-value default and there is no built-in "was it set?" bit for scalars. A `bpm` of `0` is indistinguishable from "bpm not provided." When that distinction matters, either wrap the field (`optional int32 bpm = 4;`, which generates a `*int32` in Go) or use a dedicated message. This is the protobuf analog of the `int` vs `Integer` boxing decision in Java.

The `protoc` Toolchain

Go code does not read `.proto` files at runtime. You run a compiler, `protoc`, ahead of time to generate `.go` files. Two plugins do the work: `protoc-gen-go` generates the message structs, and `protoc-gen-go-grpc` generates the client and server stubs.

Install the plugins (they are ordinary Go tools) and make sure `protoc` itself is on your `PATH`:

```
# the two code generators
go install google.golang.org/protobuf/cmd/protoc-gen-go@latest
```

```
go install google.golang.org/grpc/cmd/protoc-gen-go-grpc@latest
```

The `protoc` compiler itself does not come from `go install` — you get it from your platform's package manager:

```
# macOS (Homebrew)
brew install protobuf

# Debian/Ubuntu
apt install protobuf-compiler

# Windows (Chocolatey)
choco install protoc
```

On Windows you can also use Scoop (`scoop install protobuf`). Whichever you pick, make sure the resulting `protoc` is on your `PATH` — and that the Go tool directory (`go env GOBIN`, or `$(go env GOPATH)/bin`) is too, so the two plugins are found.

Generate the Go code from `music.proto`:

```
protoc --go_out=. --go_opt=paths=source_relative \
       --go-grpc_out=. --go-grpc_opt=paths=source_relative \
       music.proto
```

This writes two files next to the proto: `music.pb.go` (structs representing the message types `Song`, `GetSongRequest`, ...) and `music_grpc.pb.go` (the `JukeboxServer` interface, the `JukeboxClient` stub that performs the gRPC calls, and the `RegisterJukeboxServer` registration function). You commit the generated files alongside your hand-written code and regenerate whenever the `.proto` changes.

The plugins you installed with `go install` are *host tools* — they run on your machine to produce code and never ship with your program. The generated code, however, imports two *runtime libraries* that your program does link against: `google.golang.org/protobuf` (the wire encoding) and `google.golang.org/grpc` (the client and server machinery). Those have to be real dependencies in your module's `go.mod`, so add them with `go get`:

```
go get google.golang.org/grpc
go get google.golang.org/protobuf
```

In practice a single `go mod tidy` after generating will discover both from the new imports and pin them for you. The Java analog is the split between the `protoc` plugin in your build file and the `grpc-stub`/`protobuf-java` artifacts on your classpath — Go just keeps the build tools out of `go.mod` entirely.

Tip: Memorizing that `protoc` invocation is nobody's idea of fun. Most teams use `buf` (`buf.build`) instead: a `buf.gen.yaml` describes the plugins once, and `buf generate` runs them — plus `buf lint` and `buf breaking` catch style problems and wire-incompatible changes before they ship. It is the protobuf equivalent of replacing a hand-rolled `protobuf-maven-plugin` block with a single linted config.

Trap: Never hand-edit the generated `*.pb.go` files. They are overwritten on every regeneration. Put your logic in separate files that *use* the generated types.

Building a gRPC Server

There are two parts of the gRPC server that get generated for us: the interface that we need to implement and an embedded type.

The interface is named after the gRPC service — with a `Server` suffix added — and each of the RPCs for the service shows up as a method. For example, the generated `music_grpc.pb.go` declares the interface:

```
// generated --- one method per rpc in the service
type JukeboxServer interface {
    GetSong(context.Context, *GetSongRequest) (*Song, error)
    mustEmbedUnimplementedJukeboxServer()  // forward-compatibility guard
}
```

That last unexported method is deliberate: you cannot satisfy the interface by accident. Instead you **embed** the generated `UnimplementedJukeboxServer` (Chapter 6 embedding) into your type, which supplies a default "unimplemented" body for every method. Then you override only the methods you actually handle. The `Unimplemented` prefix may sound awkward, but the name comes from the fact that it provides an implementation of every method that just returns the `codes.Unimplemented` error.

In the code below, notice that `jukeboxServer` does not need to explicitly say that it implements the `JukeboxServer` interface, but it does need to embed the `musicpb.UnimplementedJukeboxServer` type.

```
package main

import (
    "context"
    "sync"

    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"
    "example/musicpb"
)

type jukeboxServer struct {
    musicpb.UnimplementedJukeboxServer            // embed for forward compatibility
    mu      sync.Mutex                            // guards catalog (Chapter 11)
    catalog map[string]*musicpb.Song
}

func (s *jukeboxServer) GetSong(
    ctx context.Context, req *musicpb.GetSongRequest,
) (*musicpb.Song, error) {
    s.mu.Lock()
    defer s.mu.Unlock()

    song, ok := s.catalog[req.GetId()]
    if !ok {
        return nil, status.Errorf(codes.NotFound, "no song with id %q", req.GetId())
    }
    return song, nil
}
```

Notice `req.GetId()` rather than `req.Id`. The generated code gives every field a getter that is `nil`-safe: calling `GetId()` on a `nil *GetSongRequest` returns the zero value instead of panicking. Prefer the getters when reading request fields.

Wiring it up looks a lot like the raw `net.Listen` server from Chapter 15, except a `grpc.Server` sits between the listener and your handler:

```
func main() {
    lis, err := net.Listen("tcp", ":50051")  // gRPC convention: port 50051
```

```
    if err != nil {
        log.Fatal(err)
    }

    s := grpc.NewServer()
    musicpb.RegisterJukeboxServer(s, &jukeboxServer{
        catalog: map[string]*musicpb.Song{
            "1": {Id: "1", Title: "Monaco", Artist: "Bad Bunny", Bpm: 130},
            "2": {Id: "2", Title: "Despecha", Artist: "Rosalia", Bpm: 130},
        },
    })

    log.Printf("jukebox listening on %s", lis.Addr())
    log.Fatal(s.Serve(lis))  // blocks until the server stops
}
```

The key functions that you will use:

```
func NewServer(opt ...ServerOption) *Server     // new server; opts add interceptors, TLS
func (s *Server) Serve(lis net.Listener) error  // accept connections until Stop/GracefulStop
func (s *Server) GracefulStop()                 // stop accepting, drain in-flight RPCs
func (s *Server) Stop()                         // stop immediately, cancelling in-flight RPCs
```

Just like the `http.Server` from Chapter 15, prefer `GracefulStop` on shutdown so in-flight RPCs finish instead of being severed mid-call.

Calling a gRPC Server

The client side is generated too. `NewJukeboxClient` wraps a connection and returns a typed stub whose methods look exactly like local function calls — the network is hidden behind the generated code.

```
func NewClient(target string, opts ...DialOption) (*ClientConn, error)  // lazy connection
func NewJukeboxClient(cc grpc.ClientConnInterface) JukeboxClient        // generated typed stub
```

```
package main

import (
    "context"
    "log"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials/insecure"
    "example/musicpb"
)

func main() {
    conn, err := grpc.NewClient(
        "localhost:50051",
        grpc.WithTransportCredentials(insecure.NewCredentials()),  // plaintext; see TLS section
    )
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
```

```
    client := musicpb.NewJukeboxClient(conn)

    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    song, err := client.GetSong(ctx, &musicpb.GetSongRequest{Id: "1"})
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("%s by %s (%d BPM)", song.GetTitle(), song.GetArtist(), song.GetBpm())
    // Monaco by Bad Bunny (130 BPM)
}
```

Trap: `grpc.Dial` is the old constructor you will see in pre-2024 examples and tutorials. It is deprecated in favor of `grpc.NewClient`. The big behavioral difference: `Dial` connected eagerly (and `WithBlock` waited for the handshake), while `NewClient` connects lazily on the first RPC. Do not block startup waiting for a connection — let the first call establish it, and rely on deadlines to bound failures.

Tip: A `*grpc.ClientConn` is expensive to create and safe for concurrent use. Create one per backend at startup and share it across all goroutines, exactly as you reuse an `http.Client` (Chapter 15) or a `*sql.DB`. Do not open a new connection per request.

The Four Kinds of RPC

HTTP/2 carries multiple message frames over a single connection, so gRPC supports streaming in either direction — not just one-shot request/response. There are four method shapes, declared in the `.proto` with the `stream` keyword:

```
service Jukebox {
  rpc GetSong(GetSongRequest) returns (Song);              // unary
  rpc ListSongs(ListRequest) returns (stream Song);        // server streaming
  rpc AddSongs(stream Song) returns (AddSummary);          // client streaming
  rpc Party(stream SongRequest) returns (stream Song);     // bidirectional
}
```

You have already seen unary above. The streaming variants generate strongly typed stream handles backed by Go generics (Chapter 18): `grpc.ServerStreamingServer[T]`, `grpc.ClientStreamingServer[Req, Res]`, and so on. A stream is the RPC analog of a Go channel (Chapter 10): you `Send` and `Recv` values until `io.EOF`.

Server Streaming

The server sends many messages for one request — think "list" or "subscribe." On the server, you receive the request plus a stream and call `Send` in a loop:

```
func (s *jukeboxServer) ListSongs(
    req *musicpb.ListRequest,
    stream grpc.ServerStreamingServer[musicpb.Song],
) error {
    s.mu.Lock()
    defer s.mu.Unlock()
    for _, song := range s.catalog {
        if err := stream.Send(song); err != nil {  // one frame per song
            return err
```

```
        }
    }
    return nil  // returning ends the stream cleanly
}
```

The client receives until `io.EOF`:

```
stream, err := client.ListSongs(ctx, &musicpb.ListRequest{})
if err != nil {
    log.Fatal(err)
}
for {
    song, err := stream.Recv()
    if err == io.EOF {
        break  // server closed the stream --- normal end
    }
    if err != nil {
        log.Fatal(err)
    }
    log.Println(song.GetTitle())
}
```

Client Streaming

The client sends many messages and gets one response back — think "bulk upload." The client calls `Send` repeatedly, then `CloseAndRecv` to signal the end and read the single reply:

```
stream, err := client.AddSongs(ctx)
if err != nil {
    log.Fatal(err)
}
for _, song := range incoming {
    if err := stream.Send(song); err != nil {
        log.Fatal(err)
    }
}
summary, err := stream.CloseAndRecv()  // close the send side, get the response
if err != nil {
    log.Fatal(err)
}
log.Printf("added %d songs", summary.GetCount())
```

On the server, you `Recv` in a loop and `SendAndClose` once at the end:

```
func (s *jukeboxServer) AddSongs(
    stream grpc.ClientStreamingServer[musicpb.Song, musicpb.AddSummary],
) error {
    var count int32
    for {
        song, err := stream.Recv()
        if err == io.EOF {
            return stream.SendAndClose(&musicpb.AddSummary{Count: count})
        }
        if err != nil {
            return err
        }
```

```
        s.mu.Lock()
        s.catalog[song.GetId()] = song
        s.mu.Unlock()
        count++
    }
}
```

Bidirectional Streaming

Both sides stream independently over the one connection. Because the directions are independent, a common pattern is to read in one goroutine (Chapter 10) and write in another. Here the server simply echoes a matching song for every request it receives:

```
func (s *jukeboxServer) Party(
    stream grpc.BidiStreamingServer[musicpb.SongRequest, musicpb.Song],
) error {
    for {
        req, err := stream.Recv()
        if err == io.EOF {
            return nil  // client closed its send side
        }
        if err != nil {
            return err
        }
        s.mu.Lock()
        song := s.catalog[req.GetId()]
        s.mu.Unlock()
        if song != nil {
            if err := stream.Send(song); err != nil {
                return err
            }
        }
    }
}
```

The stream handle methods worth memorizing:

```
// server side
func (x ServerStreamingServer[T]) Send(*T) error                      // push to client
func (x ClientStreamingServer[Req, Res]) Recv() (*Req, error)         // pull from client
func (x ClientStreamingServer[Req, Res]) SendAndClose(*Res) error     // final reply + close
// client side
func (x ServerStreamingClient[T]) Recv() (*T, error)                  // pull from server
func (x ClientStreamingClient[Req, Res]) Send(*Req) error             // push to server
func (x ClientStreamingClient[Req, Res]) CloseAndRecv() (*Res, error) // close + read reply
func (x BidiStreamingClient[Req, Res]) CloseSend() error              // close send side only
```

Wut: A streaming RPC's `Send` does not guarantee the peer received the message — only that it was handed to the transport. Ordering is guaranteed by HTTP/2, but delivery is not — the only way to know the server *processed* your messages is to read the response (or, for server streams, to reach `io.EOF` without error). If the call fails with a network error instead — the connection drops, say — the server may have processed your messages and the response was lost, or it may never have received them at all. Always check the error returned by `CloseAndRecv`/`Recv`, not just the per-`Send` errors.

Deadlines and Metadata

Every RPC takes a `context.Context` (Chapter 12), and gRPC propagates its deadline *across the network*. When the client sets `context.WithTimeout`, the server's handler receives a context with the same deadline — so a slow handler can bail out early by watching `ctx.Done()` instead of doing work nobody is waiting for anymore. This is a big upgrade over plain HTTP, where you have to plumb timeouts through headers yourself.

```
ctx, cancel := context.WithTimeout(context.Background(), 200*time.Millisecond)
defer cancel()

_, err := client.GetSong(ctx, &musicpb.GetSongRequest{Id: "1"})
// if the server takes longer than 200ms, err has code DeadlineExceeded
```

Metadata is gRPC's equivalent of HTTP headers: a string-keyed multimap carried alongside the request, used for things like auth tokens, request IDs, and tracing. The client attaches it to the context; the server reads it back out.

```
import "google.golang.org/grpc/metadata"

// client: attach an auth token
ctx = metadata.AppendToOutgoingContext(ctx, "authorization", "Bearer hunter2")

// server: read it
func (s *jukeboxServer) GetSong(
    ctx context.Context, req *musicpb.GetSongRequest,
) (*musicpb.Song, error) {
    md, ok := metadata.FromIncomingContext(ctx)
    if ok {
        tokens := md.Get("authorization")  // []string
        _ = tokens
    }
    // ...
}
```

Errors and Status Codes

A gRPC handler returns an ordinary Go `error` (Chapter 9), but gRPC wants that error to carry a **status code** — a small enum that both sides understand regardless of language. Return `nil` for success; return a `status.Error` (or `status.Errorf`) with a code for failure.

```
func Errorf(c codes.Code, format string, a ...any) error  // build an error with a status code
func FromError(err error) (*Status, bool)                 // extract the status from an error

// server
return nil, status.Errorf(codes.InvalidArgument, "id must not be empty")

// client: inspect the code
song, err := client.GetSong(ctx, req)
if err != nil {
    st, _ := status.FromError(err)
    switch st.Code() {
    case codes.NotFound:
        log.Println("no existe esa cancion")
    case codes.DeadlineExceeded:
        log.Println("el servidor tardo demasiado")
    default:
```

```
        log.Printf("rpc failed: %v", st.Message())
    }
}
```

The codes you will reach for most, and their rough equivalents:

gRPC code	When to use	HTTP analog	Java analog
`OK`	success (return `nil`)	200	normal return
`InvalidArgument`	caller sent bad input	400	`IllegalArgumentException`
`NotFound`	resource does not exist	404	`NoSuchElementException`
`AlreadyExists`	create conflict	409	duplicate-key exception
`PermissionDenied`	authenticated but not allowed	403	`SecurityException`
`Unauthenticated`	missing/invalid credentials	401	`AuthenticationException`
`DeadlineExceeded`	RPC ran past its deadline	504	`TimeoutException`
`Unavailable`	backend down; safe to retry	503	`ConnectException`
`Internal`	a bug on the server	500	unchecked `RuntimeException`

Trap: Do not return a bare `errors.New("...")` or `fmt.Errorf("...")` from a handler. gRPC wraps any error without a status code as `codes.Unknown`, which tells the client nothing useful and defeats retry logic. Always wrap failures in `status.Errorf` with the most specific code that fits.

Interceptors

An **interceptor** is gRPC's middleware: a function that wraps every RPC so you can add logging, authentication, metrics, or recovery in one place. It is the same idea as the `func(http.Handler) http.Handler` middleware from Chapter 15, just with a gRPC-shaped signature. There are two flavors — one for unary RPCs and one for streams.

```
// unary server interceptor
func(ctx context.Context, req any, info *grpc.UnaryServerInfo,
     handler grpc.UnaryHandler) (resp any, err error)
```

A logging-plus-timing interceptor:

```
func logging(ctx context.Context, req any, info *grpc.UnaryServerInfo,
    handler grpc.UnaryHandler) (any, error) {

    start := time.Now()
    resp, err := handler(ctx, req)  // call the actual RPC
    log.Printf("%s took %s (err=%v)", info.FullMethod, time.Since(start), err)
    return resp, err
}

func main() {
    s := grpc.NewServer(grpc.ChainUnaryInterceptor(logging))  // chain runs left to right
    // ... register and serve
}
```

`grpc.ChainUnaryInterceptor(a, b, c)` composes several so the first listed runs outermost — the same ordering as the `chain` helper for HTTP middleware in Chapter 15. The client side has the mirror-image `grpc.WithChainUnaryInterceptor` for adding outbound auth headers, retries, and the like.

TLS

The `insecure.NewCredentials()` we used above sends everything in plaintext — fine for local development, never for production. gRPC rides on the same `crypto/tls` machinery as HTTPS (Chapter 15), exposed through the `credentials` package.

On the server, load a certificate and pass it as a `ServerOption`:

```
import "google.golang.org/grpc/credentials"

creds, err := credentials.NewServerTLSFromFile("cert.pem", "key.pem")
if err != nil {
    log.Fatal(err)
}
s := grpc.NewServer(grpc.Creds(creds))  // every connection is now TLS
```

On the client, trust the server's CA and dial with transport credentials:

```
creds, err := credentials.NewClientTLSFromFile("ca.pem", "")  // "" = use cert's hostname
if err != nil {
    log.Fatal(err)
}
conn, err := grpc.NewClient("jukebox.example.com:443", grpc.WithTransportCredentials(creds))
```

Tip: For service-to-service calls inside a cluster, most teams let a service mesh (Istio, Linkerd) terminate mTLS transparently, so application code can dial with `insecure` while the sidecar encrypts the wire. When you do TLS in-process, the rules from Chapter 15 apply unchanged — set a minimum version and never disable verification with `InsecureSkipVerify` in production.

Try It

Type this in and run it. gRPC needs a generated stub, so this is a three-step program: write the `.proto`, generate the Go, then run a single file that starts the server in a goroutine and calls it as a client — a full round trip in one process, no separate terminals.

First, `music.proto`:

```
syntax = "proto3";
package music.v1;
option go_package = "example/musicpb";

message GetSongRequest { string id = 1; }
message Song {
  string id     = 1;
  string title  = 2;
  string artist = 3;
}

service Jukebox {
  rpc GetSong(GetSongRequest) returns (Song);
}
```

Generate the code (see the toolchain section for installing the plugins):

```
protoc --go_out=. --go_opt=paths=source_relative \
       --go-grpc_out=. --go-grpc_opt=paths=source_relative \
       music.proto
```

Then `main.go`:

```
package main

import (
    "context"
    "log"
    "net"
    "time"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/credentials/insecure"
    "google.golang.org/grpc/status"
    "example/musicpb"
)

type server struct {
    musicpb.UnimplementedJukeboxServer
}

func (server) GetSong(_ context.Context, req *musicpb.GetSongRequest) (*musicpb.Song, error) {
    if req.GetId() != "1" {
        return nil, status.Errorf(codes.NotFound, "no song %q", req.GetId())
    }
    return &musicpb.Song{Id: "1", Title: "Todo De Ti", Artist: "Rauw Alejandro"}, nil
}

func main() {
    lis, _ := net.Listen("tcp", "localhost:0")  // :0 = pick any free port
    s := grpc.NewServer()
    musicpb.RegisterJukeboxServer(s, server{})
    go s.Serve(lis)            // run the server in the background
    defer s.GracefulStop()

    conn, err := grpc.NewClient(lis.Addr().String(),
        grpc.WithTransportCredentials(insecure.NewCredentials()))
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()
    client := musicpb.NewJukeboxClient(conn)

    ctx, cancel := context.WithTimeout(context.Background(), 2*time.Second)
    defer cancel()

    song, err := client.GetSong(ctx, &musicpb.GetSongRequest{Id: "1"})
    if err != nil {
        log.Fatal(err)
    }
    log.Printf("%s by %s", song.GetTitle(), song.GetArtist())
    // Todo De Ti by Rauw Alejandro
}
```

Try these modifications:

- Request id `"2"` and confirm the error code is `NotFound` via `status.FromError`.
- Add a `ListSongs(ListRequest) returns (stream Song)` RPC, regenerate, and stream two songs back to the client.
- Add a unary interceptor that logs `info.FullMethod` and the latency of each call.

Key Points

- gRPC is contract-first: you define messages and services in a `.proto` file and generate typed Go for both client and server, so the two cannot disagree about the wire format.
- Protocol Buffers encode data as compact binary; field *tag numbers*, not names, travel on the wire — never reuse or renumber a tag.
- `protoc` with `protoc-gen-go` and `protoc-gen-go-grpc` generates `*.pb.go` files; commit them and regenerate when the `.proto` changes. `buf` is the friendlier modern front end.
- Embed `UnimplementedXxxServer` in your server type for forward compatibility, then override the methods you handle.
- Use field getters (`req.GetId()`) when reading messages — they are `nil`-safe.
- `grpc.NewServer` + `Serve` runs a server (prefer `GracefulStop` on shutdown); `grpc.NewClient` + the generated `NewXxxClient` make calls. `grpc.Dial` is deprecated.
- Reuse a single `*grpc.ClientConn` across goroutines, like an `http.Client` or `*sql.DB`.
- Four RPC shapes: unary, server streaming, client streaming, bidirectional — streams expose `Send`/`Recv` and end at `io.EOF`.
- Deadlines from the client's `context` propagate across the network; metadata is gRPC's header map.
- Return failures as `status.Errorf(code, ...)` with a specific `codes` value — a bare `error` becomes the useless `codes.Unknown`.
- Interceptors are gRPC's middleware; TLS comes from the `credentials` package over the same `crypto/tls` stack as HTTPS.

Exercises

1. **Think about it:** Chapter 15 built a JSON-over-HTTP service for the same song catalog. For a public API consumed by third-party web and mobile clients you do not control, and for high-volume internal traffic between your own microservices, which would you reach for — REST/JSON or gRPC — and why? Consider browser support, human debuggability with `curl`, payload size, schema evolution, and streaming.

2. **What does this do?** A teammate writes this proto and regenerates, then is surprised the change "broke" old clients:

   ```
   // before
   message Song {
     string id    = 1;
     string title = 2;
   }
   // after
   message Song {
     string id     = 2;
     string title  = 1;
   }
   ```

   They only swapped the tag numbers, not the field names or types. What happens when a new server sends a `Song` to a client built from the *old* proto, and why?

3. **Calculation:** A unary RPC handler does 80 ms of work. A client calls it with `context.WithTimeout(ctx, 50*time.Millisecond)`. The server-side handler does not check `ctx.Done()` and runs to completion.

(a) What status code does the *client* observe?
(b) Does the server's handler still finish its 80 ms of work?
(c) What single change makes the server stop early when the deadline passes?

4. **Where is the bug?**

```go
func (s *jukeboxServer) ListSongs(
    req *musicpb.ListRequest,
    stream grpc.ServerStreamingServer[musicpb.Song],
) error {
    for _, song := range s.catalog {
        stream.Send(song)
    }
    return errors.New("done sending")
}
```

There are two distinct problems in this handler. Identify both and say what the client sees for each.

5. **Write a program:** Define a `.proto` with a `Library` service exposing `AddSongs(stream Song) returns (Summary)` where `Summary` has an `int32 count` and an `int32 total_bpm`. Implement the server so it accumulates the count and the sum of all `bpm` fields across the streamed songs, then returns the summary with `SendAndClose`. Write a client that streams three songs and prints the returned count and average BPM. Use `status.Errorf(codes.InvalidArgument, ...)` if any streamed song has an empty `id`.

The gRPC Authors. 2025. "Basics Tutorial: gRPC in Go." gRPC Documentation. https://grpc.io/docs/languages/go/basics/.