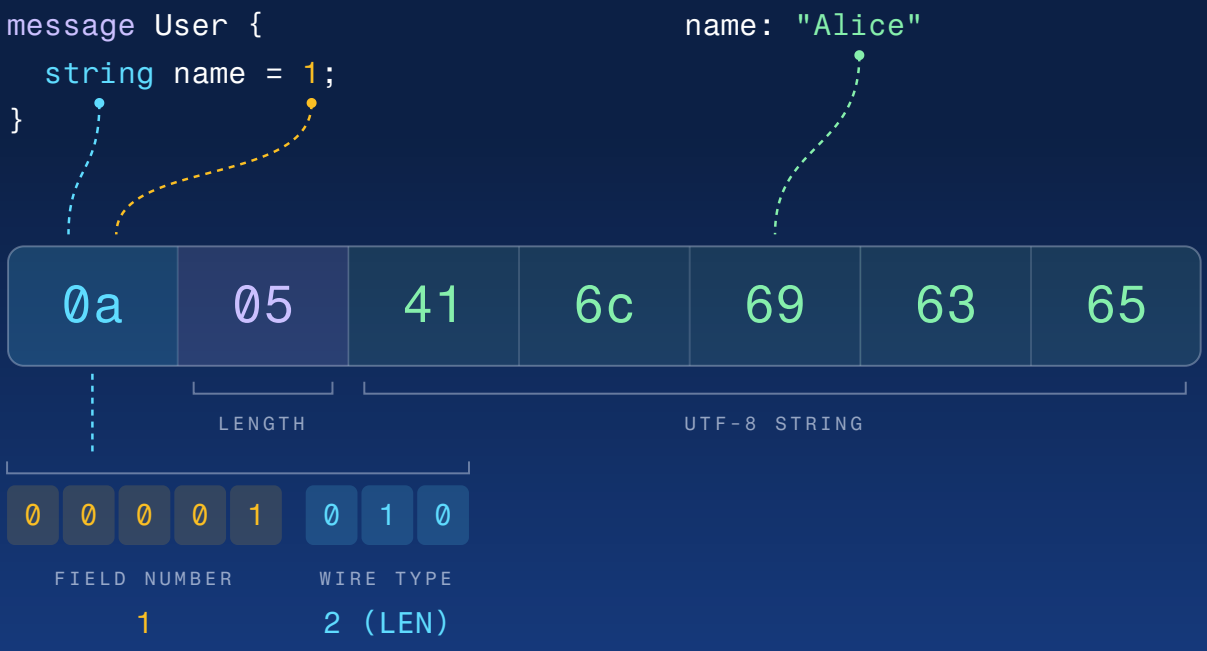
PROTOBUF.COM

Protobuf Explained

A visual guide to Protocol Buffers: the schema language, the data it describes, the code it generates, and the bytes it puts on the wire.



Contents

Foreword 4

Introduction 5

Basics 7

- Messages 8
- Fields 9
- Enums 10
- Packages 11
- Composition 12
- Repeated Fields 13
- Maps 14
- Oneof 15
- Generating Code 16
- Services 20

Advanced 30

- Imports 31
- The Any Type 33
- The Value Type 34
- Editions 35
- Standard Options 39
- Unknown Fields 40
- Schema Evolution 42
- Field Presence 44
- Required Fields 46
- Size Limits 47

Binary 49

- Binary 101 49
- Base-128 Varints 50
- ZigZag 52
- The Field Tag 53
- Wire Types 55
- Map Encoding 60
- Presence 60
- Parse Loop 61

Efficiency 65

Descriptors 68

Extending 71

Validation 75

Community 86

Conclusion 90

Appendix A. Types 91

Appendix B. Quiz 101

Appendix C. On the web 143

Foreword

This guide explains Protocol Buffers in a more casual tone. For per-language API docs or specifics, the official documentation at protobuf.dev is where you can go for that. What we try to add here is the reasoning: why a field has a number, why renaming one is safe on the wire and still breaks callers, what a varint actually is, and why any of that matters once a schema has been in production for a year.

One idea runs through all of it: a schema is a contract. Code generation turns the contract into types that compilers can check. The binary encoding uses the contract to save space on the wire so encoded protobuf messages can be pretty small. Field numbers, unknown fields, and reserved ranges let the two ends of a connection change at different times without breaking each other. And because the contract is written and checked in, a tool can compare two versions of it and tell you, before you ship, that you are about to break downstream clients.

We have opinions, and they show up throughout the book. The examples use edition 2024, because it's the latest well-supported edition of Protobuf. Don't know what editions are? Good news, we cover that, too. Tooling examples use the Buf CLI, because it is what we use.

You don't have to read this book in order. Basics and Advanced sections cover the language. Binary covers how messages are translated into binary. Descriptors, Extending, and Validation cover what you can build by using the schema itself as data. The appendices contain a full type reference and practice questions along with the answers.

Most of the diagrams in this book are interactive on the website. If you feel like a concept is not clicking for you from the book, we recommend giving the interactive elements on protobuf.com a try. If something here is wrong, or a page leaves you with a question, feel free to reach out at the Buf Slack: buf.build/links/slack.

The Buf team

Introduction

What is Protobuf?

Protocol Buffers (Protobuf) is a schema-driven format for serializing structured data.

SHARED CONTRACT

Write your type definitions once and use the same ones across every team and language that uses them.

TYPE SAFETY

Generate types from the schema so your compiler catches wrong field names and wrong types before anything runs.

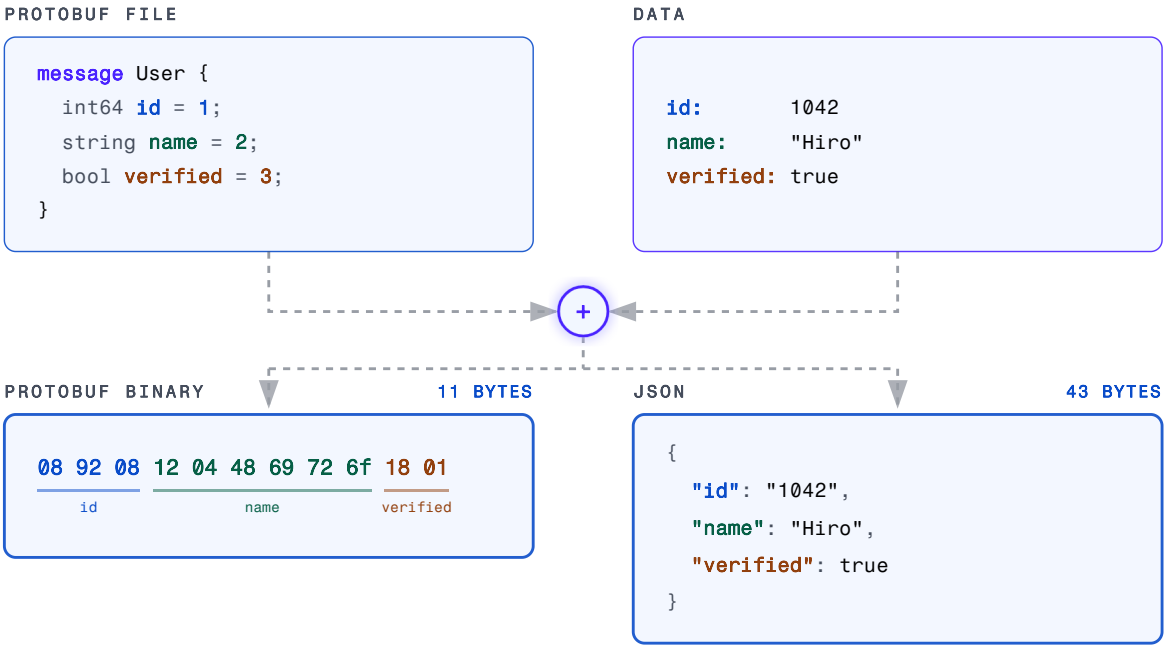
COMPATIBILITY

Add fields and keep already-deployed clients working, as long as you follow Protobuf's compatibility rules. A client built against an older schema ignores what it does not recognize in the binary encoding. ProtoJSON readers reject unknown fields by default, so they need more care.

How it works

Protocol Buffers, or "Protobuf", refers to two things: an Interface Definition Language (IDL) for writing schemas, and the encodings those schemas produce. This guide covers the two encodings you will use in practice. One is a compact binary form; the other is JSON, which uses a standardized JSON mapping. Both encodings are part of the specification, and either can go over the wire. Code generated from a schema can read and write both representations.

A JSON payload carries field names and readable values as text, so you (or any JSON tool) can read it without the schema. The binary encoding leaves the names out and identifies fields only by number, so a decoder needs the schema to know that field 2 is `name`.



The Binary section explains how those bytes are laid out.

From schema to code

A compiler reads the schema and generates native code for each language you target: structs in Go, types in TypeScript, classes in Java, and so on. The generated code gives you typed constructors, the binary serialization, and the JSON mapping, so a message behaves like a native type in your language.

Basics

A `.proto` file defines the shape of your data and assigns each field a stable numeric identity. Names keep the schema readable, and numbers keep the binary format compact and compatible over time.

Anatomy of a .proto file

This diagram breaks down the parts of a `.proto` file. Each label points to the topic that covers it in detail.



Messages

A message is a named group of fields. The `SearchRequest` here holds a `string` called `query` and two `int32` values for paging. Each field gets a number, and the encoded bytes identify fields by those numbers.

A field can hold another message. A `SearchResponse` can carry a list of `Result` messages. You can also nest one message inside another when the inner type only makes sense there. Either way, the declaration describes data. Messages don't inherit from each other, and the schema attaches no behavior to them.

What you get in generated code depends on the language and the generator. Go and Rust give you a `struct`. TypeScript gives you an object type. Java gives you a `class` with a builder. Most generators add accessors and serialization helpers on top, so your code works with field names rather than field numbers.

Adding a field is the common safe change, as long as you give it a new number and leave the existing ones alone. Code compiled against the older schema skips the number it does not recognize and keeps parsing the rest, so a server can send a new field before every client has been updated. That holds for the binary encoding. Standard ProtoJSON parsers reject unknown fields by default, so roll out new fields to ProtoJSON readers with more care, or turn on their option to ignore unknown fields. The full set of rules for what you can change is in Protobuf's guide to updating a message type.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 results_per_page = 3;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "query": "protobuf",
  "pageNumber": 2,
  "resultsPerPage": 25
}
```

Fields

Type

Every field has a type: `string`, `int32`, `bool`, a message, an enum. In statically typed languages, code that tries to put text in an `int32` field doesn't compile, and in Python the generated class raises an error as soon as you try. A JSON API can get typed models too, either hand-written or generated from an OpenAPI spec. A Protobuf schema also defines the wire format and the rules for changing it safely. Decoding is more forgiving, because the binary format doesn't carry enough type information to catch every mismatch. A string sent where the schema expects an `int32` parses without an error and lands in the message's unknown fields. Each language maps every Protobuf type onto one of its own, and the Types reference lists them all.

Name

The name is used by generated code and by people reading the schema. It becomes the accessor in Go or TypeScript and the key in the JSON mapping, and it is never written into the binary encoding. That is why you can rename a field without breaking binary readers. It is also why renaming still breaks JSON consumers and every line of code that used the old accessor, so treat a rename as a breaking change.

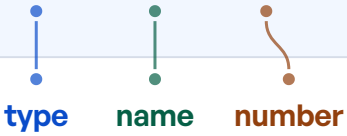
Number

Instead of sending a name like `"username"` over the wire, Protobuf identifies each field by its number. On the wire it is packed together with the field's wire type into a single value called the tag, and the Binary chapter shows exactly how. Because the number is the field's identity, it must never change once a message type is in use, and a number must never be reused for a different field: old data would decode as the new field without any error.

Tip: Numbers 1 through 15 take 1 byte to encode (including the field number and wire type). Use 1-15 for your most frequently sent fields.

Numbers go up to 536,870,911, except 19000 through 19999, which Protobuf reserves for its own use.

```
message User {
  string username = 1;
}
```



>_ JSON

```
{
  "username": "hiro"
}
```

Enums

An enum is a fixed set of named values, such as a status that can be active, deferred, or unspecified.

Enums are open by default in Edition 2024. Open means a client that reads a value its schema doesn't list will keep that value instead of dropping it, and includes that number when it re-serializes. This lets you add a value to an enum without breaking clients that were built against the older schema.

The first value of an open enum has to be zero. An unset field also reads as zero, so by convention the zero value is named `STATUS_UNSPECIFIED` and isn't used for real data. Reading `STATUS_UNSPECIFIED` doesn't tell you whether the field was set. A field explicitly set to zero is still present, and under edition 2024's explicit presence you check presence to tell the two apart.

Enum value names are scoped to the package or message that contains the enum, not to the enum itself. So if you have two enums in the same package, they can't both have a value named `ACTIVE`, and the compiler will reject the file if they do. This comes from C++, where the generated values land directly in the surrounding namespace. Prefixing every value with the enum name avoids the clash. Buf's style guide says to do this and `buf lint` checks it for you.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

enum Status {
  STATUS_UNSPECIFIED = 0;
  STATUS_ACTIVE = 1;
  STATUS_DEFERRED = 2;
}

message User {
  Status current_status = 1;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "currentStatus": "STATUS_ACTIVE"
}
```



Packages

Once a project has a few schemas in it you will start running into messages with the same name. Protobuf uses a `package` declaration to keep them apart.

The package name becomes part of the type name. The `Account` message in the example is really `demo.identity.v1.Account`. Other files use that full name to refer to the message, the Any type stores it in a type URL, and reflection uses it to look up the type at runtime. Inside the same package you can just write `Account` and leave the rest off.

Put the `v1` on the end from the start. It is the major version. When you need to make a breaking change later you write a new package ending in `v2` and leave the old one alone, so clients that are still on `v1` keep working. Buf's style guide covers this too, and `buf lint` enforces it, along with a related rule that the file path should match the package: `demo/identity/v1/account.proto`.

In generated code the package turns into whatever that language uses for namespacing: a namespace in C++, a package in Java and Go, and a module in TypeScript. Some languages don't map cleanly (Go import paths, for example), so there are file-level options like `go_package` to steer it.

Buf's files and packages reference goes through how packages, file paths, and directories relate in more detail.

```
edition = "2024";

// Defines the namespace. The file lives at
// demo/identity/v1/account.proto to match.
package demo.identity.v1;

import "demo/billing/v1/invoice.proto";

// Full name: demo.identity.v1.Account
message Account {
  string id = 1;

  // A type from another package needs its
  // qualified name.
  demo.billing.v1.Invoice last_invoice = 2;
}
```



Composition

A field's type can be another message, whether it is defined in the same file, elsewhere in the package, or in an import, and a message can be declared inside another when the inner type only makes sense there.

It also gives teams one definition of a shared shape. If `User`, `Event`, and `Office` all use the same `Location` message, the code generated for every language agrees on what a location contains.

On the wire an embedded message is length-delimited, which means its bytes are preceded by their length. A decoder that lacks the inner schema can still skip the whole sub-message and read the fields after it.

```
message Result {
  string url = 1;
  string title = 2;
}

message SearchResponse {
  Result top_result = 1;
}
```

>_ JSON

```
{
  "topResult": {
    "url": "https://protobuf.com/",
    "title": "Protobuf Explained"
  }
}
```

Repeated Fields

A `repeated` field holds a list: zero or more values of one type, in the order they were added. The example has a list of numbers and a list of strings. An empty list writes no bytes, so on the wire it looks the same as a field that was never set. A repeated field has no presence to check, and generated code gives you an empty list until you add to it. Order is preserved when the list is encoded and decoded, and duplicates are allowed.

Repeated scalars, such as `int32`, `float`, `bool`, and enums, are packed by default: the whole list travels as one length-prefixed block instead of a tag per element, which keeps long lists of numbers compact. Strings, bytes, and messages can't be packed and are written one element at a time. The Binary chapter shows both layouts.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

message Batch {
  repeated int32 ids = 1;
  repeated string tags = 2;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "ids": [1, 2, 3, 150]
}
```

{} Maps

A map field is a dictionary. The example uses string keys and string values for labels, integer keys for port numbers, and message values for entries that need more than one piece of data.

A key can be any integer type, a bool, or a string. Messages, enums, floats, and bytes can't be keys. A value can be any type, including another message, except another map or a repeated field; wrap those in a message if you need them.

Underneath, a map is a `repeated` message with a `key` field and a `value` field, so decoders written before maps existed can still read them. The Binary chapter shows the bytes.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

message Project {
  string name = 1;
  map<string, string> labels = 2;
  map<uint32, string> port_names = 3;
  map<string, Contributor> contributors = 4;
}

message Contributor {
  string name = 1;
  uint32 commit_count = 2;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "name": "protobuf.com",
  "labels": {
    "team": "docs",
    "tier": "public"
  }
}
```



Oneof

A `oneof` groups several fields where at most one can be set at a time. Setting one member clears the others. The group behaves like a tagged union, or a `variant` in languages that have one, and the generated code stores a single value.

A common use is a value that comes in one of several shapes, such as an `Event` that is a `ClickEvent`, a `HoverEvent`, or a `ScrollEvent`, with the oneof telling the reader which one arrived. Members always have explicit presence, so a reader can tell which member was set even when its value is zero. A oneof cannot contain repeated fields or maps.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

message ErrorStatus {
  string message = 1;

  oneof details {
    string stack_trace = 2;
    int32 error_code = 3;
  }
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "message": "invalid argument",
  "errorCode": 3
}
```

Generating Code

The usual way to work with Protobuf is to generate code. You run a compiler over your `.proto` files, and plugins emit the types, binary serializers, JSON mapping, and RPC stubs for your language.

>_ PROTO/INTRO/V1/USER.PROTO

```
edition = "2024";

package intro.v1;

message User {
  int64 id = 1;
  string name = 2;
  uint32 age = 3;
  float height_cm = 4;
  double weight_kg = 5;
  bool verified = 6;
}
```

What you get

Once code is generated from this schema, you can create `User` objects in your language with type checking and editor support, and convert them to compact binary or JSON for sending or storing.

Generating Code with buf

This guide uses the Buf CLI, which keeps generation declarative with a `buf.gen.yaml` file. Checking it into your repo lets everyone on the team generate the same output with one command. The `remote` plugins in this example run on the Buf Schema Registry, so there are no plugin binaries to install locally, and pinning their versions keeps everyone's output identical. The `buf.yaml` beside it marks `proto/` as the root of the schema files. Go code also needs an import path for each file; managed mode fills in `go_package` from the prefix, so you don't have to set it in the schema.

>_ BUF.YAML

```
version: v2
modules:
  - path: proto
```

>_ BUF.GEN.YAML

```yaml
version: v2
managed:
  enabled: true
  override:
    - file_option: go_package_prefix
      value: example.com/gen/go
plugins:
  - remote: buf.build/bufbuild/es:v2.14.0
    out: web/src/gen
    opt: target=ts
  - remote: buf.build/protocolbuffers/go:v1.36.12
    out: gen/go
    opt: paths=source_relative
```

>_ GENERATE CODE

```
$ buf generate
```

✓ GENERATION COMPLETE — 2 FILE(S) CREATED

web/src/gen/intro/v1/user_pb.ts
TypeScript types & binary/JSON codec metadata

gen/go/intro/v1/user.pb.go
Go getters, setters, builders, ProtoReflect, & binary marshal/unmarshal

Generating with `protoc`

`protoc` is the reference compiler for Protobuf. Plugins (like `protoc-gen-es`) are executables that `protoc` finds on your `PATH`, or at the path you give it with `--plugin`. Without managed mode, the Go import path comes from a `go_package` option in the file or an `M` flag on the command line. `protoc` creates the folders inside each output directory but not the output directory itself, so make those first.

>_ INSTALL PLUGINS

```
$ npm install @bufbuild/protobuf
$ npm install --save-dev @bufbuild/protoc-gen-es
$ go get google.golang.org/protobuf@v1.36.12
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.36.12
```

>_ GENERATE CODE

```
$ mkdir -p web/src/gen gen/go
$ protoc -I proto \
    --plugin=protoc-gen-es=node_modules/.bin/protoc-gen-es \
    --es_out=web/src/gen --es_opt=target=ts \
    --go_out=gen/go --go_opt=paths=source_relative \
    --go_opt=Mintro/v1/user.proto=example.com/gen/go/intro/v1 \
    proto/intro/v1/user.proto
```

Using the Generated Code

With code generated by protobuf-es, the schema becomes a native TypeScript API.

>_ WEB/SRC/MAIN.TS

```
import { create, toBinary, toJsonString } from "@bufbuild/protobuf";
import { UserSchema } from "./gen/intro/v1/user_pb";

const user = create(UserSchema, {
  id: 1042n,
  name: "cyber_ninja",
  age: 28,
  verified: true,
});

const bytes = toBinary(UserSchema, user);
const json = toJsonString(UserSchema, user);

console.log("JSON Output:", json);
console.log("Binary Output:", bytes);
```

DEVELOPER CONSOLE main.ts:12

```
> JSON Output: "{"id":"1042","name":"cyber_ninja","age":28,"verified":true}"

> Binary Output: Uint8Array(20) [ 8, 146, 8, 18, 11, 99, 121, 98, 101, 114, 95,
  110, 105, 110, 106, 97, 24, 28, 48, 1 ]
```

DIFFERENT LANGUAGES AND RUNTIMES

The same schema-first workflow applies across supported languages, but import paths, package names, generated types, and runtime APIs differ by language. Getting-started tutorials for each language are listed under Further Reading.

Services

A `service` declares `rpc` methods: a name, a request message, and a response message for each. Protobuf only defines the interface. An RPC framework such as gRPC or ConnectRPC turns the declaration into clients and servers, picks the transport, and handles the networking.

A method is unary when one request gets one response. Marking either side with `stream` lets that side send a sequence of messages within a single call: a server stream for a feed of results, a client stream for an upload, or both at once for a chat. The example declares one method of each kind.

In Go, the message plugin covered in Generating Code only produces message and enum types. Clients and servers come from an additional framework-specific plugin: `protoc-gen-go-grpc` for gRPC, and `protoc-gen-connect-go` for ConnectRPC. In TypeScript, `protoc-gen-es` also generates the service definition that Connect-ES builds clients from.

```
service UserService {
  // Unary: One request, one response
  rpc GetUser(GetUserRequest) returns (User);

  // Server Stream: One request, many responses
  rpc ListUsers(ListUsersRequest) returns (stream User);

  // Client Stream: Many requests, one response
  rpc UploadAvatar(stream AvatarChunk) returns (UploadAvatarResponse);

  // Bidirectional Stream: Real-time chat
  rpc Chat(stream Message) returns (stream Message);
}
```



gRPC

This guide covers two RPC frameworks: gRPC, in this section, and ConnectRPC, in the next section. Both turn a `service` definition into a typed client and server, and both send Protobuf messages over HTTP. They differ in the HTTP versions they need, which clients can call them, and how much comes built in.

STRENGTHS

- The most widely deployed Protobuf RPC framework, with official implementations for nearly every major language.
- Service meshes, proxies like Envoy, and cloud load balancers understand it natively.
- Deadlines, health checks, server reflection, and client-side load balancing come built in.

TRADE-OFFS

- Requires HTTP/2 end to end, including trailers. Some L7 proxies and load balancers don't pass HTTP/2 trailers through.
- Browsers can't make gRPC calls directly, so web apps need gRPC-Web and a translating proxy in front of the server.
- The wire format is binary framing, so you debug with tools like grpcurl rather than curl or browser devtools.

More details and examples at grpc.io.

The example service

The examples in this section and the next use this schema. One `buf generate` produces everything: message types from `protoc-gen-go`, the gRPC server and client from `protoc-gen-go-grpc`, the ConnectRPC handler and client from `protoc-gen-connect-go`, and the

TypeScript messages and service definition from `protoc-gen-es`.

>_ PROTO/USER/V1/USER.PROTO

```
edition = "2024";

package user.v1;

service UserService {
  rpc GetUser(GetUserRequest) returns (GetUserResponse);
}

message GetUserRequest {
  int64 id = 1;
}

message GetUserResponse {
  User user = 1;
}

message User {
  int64 id = 1;
  string name = 2;
}
```

>_ BUF.GEN.YAML

```yaml
version: v2
managed:
  enabled: true
  override:
    - file_option: go_package_prefix
      value: example.com/gen/go
plugins:
  - remote: buf.build/protocolbuffers/go:v1.36.12
    out: gen/go
    opt: paths=source_relative
  - remote: buf.build/grpc/go:v1.6.2
    out: gen/go
    opt: paths=source_relative
  - remote: buf.build/connectrpc/go:v2.0.2
    out: gen/go
    opt: paths=source_relative
  - remote: buf.build/bufbuild/es:v2.14.0
    out: web/src/gen
    opt: target=ts
```

gRPC between microservices

A users service implements the generated interface and registers it with a gRPC server. Any other service in the cluster dials it and calls `GetUser` like a local function, with the request and response types checked by the compiler on both sides. Embedding `UnimplementedUserServiceServer` keeps the server compiling when new methods are added to the schema.

>_ USERS/MAIN.GO

```go
type userServer struct {
  userv1.UnimplementedUserServiceServer
}

func (s *userServer) GetUser(
  ctx context.Context,
  req *userv1.GetUserRequest,
) (*userv1.GetUserResponse, error) {
  return userv1.GetUserResponse_builder{
    User: userv1.User_builder{
      Id:   proto.Int64(req.GetId()),
      Name: proto.String("Ada"),
    }.Build(),
  }.Build(), nil
}

func main() {
  listener, err := net.Listen("tcp", ":8080")
  if err != nil {
    log.Fatal(err)
  }
  server := grpc.NewServer()
  userv1.RegisterUserServiceServer(server, &userServer{})
  log.Fatal(server.Serve(listener))
}
```

>_ ORDERS/USERS_CLIENT.GO

```
conn, err := grpc.NewClient(
  "users.internal:8080",
  grpc.WithTransportCredentials(insecure.NewCredentials()),
)
if err != nil {
  return err
}
defer conn.Close()

users := userv1.NewUserServiceClient(conn)
req := userv1.GetUserRequest_builder{Id: proto.Int64(42)}.Build()
res, err := users.GetUser(ctx, req)
if err != nil {
  return err
}
fmt.Println(res.GetUser().GetName())
```



ConnectRPC

ConnectRPC uses the same service definitions and generated message types as gRPC. Its servers answer three protocols on one port: its own Connect protocol, gRPC, and gRPC-Web.

STRENGTHS

- The Connect protocol works over HTTP/1.1 or HTTP/2. A unary call is a plain POST with a JSON or binary body.
- Browsers can call a Connect server directly, with no proxy in between.
- A Connect server also speaks gRPC over HTTP/2 and gRPC-Web over either version, so existing gRPC clients keep working.
- Builds on each language's standard HTTP stack, such as net/http in Go, so existing middleware applies.

TRADE-OFFS

- Supports fewer languages than gRPC.
- Less built-in infrastructure: features like xDS load balancing are left to the surrounding platform.

More details and examples at connectrpc.com.

A ConnectRPC server and client

With connect-go, the handler has the same shape as the gRPC one, but it is mounted on a standard `net/http` server. That means standard Go HTTP libraries and middleware work with ConnectRPC without changes. The server answers Connect, gRPC, and gRPC-Web on one port, so the gRPC client from the previous section can call it unchanged. The ConnectRPC client takes a standard `http.Client` too.

>_ USERS/MAIN.GO

```
type userServer struct {
  userv1connect.UnimplementedUserServiceHandler
}

func (s *userServer) GetUser(
  ctx context.Context,
  req *userv1.GetUserRequest,
) (*userv1.GetUserResponse, error) {
  return userv1.GetUserResponse_builder{
    User: userv1.User_builder{
      Id:   proto.Int64(req.GetId()),
      Name: proto.String("Ada"),
    }.Build(),
  }.Build(), nil
}

func main() {
  server := connect.NewServer()
  userv1connect.RegisterUserServiceHandler(server, &userServer{})
  mux := http.NewServeMux()
  connecthttp.Mount(mux, server)

  protocols := new(http.Protocols)
  protocols.SetHTTP1(true)
  protocols.SetUnencryptedHTTP2(true)
  httpServer := &http.Server{
    Addr:      ":8080",
    Handler:   mux,
    Protocols: protocols,
  }
  log.Fatal(httpServer.ListenAndServe())
}
```

>_ ORDERS/USERS_CLIENT.GO

```
client := connect.NewClient(
  connecthttp.NewTransport(http.DefaultClient,
"http://users.internal:8080"),
)
users := userv1connect.NewUserServiceClient(client)

req := userv1.GetUserRequest_builder{Id: proto.Int64(42)}.Build()
res, err := users.GetUser(ctx, req)
if err != nil {
  return err
}
fmt.Println(res.GetUser().GetName())
```

ConnectRPC in the browser

With Connect-ES, you call the service from TypeScript in the browser. Here it calls the Go server above. The browser speaks the Connect protocol, which the server understands directly, so unlike gRPC+gRPC-Web there is no proxy in between. Each call shows up in devtools as an ordinary POST. Because a unary call is just HTTP with a JSON body, curl works too.

>_ WEB/SRC/USER.TS

```
import { createClient } from "@connectrpc/connect";
import { createConnectTransport } from "@connectrpc/connect-web";
import { UserService } from "./gen/user/v1/user_pb";

const transport = createConnectTransport({
  baseUrl: "https://api.example.com",
});
const users = createClient(UserService, transport);

const res = await users.getUser({ id: 42n });
console.log(res.user?.name);
```

```
>_ TERMINAL

$ curl https://api.example.com/user.v1.UserService/GetUser \
    -H "Content-Type: application/json" \
    -d '{"id": "42"}'

{"user":{"id":"42","name":"Ada"}}
```

Further reading: Basics

Fields

Tip of the week #1: Field names are forever
https://buf.build/blog/totw-1-field-names

Why renaming a field is safe on the wire but still a breaking change almost everywhere else.

Tip of the week #9: Some numbers are more equal than others
https://buf.build/blog/totw-9-some-numbers-are-more-equal-than-others

How field numbers pack into tags, and why numbers 1 through 15 should go to your most common fields.

Enums

Protobuf Tip #3: Enum names need prefixes
https://buf.build/blog/totw-3-enum-names-need-prefixes

Enum values live in their parent scope, so unprefixed names collide.

Protobuf Tip #6: The subtle dangers of enum aliases
https://buf.build/blog/totw-6-dangers-of-enum-aliases

What happens when two enum names share one number.

Packages

The Protobuf Language Specification
https://buf.build/blog/protobuf-language-specification

The complete spec, including how packages scope names and how references resolve.

Generating Code

The modern Protobuf workflow
https://buf.build/blog/the-modern-protobuf-workflow

How linting, code generation, and breaking-change checks fit together in a schema-first workflow.

Protobuf-ES: the TypeScript runtime we all deserve
https://buf.build/blog/protobuf-es-the-protocol-buffers-typescript-javascript-runtime-we-all-deserve

The design behind the generated TypeScript that powers the site's demos.

Getting started in Go
https://protobuf.dev/getting-started/gotutorial/

The official Go tutorial: generate code from a .proto and use it to write and read messages.

Getting started in Python
https://protobufpy.com/

Generating Python code from a schema and using the Protobuf runtime API.

Getting started in TypeScript
https://protobufes.com/

The Protobuf-ES guide: the same runtime the site's interactive demos use.

Getting started in Java
https://protobuf.dev/getting-started/javatutorial/

The official Java tutorial.

Getting started in Kotlin
https://protobuf.dev/getting-started/kotlintutorial/

The official Kotlin tutorial.

Getting started in C++
https://protobuf.dev/getting-started/cpptutorial/

The official C++ tutorial.

Getting started in C#
https://protobuf.dev/getting-started/csharptutorial/

The official C# tutorial.

Getting started in Dart
https://protobuf.dev/getting-started/darttutorial/

The official Dart tutorial.

Services

gRPC
https://grpc.io/

The original Protobuf RPC framework, with implementations for most major languages.

ConnectRPC
https://connectrpc.com/

A CNCF RPC framework built on Protobuf services, compatible with gRPC clients and servers. Its origin story is in Connect: A better gRPC.

Advanced

Beyond basic messages, Protobuf has tools for splitting schemas across files, carrying dynamic data, and evolving types in production without breaking callers. Descriptors, extensions, and validation are covered in the chapters that follow.

Anatomy of an evolving schema

This diagram breaks down the features this chapter covers. Each label points to the topic that covers it in detail.



Imports

The `import` statement brings definitions from another `.proto` file into scope. Paths resolve from your module root rather than relative to the file doing the importing. Always write the full path from that root, so the same import string refers to the same file no matter which `.proto` it appears in. `protoc` works the same way once you tell it where the roots are: every `-I` (or `--proto_path`) flag names one, and if two files spell the same import differently, such as `import "proto/user.proto"` in one and `import "user.proto"` in another, `protoc` treats them as two different types.

Often the definitions you want already live in another project. You can vendor them: copy that team's `.proto` files into your repository, then copy them again whenever they change. Vendored copies don't record which version they came from.

With a schema registry, the other project is a declared dependency. The Buf Schema Registry pins each one to an exact version in a lockfile, so every checkout resolves imports identically, the same way npm and Cargo handle package dependencies. The `buf.yaml` file in the example declares `buf.build/bufbuild/protovalidate` as a dependency, so `service.proto` can import `buf/validate/validate.proto`.

auth/v1/service.proto

```
edition = "2024";
package auth.v1;

import "buf/validate/validate.proto";
import "common/v1/user.proto";
import "google/type/datetime.proto";

message LoginRequest {
  string email = 1 [(buf.validate.field).string.email = true];
}

message LoginResponse {
  common.v1.User user = 1;
  string session_token = 2;
  google.type.DateTime expires_at = 3;
}
```

common/v1/user.proto

```
edition = "2024";
package common.v1;

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

buf.yaml

```
version: v2
modules:
  - path: proto
deps:
  - buf.build/bufbuild/protovalidate
  - buf.build/googleapis/googleapis
```

terminal

```
buf dep update
buf build
```

The Any Type

`Any` holds a message whose type isn't known when the schema is compiled.

`google.protobuf.Any` embeds an arbitrary serialized Protobuf message along with a URL that identifies its type, such as `type.googleapis.com/mypackage.MyMessage`.

In ProtoJSON, the type URL becomes a `@type` property alongside the embedded message's fields. A parser reads that property first to learn which message type the rest of the object holds. Decoding the embedded message requires that type's generated code or descriptor, usually looked up by URL in a type registry, and converting an `Any` to JSON fails when the type isn't registered.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

import "google/protobuf/any.proto";

message Envelope {
  string kind = 1;
  google.protobuf.Any payload = 2;
}

message Note {
  string text = 1;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "kind": "note",
  "payload": {
    "@type": "type.googleapis.com/demo.v1.Note",
    "text": "hello"
  }
}
```

The Value Type

`google.protobuf.Value` and `google.protobuf.Struct` cover data that isn't a Protobuf message at all, just arbitrary JSON you don't want to model up front. `Any`, by contrast, carries a Protobuf message whose type isn't known until runtime.

A `Value` can be a null, a number, a string, a boolean, a struct of further values, or a list of them. Those six cases are exactly JSON's own types: null, number, string, boolean, object, and array.

A field typed as `Value` or `Struct` has no schema checking, so generated code can't tell you what's inside. Keep them for the places that need them, such as integrating with a schemaless NoSQL database or passing along a metadata blob nobody wants to write a schema for.

`Any`, `Value`, and `Struct` all fall under Well-Known Types. The Types reference lists the rest, including `Timestamp` and `Duration`.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

import "google/protobuf/struct.proto";

message Config {
  google.protobuf.Struct settings = 1;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "settings": {
    "theme": "dark",
    "retries": 3,
    "beta": true
  }
}
```

Editions

Protobuf Editions replaces the old `proto2` and `proto3` syntax switch with individual feature settings. Instead of upgrading a whole file at once, you can change behaviors one at a time:

- **Field Presence:** whether a reader can tell a field that was never set apart from one set to its default value. Choose `IMPLICIT` (the proto3 behavior) or `EXPLICIT` (the proto2 behavior); the Field Presence page covers the tradeoff.
- **Enum Type:** where a value the enum does not declare ends up. `OPEN` keeps it in the field; `CLOSED` moves it to the unknown fields.
- **Repeated Encoding:** Standardize on `PACKED` (for efficiency) or `EXPANDED` (for compatibility).

Under `proto2` and `proto3`, these behaviors came as a set with the syntax keyword at the top of the file, so migration was all or nothing. Moving a file to `proto3` opened its closed enums, dropped its custom field defaults, and removed explicit presence from its singular scalars.

Editions makes each of those a separate feature. A feature can be set for a whole file and then overridden on the elements it applies to, such as one enum for enum type or one field for field presence, so a schema can adopt one new behavior without taking the others. New behavior ships as a new feature with a per-edition default, so there is no "proto4" forcing another migration of every behavior at once.

Three editions have shipped so far: edition 2023, which unified proto2 and proto3; edition 2024, which added symbol visibility, `import option`, and naming-style enforcement; and edition 2026 (protobuf 36.x), which tightens the naming rules, defaults symbol visibility to strict, and adds custom JSON names for enum values. The editions overview and the feature settings reference track exactly what each edition changes.

```
edition = "2024";

package demo.v1;

// Set for the whole file...
option features.enum_type = CLOSED;

enum Unit {
  UNIT_UNSPECIFIED = 0;
  UNIT_CELSIUS = 1;
}

enum Status {
  // ...and overridden on one enum.
  option features.enum_type = OPEN;

  STATUS_UNSPECIFIED = 0;
  STATUS_ACTIVE = 1;
}
```

Edition by edition

No edition changes the wire format. A field is encoded the same way under proto3, edition 2023, and edition 2026, and a reader on any of them can parse what a writer on any other produced. The bytes can still differ where the defaults do: a field explicitly set to zero is written under explicit presence, the editions default, and skipped under proto3's implicit presence. Editions change defaults and language rules instead, such as how an unset field behaves, which schemas the compiler accepts, and the shape of the generated code. Moving a file forward is an edit to its first line, plus explicit features wherever the new defaults would change the file's behavior; the compiler reports anything it cannot carry over.

2008 AND 2016

Before editions: proto2 and proto3

The syntax line at the top of a file chose between two fixed sets of behavior.

- **proto2** has explicit presence on every singular field, closed enums, `required`, custom default values, extensions on any message, and groups.
- **proto3** drops `required`, custom defaults, and general extensions. Scalar fields get implicit presence, enums are open, repeated scalars are packed, and the JSON mapping is standard. The `optional` keyword came back in 2021 to give a proto3 scalar explicit presence again.

THE FIRST EDITION

Edition 2023

Replaces the syntax switch with features. Each behavior proto2 and proto3 disagreed on becomes a setting with a default, and the default can be changed for a whole file or on the individual elements each feature applies to.

- `features.field_presence`: `EXPLICIT` by default, so every singular field can tell unset from zero without `optional`. `IMPLICIT` restores the proto3 behavior; `LEGACY_REQUIRED` is the editions form of the proto2 `required` label.
- `features.enum_type`: `OPEN` by default, the proto3 behavior. `CLOSED` keeps proto2 enums working as they did.
- `features.repeated_field_encoding`: `PACKED` by default; `EXPANDED` writes a tag per element.
- `features.utf8_validation` (`VERIFY`), `features.message_encoding` (`LENGTH_PREFIXED`, with `DELIMITED` standing in for proto2 groups), and `features.json_format` (`ALLOW`) cover the rest.
- The `optional` and `required` keywords are gone, and extensions are allowed on any message again.

THE EDITION THIS GUIDE WRITES IN

Edition 2024

Keeps the 2023 presence, enum, and packing defaults, adds control over what a file exposes to the files that import it, and starts enforcing the style guide's names.

- Symbol visibility: `export` and `local` modifiers on messages and enums, and `features.default_symbol_visibility` to set the default. Out of the box, top-level types are importable and nested types are local to their file.
- `import option`, for a file you import only to use the custom options it defines. Its types do not come into scope, and its descriptors are not needed at runtime.
- Naming-style enforcement, `features.enforce_naming_style`, is on: fields in `lower_snake_case`, enum values in `UPPER_SNAKE_CASE`, messages, enums, and services in `PascalCase`, with the compiler rejecting anything else. A file with older names can set it to `STYLE_LEGACY` until the names are updated.
- Presence, enums, and packing behave the same as in 2023, but Edition 2024 changes symbol visibility and some generated APIs. Nested types become local, so other files that refer to them stop compiling. Some generated code changes too, such as Go switching to the Opaque API and C++ string fields returning views. Each change has a feature that restores the 2023 behavior, such as `option features.(pb.go).api_level = API_OPEN;`, and the feature settings reference lists them.

PROTOBUF 36

Edition 2026

Tightens the 2024 defaults, and tidies the JSON mapping for enums.

- Naming style tightens: a field can no longer be named so that it collides with a generated accessor, such as `has_x`, `get_x`, or `clear_x`, and no field can be named `descriptor`.
- Symbol visibility defaults to strict, so a type is importable only when it is explicitly exported.
- Enum values can carry a custom JSON name, just as fields do with `json_name`.

How the language got here

Protobuf has been public since 2008. These are the releases that changed the language, from proto2 through editions.



INTERNAL 2001

Google Internal Launch

Protobuf was born inside Google to solve the overhead of XML. It became widely used for internal communication.

protobuf.dev/overview/#history



PROTO2 2008

Public Open Source

First public release, bringing the internal language's "required" fields and extension mechanism into the open.

opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html



PROTO3 2016

Proto3 Released

Removed "required" fields, standardized the JSON mapping, and dropped explicit presence for singular scalar fields.

github.com/protocolbuffers/protobuf/releases/tag/v3.0.0

EDITIONS 2024

Editions Arrive

Moves the language from syntax "versions" to individually settable "features", so it can add behavior without waiting on a new syntax release. Three have shipped so far: 2023, 2024, and 2026.

github.com/protocolbuffers/protobuf/releases/tag/v27.0

Standard Options

An option is a setting attached to a schema element: a file, a message, a field, an enum, or a service. The options on this page change what the compiler generates and how data maps to other formats, and they never change the wire format. (Editions features use the same syntax and are an exception. Field presence, for example, changes what gets written.) A file with `go_package` set and the same file without it encode the same bytes.

The most common ones:

- `go_package` and `java_package`: where the generated Go and Java code goes.
- `optimize_for`: trades generated-code size for speed in C++. `SPEED`, `CODE_SIZE`, or `LITE_RUNTIME`.
- `[deprecated = true]`: marks a field so generated code warns when it is used.
- `[json_name = "uid"]`: changes the key the JSON mapping uses, as the example shows.

The full set lives in `descriptor.proto`, and the Custom Options page shows how to define your own.

```
edition = "2024";

option go_package = "github.com/example/v1";
option java_package = "com.example.v1";
option optimize_for = SPEED;

message User {
  string user_id = 1 [json_name = "uid"];
  string old_field = 2 [deprecated = true];
}
```

>_ JSON

```
{
  "uid": "01H8XGJWBWBAQ4",
  "oldField": "still serialized"
}
```

Unknown Fields

When a parser finds a field number its schema doesn't define, it keeps the data. The wire format already says how many bytes the value takes up, so the parser reads them, stores them as an unknown field, and keeps going. When the message is serialized again, the unknown fields are written out with it. The data is the same, but the bytes may differ. C++, for example, stores an unknown varint as a decoded integer and re-encodes it.

Much of Protobuf's compatibility story depends on this. An old client can accept a message from a new server, touch the fields it knows about, and pass the message along without stripping fields added after it was compiled. The same goes for proxies and queues sitting between services that upgrade on different schedules.

Unknown fields only survive on the *decoded message*. They are kept when that message is serialized again, or copied with a message-aware operation like `proto.Clone` in Go or `CopyFrom` in Python, and many services do neither. If a service unpacks a request into a domain object and builds a fresh response (or even a fresh copy of the same type, field by field), the new message has no unknown fields at all. Converting to JSON drops them too, because the JSON mapping has no way to name a field the reader doesn't know. So unknown fields make it through services that forward messages as they are, and get lost at any service that rebuilds them.

Be careful with this at the edge of your system. A payload from outside can carry unknown fields your validation never inspected, and preservation passes them on to internal services that do understand them. Many teams that expose Protobuf APIs publicly drop unknown fields at the boundary as a precaution. Most runtimes give you a discard option when parsing.

USER.PROTO (WRITER, V2)

```
message User {
  string id = 1;
  string email = 2;
}
```

sending

```
{ "id": "u_1", "email": "a@b.c" }
```

↓ serialize

ON THE WIRE

0a 03 75 5f 31 — field 1 — id

12 05 61 40 62 2e 63 — field 2 — email

↓ parsed by a reader compiled against v1

USER.PROTO (READER, V1)

```
message User {
  string id = 1;
}
```

parsed into a field

id = "u_1"

kept as an unknown field

12 05 61 40 62 2e 63

↓ then the reader passes the message along

FORWARDS THE DECODED MESSAGE

send(user)

0a 03 75 5f 31 12 05 61 40 62 2e 63

unknown field serialized with it

REBUILDS THE MESSAGE

send(toProto(toDomain(user)))

0a 03 75 5f 31

~~12 05 61 40 62 2e 63~~

field 2 dropped: a new instance has no unknown fields

Unknown fields live on the decoded message. Forwarding it, or copying it with a message-aware clone or merge, keeps them. Rebuilding it field by field, or converting it to JSON, drops them.

Schema Evolution

Levels of Breakage

Because the schema is written down, breaking changes can be caught before they ship. A CI check like `buf breaking` compares your schema against the previous version and flags the exact line.

Not every change breaks in the same way. Renaming a field, for example, is invisible on the binary wire but breaks JSON clients. Breaking changes fall into four categories:

- **WIRE:** The most severe level. Changing a field number or switching to an incompatible type (such as `string` to `int32`) corrupts data when old and new endpoints decode each other's messages. Never do this.
- **WIRE_JSON:** Breaks clients that use the JSON encoding. Renaming a field is safe on the binary wire, but clients expecting the old JSON key will fail. Setting `[json_name="old_name"]` keeps the old key. Even adding a field needs care here: binary readers skip fields they don't know, but standard ProtoJSON parsers reject unknown keys by default, so a JSON client on the old schema fails once the new field starts appearing, unless it was set to ignore unknown fields.
- **PACKAGE:** Breaks generated code within a package. Changing a type in a wire-compatible way (such as `int32` to `int64`) keeps old and new endpoints decoding each other's messages, as long as values stay in the old range until every reader is upgraded: an old reader truncates anything bigger. Builds still fail when developers regenerate until they update their types.
- **FILE:** The strictest level. It protects generated code down to the individual file. Moving a message to another file can break code that imports it from the old location.

>_ BEFORE

```
edition = "2024";
package api.v1;

message User {
  string id = 1;
  int32 age = 2;
  string display_name = 3;
}
```

>_ AFTER

```
edition = "2024";
package api.v1;

message User {
  // [WIRE] breakage: type changed from string
  int32 id = 1;

  // [PACKAGE] breakage: source code type change
  int64 age = 2;

  // [WIRE_JSON] breakage: JSON key changed
  string full_name = 3;
}
```

Deprecation

Protobuf identifies data on the wire by field number, so removing a field takes more than deleting its line. Once a schema has been in production, older clients, queues, and databases still hold messages that use the number, and a new field that reused it would decode their data as its own. A field is retired in stages instead, shown in the panels below.

First, mark it `[deprecated = true]`. Nothing changes on the wire, but generated code picks up the annotation, `@Deprecated` in Java for instance, and editors start warning anyone who still writes to it. Next, use whatever metrics you have to confirm nothing reads or writes the field anymore. Then remove it, and in the same change add its number and its name to `reserved` statements, so the compiler rejects any later attempt to use either. Keep the reservation forever, since old data with that number can turn up at any time.

>_ STEP 1: THE ORIGINAL SCHEMA

```
message Product {
  int32 price_cents = 1;
}
```

>_ STEP 2: DEPRECATE THE OLD FIELD, ADD THE NEW ONE

```
message Product {
  int32 price_cents = 1 [deprecated = true];
  int64 price_micros = 2;
}
```

>_ STEP 3: REMOVE THE OLD FIELD AND RESERVE ITS ID/NAME

```
message Product {
  reserved 1;
  reserved price_cents;

  int64 price_micros = 2;
}
```

Field Presence

Field presence decides whether a reader can tell a field holding its zero value, an int at `0` or a string at `""`, apart from a field nobody set. Edition 2024 gives every singular field explicit presence by default, so a value of `0` and an absent field are different states. With `IMPLICIT`, that distinction disappears. A zero stops getting written at all, and a reader has no way to tell it from a field that was never set.

The two schemas that follow differ only in their `field_presence` setting, `EXPLICIT` or `IMPLICIT`. Every field in both messages is set to its zero value, the only case where the two behave differently.

>_ EXPLICIT PRESENCE

```
edition = "2024";

package demo.v1;

option features.field_presence = EXPLICIT;

message Settings {
  string label = 1;
  int32 retries = 2;
  bool verbose = 3;
}
```

>_ JSON

```
{
  "label": "",
  "retries": 0,
  "verbose": false
}
```

With explicit presence, a field set to zero is still set, so the JSON includes it.

>_ IMPLICIT PRESENCE

```
edition = "2024";

package demo.v1;

option features.field_presence = IMPLICIT;

message Settings {
  string label = 1;
  int32 retries = 2;
  bool verbose = 3;
}
```

>_ JSON

```
{}
```

With implicit presence, zero and unset are indistinguishable, so the JSON is empty.

Required Fields

proto3 dropped the `required` keyword, and editions did not bring it back except as a legacy feature for proto2 files that already use it. Required fields get in the way of schema evolution. Every message has to include a required field. If you stop sending it, every older reader fails to decode the new message, and there is no way to retire the field gradually. Protobuf's own maintainers came to see `required` as a mistake, and the designers of Cap'n Proto reached the same conclusion.

You can still require a field. The check happens in validation now, which can change without breaking the wire format. Tightening a rule can still reject requests that existing clients send, so roll it out like any other API change. In application code, generated getters return the zero value for a missing field and your handler checks for it. In the schema, a custom option declares the field required and a validator enforces it when a message is checked rather than when it is decoded, so the field stays wire compatible. protovalidate's `required` rule is the common choice, and since the mechanism is just options, you can define and check your own the same way.

>_ METADATA VALIDATION

```
import "buf/validate/validate.proto";

message CreateUserRequest {
  // Required at the validation layer
  // but optional at the wire layer.
  string email = 1 [
    (buf.validate.field).string.email = true,
    (buf.validate.field).required = true
  ];
}
```

>_ APPLICATION VALIDATION (GO)

```
// Safe access even if req is nil
if req.GetEmail() == "" {
    return status.Error(codes.InvalidArgument, "email is required")
}
```

Realistic Limits & Size

Maximum size

A Protobuf message can't be larger than **2 GiB**. The wire format could describe bigger lengths, since length prefixes are varints, but the official implementations track sizes as 32-bit signed integers. A standard parser that hits a payload past it throws an overflow error and refuses to read it, so treat the limit as fixed rather than a setting you can raise.

Practical sizes

In practice, keep messages under **1 MB**, even though the official recommendation only asks for a few megabytes. Protobuf was designed with small messages in mind.

Past 10 MB, parsing uses a noticeable amount of CPU and memory. To move large datasets, chunk the data into a stream of smaller messages.

Whole-message parsing

The generated API parses a whole message and builds a complete object graph in memory before returning it. The parser can read from a stream, and some runtimes decode parts lazily, but you still get the message as one unit. Plan for the entire message to be in memory at once.

That in-memory copy is usually bigger than the wire bytes. Pointers, object overhead, and padding in your data structures can push memory usage to several times the size of the original payload. Most serialization formats behave the same way.

Further reading: Advanced

Imports

Tip of the week #5: Avoid import public/weak
https://buf.build/blog/totw-5-avoid-import-public-weak

Two special import modes, and why Buf lints against both by default.

Editions

[Protobuf Editions are here: don't panic](https://buf.build/blog/protobuf-editions-are-here)
https://buf.build/blog/protobuf-editions-are-here

An overview of editions for teams on proto2 or proto3, and how to migrate.

Unknown Fields

[Unknown Fields in Protobuf](https://kmcd.dev/posts/protobuf-unknown-fields/)
https://kmcd.dev/posts/protobuf-unknown-fields/

How each runtime preserves unknown fields, and the risks of forwarding them across a security boundary.

[Unknown fields in the official language guide](https://protobuf.dev/programming-guides/proto3/#unknowns)
https://protobuf.dev/programming-guides/proto3/#unknowns

The reference semantics: what parsers must retain and how each runtime exposes it.

Schema Evolution

[Tip of the week #4: Accepting mistakes we can't fix](https://buf.build/blog/totw-4-accepting-mistakes)
https://buf.build/blog/totw-4-accepting-mistakes

Schema mistakes that can't be undone once data exists, and how to work around them.

[Tip of the week #1: Field names are forever](https://buf.build/blog/totw-1-field-names)
https://buf.build/blog/totw-1-field-names

The rename that's wire-safe but still breaks generated code and JSON.

Field Presence

[Application note: Field presence](https://protobuf.dev/programming-guides/field_presence/)
https://protobuf.dev/programming-guides/field_presence/

The official deep dive on presence semantics across proto2, proto3, and editions, and how each runtime exposes them.

[Protobuf Editions are here: don't panic](https://buf.build/blog/protobuf-editions-are-here)
https://buf.build/blog/protobuf-editions-are-here

Presence became a per-field feature with editions; this covers the migration story.

Required Fields

[Protobuf Tip #8: Never use required](https://buf.build/blog/totw-8-never-use-required)
https://buf.build/blog/totw-8-never-use-required

The history of proto2's required label, and why Buf recommends never using it.

Size Limits

[Protobuf Tip #2: Compress your Protos!](https://buf.build/blog/totw-2-compress-protos)
https://buf.build/blog/totw-2-compress-protos

Wire size tradeoffs matter less than they used to; compression is everywhere.

Binary

The wire format is how a message is turned into bytes that one program can send and another can read. This chapter covers those bytes, which the rest of Protobuf is built on.

A serialized Protobuf message is a flat stream of tag/value pairs. Each tag names a field by number and says what its wire type is. The wire type tells the decoder how many bytes the value takes up. The payload has no header, no field count, and no schema. The decoder just reads fields one after another until the bytes run out.

Most of the format's compactness comes from the varint, which takes a number of bytes that depends on the number's actual size, not its declared width. Tags, lengths, and most integer values are all varints, and ZigZag extends the same encoding to negative numbers.

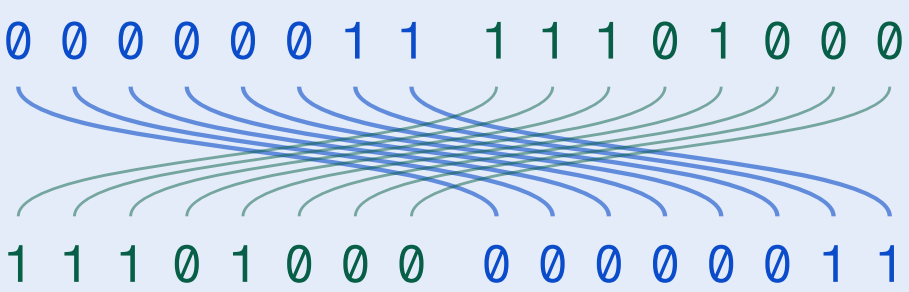
If you run into an edge case we don't cover here, the official encoding guide has the full specification.

Binary 101

Protobuf's encoding uses three bit-level operations: byte order, bit shifting, and bitwise OR. If you can already read `x << 3 | y`, skip ahead. If not, here's the short version.

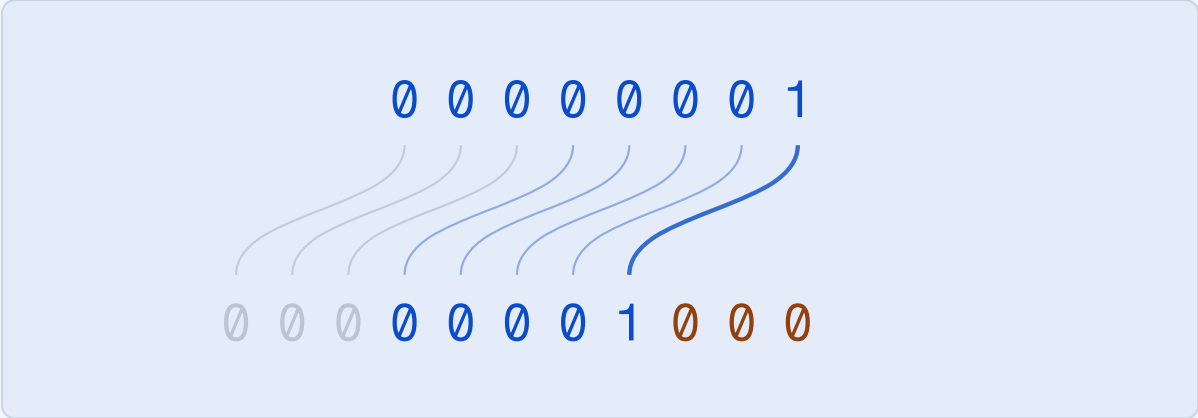
Endianness

Multi-byte numbers can be laid out least significant byte first (little-endian) or most significant byte first (big-endian). Protobuf's fixed-width types (`fixed32`, `fixed64`, `float`, `double`) are little-endian. That's why the low bytes of a number show up first in the raw stream.



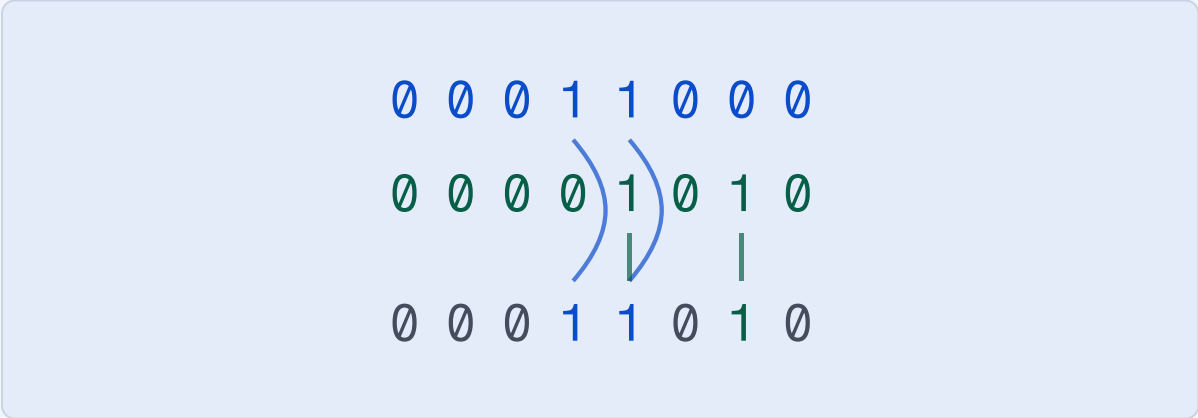
Bit Shifting

A left shift (`<<`) moves every bit left and fills the gap with zeroes. Shifting left by 3 multiplies by 8 and leaves 3 empty bits on the right. Protobuf uses this to build the field tag. It shifts the field number left by 3 to make room for the wire type.



Bitwise OR

OR (`|`) merges two numbers. An output bit is 1 if either input bit is 1. After a shift clears space, OR writes another value into it. As long as the two values occupy non-overlapping bit ranges, neither one corrupts the other.



Base-128 Varints

An `int32` in memory is four bytes whether it holds 3 or three hundred million, but a varint uses one byte for small numbers and adds a byte for every 7 more bits a larger number needs. That's why tags for field numbers 1 through 15 take a single byte.

Only 7 bits of each byte carry the number. The top bit, the most significant one, is a continuation flag. It is set to `1` on every byte except the last, so a decoder reading bytes one at a time keeps going until it reaches a byte whose top bit is `0`, which is the last byte of the varint.

The 7-bit groups come out least significant group first. So the groups are in the reverse of the order you would read the number in, and a decoder has to read the whole varint before it knows what value it has. In practice this never matters to you, but it does explain why varints look reversed in a hex dump.

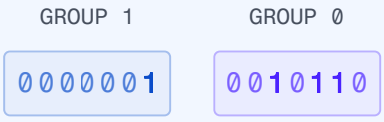
INPUT NUMBER

150

>_ VARINT ENCODING STEPS

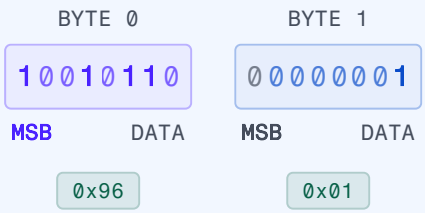
1 **Chunk Data**

The number gets split into 7-bit groups. A byte has 8 bits, so that leaves the top one free to use as a continuation bit.



2 **Reverse & Add MSB Flag**

Those groups go out least significant first, which is the little-endian order. Every byte then gets its top bit set to `1`, apart from the last one.



ZigZag

Plain varints are compact for small positive numbers, but a negative number always takes the maximum 10 bytes. ZigZag encoding maps small negative numbers to small positive ones so they stay short too.

Two's complement stores the sign in the *most significant* bit. That means every negative number has its high bits set. Varints save space by dropping leading zeros, and a negative number doesn't have any to drop. Even `-1` takes the full 10 bytes of a 64-bit varint. That holds for `int32` too: a negative `int32` is sign-extended to 64 bits before encoding, so a reader can parse the same bytes as an `int64`.

ZigZag encoding moves the sign to the *least significant* bit instead. Positive numbers map to even integers (`n << 1`) and negative numbers to odd ones.

With the sign at the bottom, small negative numbers have leading zeros again and encode as compactly as small positive ones.

The name comes from how the mapping alternates between positive and negative as the encoded value counts up. `0` maps to `0`, `-1` to `1`, `1` to `2`, `-2` to `3`, `2` to `4`, and so on.

In a schema, `sint32` and `sint64` use ZigZag. The Types reference covers when to reach for those over plain `int32` and `int64`.

SIGNED INPUT

-1

>_ ZIGZAG TRANSFORMATION

ORIGINAL SIGNED

-1

→

`(N << 1) ^ (N >> 63)`

ENCODED UNSIGNED

1

Standard Varint (Two's Complement) — 10 Byte(s)

`11111111` `11111111` `11111111` `11111111` `11111111`
`11111111` `11111111` `11111111` `11111111` `00000001`

As a plain varint, `-1` takes all 10 bytes: a negative number has no leading zeros to drop.

ZigZag Varint — 1 Byte(s)

`00000001`

After ZigZag, size depends on the number's magnitude rather than its sign: `-1` encodes in a single byte, and the byte count grows only as the value moves away from zero.

The Field Tag

Every field in a Protobuf message starts with a tag. The tag tells the decoder which field comes next and how to read the bytes that follow.

TAG COMPOSITION

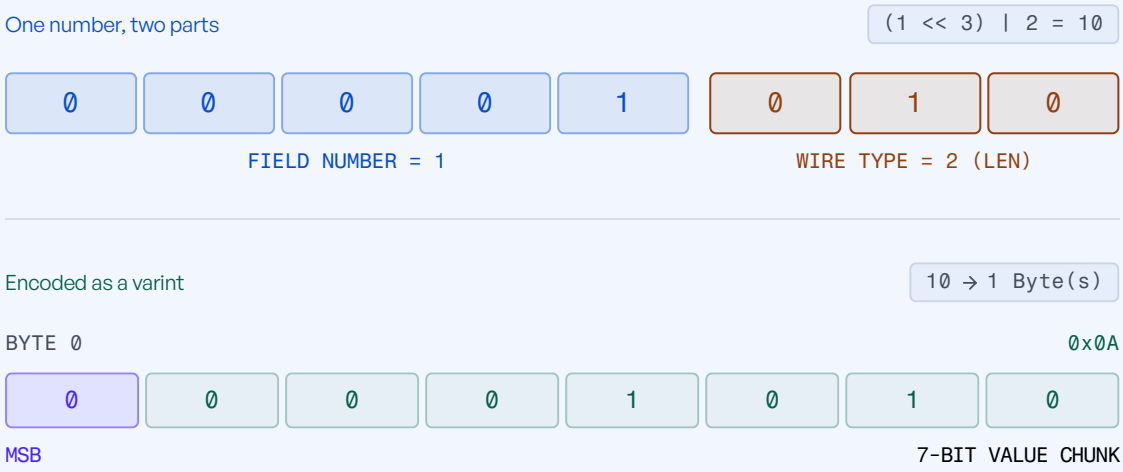
A tag is a single Varint that combines two pieces of information:

- Field Number (bits 3 through N)
- Wire Type (the bottom 3 bits)

The formula for the tag value is `(field_number << 3) | wire_type`. Small field numbers fit in a single byte. Larger ones, 16 and above, spill into additional bytes under the same continuation rule varints always follow.

>_ TAG STRUCTURE: FIELD 1

The lowest three bits of a tag hold the wire type, and everything above them holds the field number.



>_ TAG STRUCTURE: FIELD 16

The lowest three bits of a tag hold the wire type, and everything above them holds the field number.

One number, two parts

`(16 << 3) | 2 = 130`



FIELD NUMBER = 16 WIRE TYPE = 2 (LEN)

Encoded as a varint

`130 → 2 Byte(s)`



VARINT CONTINUATION: With field 16, the tag value is 130. That no longer fits in the 7 payload bits of a single varint byte, so the tag spills into a second byte. This is why the Fields page suggests saving numbers 1 through 15 for the fields you send most.

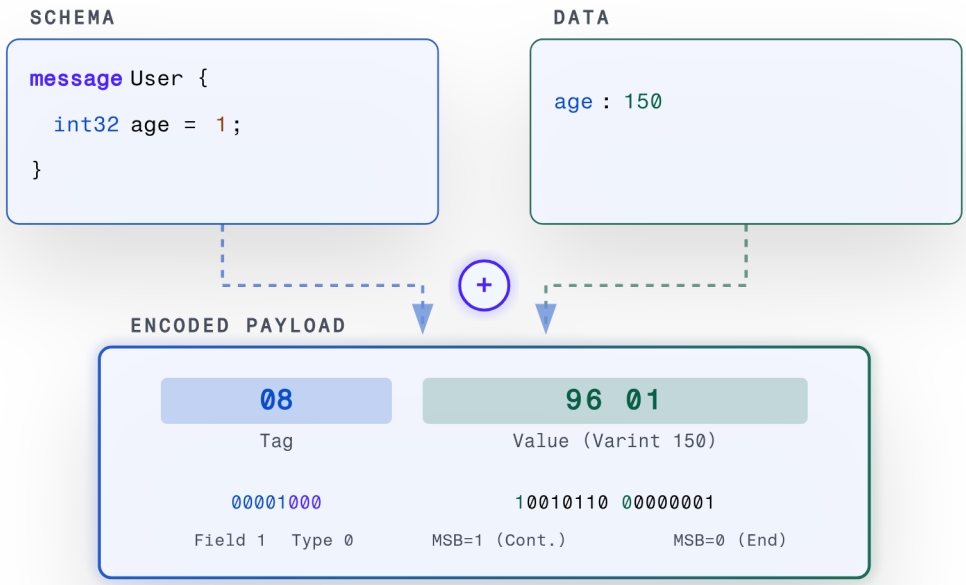
Wire Types

Every field on the wire starts with a tag: the field number and a 3-bit wire type, packed into a single varint. The wire type tells the decoder how to read the bytes that follow.

WIRE TYPE	PROTOBUF TYPES
0 (Varint)	`int32` `int64` `uint32` `uint64` `sint32` `sint64` `bool` `enum`
1 (I64)	`fixed64` `sfixed64` `double`
2 (LEN)	`string` `bytes` `message` also used for packed repeated scalars
3 (SGROUP)	group start (deprecated)
4 (EGROUP)	group end (deprecated)
5 (I32)	`fixed32` `sfixed32` `float`

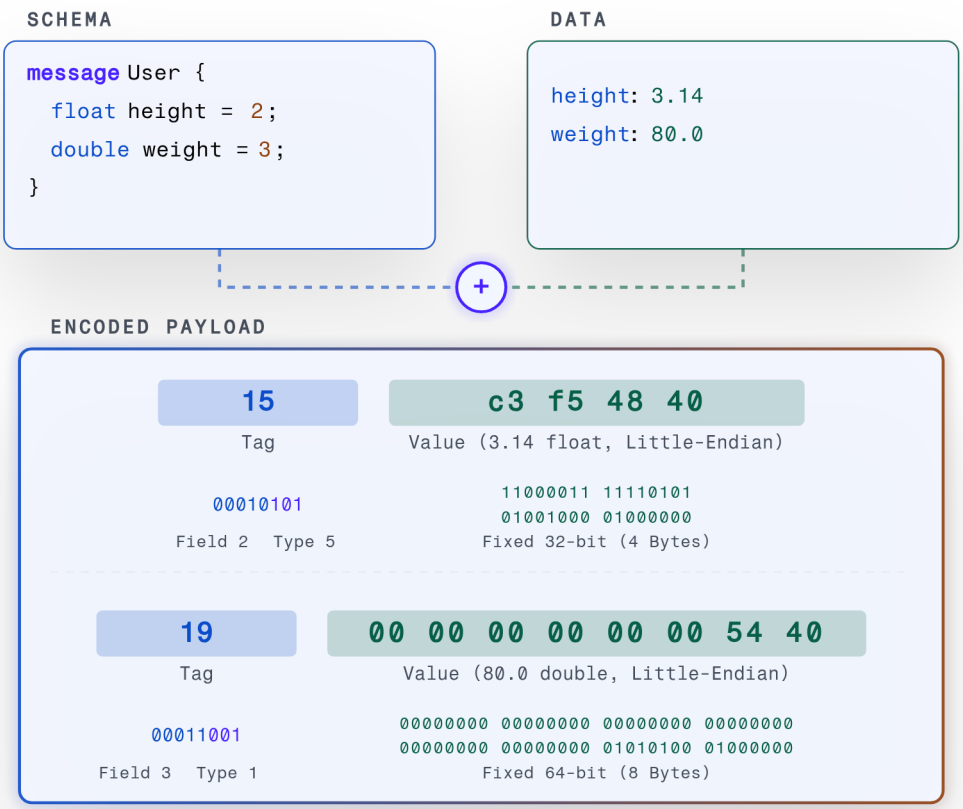
Varint Fields (Wire Type 0)

Most numeric types use Wire Type 0. The length is implicit because the decoder reads bytes one by one until it finds a byte where the MSB is 0, using the same continuation-bit rule as the varint section earlier.



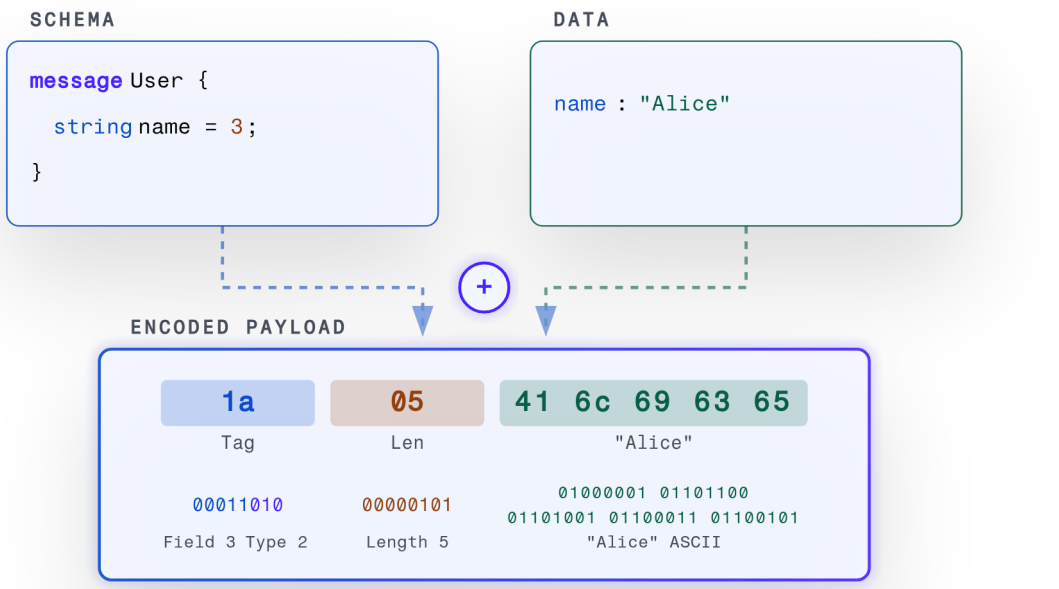
Fixed-Size Fields (Wire Type 1, 5)

Fixed-width types have a constant size, so they need no length prefix or continuation bits. Wire Type 1 is always 8 bytes, used by `double` and `fixed64`. Wire Type 5 is always 4 bytes, used by `float` and `fixed32`. The decoder reads that many bytes after the tag.



Length-Delimited (Wire Type 2)

`string`, `bytes`, and nested `message` fields use Length-Delimited encoding. These fields carry an explicit length prefix, encoded as a varint, right after the tag. That prefix tells the decoder exactly how many of the bytes that follow belong to this field.



Packed Repeated Fields

Repeated scalar fields are packed by default. On the wire that means one tag and one length prefix for the whole list instead of a tag per element. Set `features.repeated_field_encoding`

`= EXPANDED` on a file or a field to get the older one-tag-per-element layout back.

>_ EXPANDED

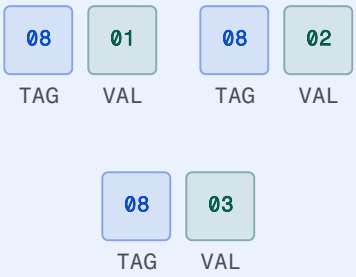
SCHEMA

```
repeated int32 ids = 1
[features.repeated_field_encoding
= EXPANDED];
```

VALUES

[1, 2, 3]

WIRE LAYOUT



Each element repeats the field tag, which adds up for many small elements.

>_ PACKED (EDITION 2024 DEFAULT)

SCHEMA

```
repeated int32 ids = 1;
```

VALUES

[1, 2, 3]

WIRE LAYOUT



Elements are concatenated into a single length-delimited record with one tag.

Map Encoding

A map field is shorthand for a repeated message of key/value pairs.

```
map<string, int32> items = 1;
```

Is equivalent to:

```
message Entry {
  string key = 1;
  int32 value = 2;
}
repeated Entry items = 1;
```

On the wire, each entry is an ordinary length-delimited sub-message, with the key in field 1 and the value in field 2. A decoder written before maps existed still parses the data correctly, as a repeated field, so maps stay compatible with older Protobuf implementations.

This representation explains why entries carry no defined order, and why the last entry wins when the same key shows up twice in a payload. A repeated field would keep both entries; a map keeps one value per key.

Presence

Whether a field holding its zero value reaches the wire at all is a per-field choice, set by its field presence mode.

On the wire, an **implicit presence** field holding its zero value writes no bytes, not even a tag. An **explicit presence** field writes its tag and the value, `08 00` for an `int32` in field 1 set to zero, as soon as your code sets it. Message fields and members of a `oneof` always have explicit presence, regardless of the file's default.

Edition 2024 defaults to `EXPLICIT`. Set `features.field_presence = IMPLICIT;` on a file or a field to leave default-valued fields off the wire instead.

>_ IMPLICIT PRESENCE

SCHEMA DEFINITION

```
int32 count = 1
[features.field_presence =
IMPLICIT];
```

ASSIGNED VALUE

0 OMITTED

WIRE REPRESENTATION

(ZERO BYTES)

Omitted from the wire whenever it holds the zero value, so a reader cannot tell this from never set.

>_ EXPLICIT PRESENCE

SCHEMA DEFINITION

```
int32 count = 1;
```

ASSIGNED VALUE

0 (set) SERIALIZED

WIRE REPRESENTATION



Edition 2024's default. Once set, the field is written even when the value is zero.

Field-by-Field Parsing

When multiple fields are sent together, Protobuf just concatenates them into one binary stream. The decoder doesn't need separators between fields. Each field tells the decoder how many bytes to consume before it moves on.

01

Read the tag

The first varint in each field is the tag. Its lower three bits identify the wire type, and the remaining bits identify the field number.

02

Read the payload

The wire type tells the decoder where the payload ends. A varint ends at the first byte with its top bit clear, so the end is found while reading it. A fixed field is 4 or 8 bytes. A length-delimited field carries its size in a prefix.

03

Consume that field

The decoder takes exactly those bytes, maps them to the schema field when it recognizes the number, and moves to the next field.

04

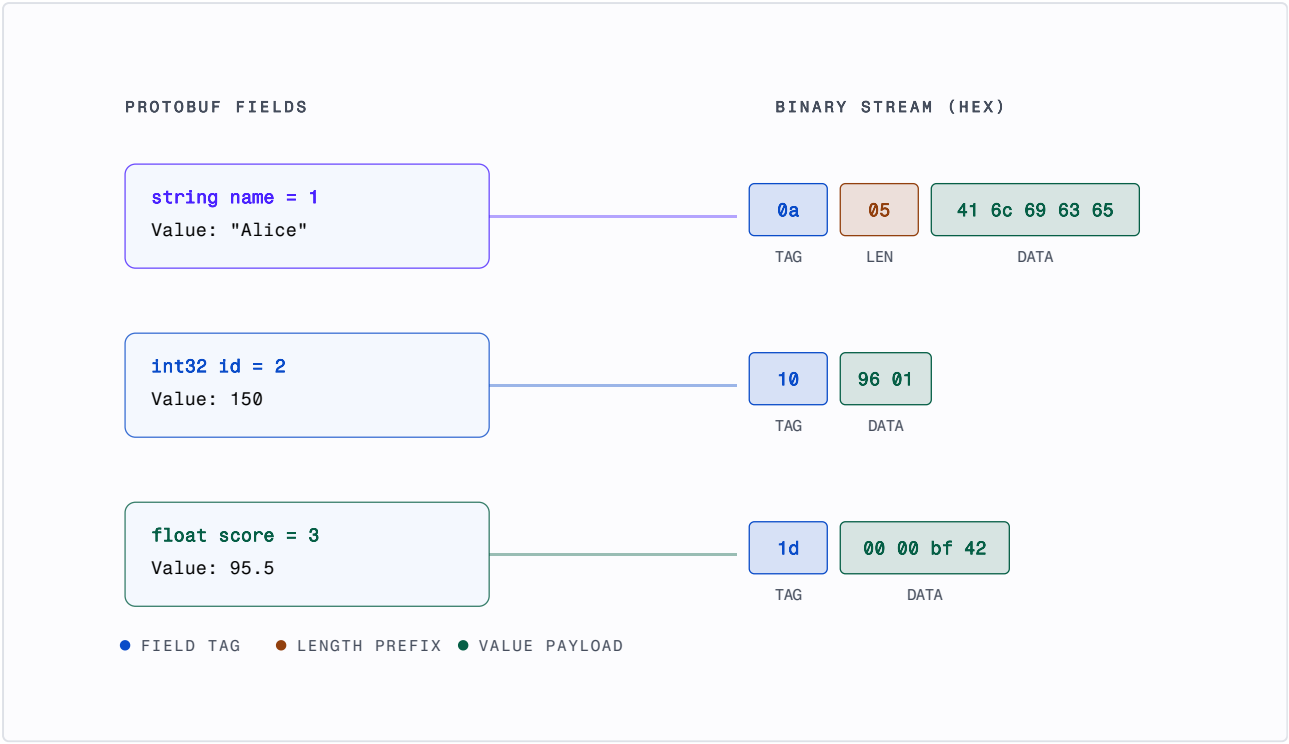
Repeat to the end

The stream carries no outer field count. Parsing picks up at the next byte and continues until the end of the message.

When a field number appears twice

A payload can contain the same field number more than once. For a singular scalar or string field, the last value wins. For a singular message field, the decoder merges each occurrence into the same message. Repeated fields append. Because of this, two serialized messages of the same type concatenated together parse as one message with both merged.

It also means serialization isn't canonical. The same message can be encoded as different bytes depending on field order, map order, unknown fields, and the library that wrote it, and deterministic serialization only promises the same output from the same binary. Don't compare or hash encoded messages to check whether they are equal. The Protobuf documentation explains the details.



Further reading: Binary

Binary 101

[Endianness](https://en.wikipedia.org/wiki/Endianness)
https://en.wikipedia.org/wiki/Endianness

Little- and big-endian byte order, and why it matters when bytes move between machines.

[Bit shifts](https://en.wikipedia.org/wiki/Bitwise_operation#Bit_shifts)
https://en.wikipedia.org/wiki/Bitwise_operation#Bit_shifts

The shift operations behind the tag formula, including the arithmetic/logical distinction this page glosses over.

[Bitwise OR](https://en.wikipedia.org/wiki/Bitwise_operation#OR)
https://en.wikipedia.org/wiki/Bitwise_operation#OR

How OR merges bit patterns, along with the AND and XOR operations decoders also use.

Base-128 Varints

[Delta Encoding in Protobuf](https://buf.build/blog/delta-encoding-in-protobuf)
https://buf.build/blog/delta-encoding-in-protobuf

Storing differences between consecutive values so varints stay small, which shrinks repeated numeric data further.

ZigZag

[Protobuf Tip #10: Choosing the right integer type](https://buf.build/blog/choosing-the-right-protobuf-integer-type)
https://buf.build/blog/choosing-the-right-protobuf-integer-type

When the ZigZag-encoded sint32 and sint64 types actually pay off.

The Field Tag

[Tip of the week #9: Some numbers are more equal than others](https://buf.build/blog/totw-9-some-numbers-are-more-equal-than-others)
`https://buf.build/blog/totw-9-some-numbers-are-more-equal-than-others`

The tag math on this page, taken further: number ranges, reserved blocks, and rationing single-byte tags.

Wire Types

[Message structure, in the encoding reference](https://protobuf.dev/programming-guides/encoding/#structure)
`https://protobuf.dev/programming-guides/encoding/#structure`

The official wire-type table and the rules for how tags frame each payload.

[Repeated elements and packed encoding](https://protobuf.dev/programming-guides/encoding/#repeated)
`https://protobuf.dev/programming-guides/encoding/#repeated`

The reference treatment of repeated fields on the wire, including when they pack and when they don't.

Map Encoding

[Maps in the language guide](https://protobuf.dev/programming-guides/editions/#maps)
`https://protobuf.dev/programming-guides/editions/#maps`

The official maps documentation, including the backwards-compatibility note behind the repeated-entry representation on this page.

Presence

[Application note: Field presence](https://protobuf.dev/programming-guides/field_presence/)
`https://protobuf.dev/programming-guides/field_presence/`

The official deep dive on presence semantics across proto2, proto3, and editions, and how each runtime exposes them.

[Tip of the week #7: Scoping it out](https://buf.build/blog/totw-7-scoping-it-out)
`https://buf.build/blog/totw-7-scoping-it-out`

The command-line version of the site's Binary Explorer: disassembling real payloads with buf convert and protoscope.

Efficiency

A Protobuf schema already defines the field names and the structure, so the binary encoding doesn't repeat any of that on the wire. It just carries the data. Most teams pick Protobuf for the schema, and the smaller payload comes along with it. This page measures how much space the binary encoding saves, and the cases where it saves little.

Each row is a `demo.v1.User` message. Basic is one user with every field set except `manager`. Large is a user with four levels of managers nested inside it. Minimal sets five short fields. JSON sizes are compact ProtoJSON in UTF-8 bytes, binary sizes are the serialized message, and the gzip columns compress each one at default settings.

EXAMPLE	JSON	BINARY	JSON, GZIP	BINARY, GZIP
Basic	207 B	103 B (-50.2%)	191 B	123 B (-35.6%)
Large	1045 B	518 B (-50.4%)	338 B	283 B (-16.3%)
Minimal	133 B	78 B (-41.4%)	141 B	98 B (-30.5%)

ON THE WEB

Size comparison: weigh any message's JSON against its binary encoding, raw or compressed
protobuf.com/efficiency/

Where the binary encoding saves its bytes

Numeric Tags

Field names aren't sent, only the small numeric tags from the schema.

No Syntax Overhead

Structure comes from the schema, so no braces, quotes, or commas are sent.

Varints

Small integers take 1-2 bytes, not their full width.

Unset Fields

A field that isn't set takes exactly zero space in the payload.

Size vs. Compression

JSON's repeated keys compress well, so gzip recovers much of what the binary encoding saves by dropping field names. Compression also costs CPU on both ends, and on small payloads the framing it adds can exceed what it saves. Measure both with your own data.

The binary encoding does best on numbers, enums, and messages that leave most fields unset. It gains the least on long strings. When you need the payload to be human-readable, Protobuf messages serialize to JSON too. ProtoJSON is part of the specification, so the same schema and the same generated types produce either encoding.

Performance in Practice

Parse cost depends on the language and the library. In C++, Go, and Java, the binary encoding can parse much faster than JSON. In JavaScript and Python the gap narrows, since the data still has to cross into the runtime either way. Published benchmarks are a starting point; measure your own services before making design decisions on them.

A few write-ups measure that difference in practice:

Auth0 Engineering

Classic comparison of binary vs text overhead in real-world API requests.

Official gRPC Benchmarks

Throughput and latency metrics for Protobuf-over-HTTP/2 across various languages.

Atlassian Engineering

A case study on cutting p99 latency by 20% and CPU usage by 75% with Protobuf.

hyperpb

A dynamic parser for Go that works from descriptors instead of generated code, benchmarked against the standard runtime.

Descriptors & Reflection

The structure of a `.proto` file is itself defined in Protobuf, in a file called `descriptor.proto` that ships with the compiler. When the compiler reads your schema, it produces descriptors, which are messages of those types that describe every file, message, and field. Reflection, dynamic decoding, validation, and compiler plugins all work from descriptors.

Schemas Describing Schemas

The compiler can write your schema out as a `FileDescriptorSet` with `buf build -o descriptor.binpb` or `protoc --descriptor_set_out`. That output is an ordinary Protobuf message, defined by `descriptor.proto`, so the same tools that work with any other message can read and analyze a schema at runtime. Buf's descriptors reference goes through the details.

Why is this useful?

Dynamic Decoding

Tools like the site's Binary Explorer use descriptors to decode arbitrary binary data without generating static code.

Validation

Complex rule engines (like protovalidate) use descriptors to apply constraints dynamically.

Code Generation

Code generator plugins receive these descriptors as input. Every custom generator is built on them, and the same plugin binary works under both `buf generate` and `protoc`.

Server Reflection

A server can send its own descriptors to clients over server reflection, so a client can call it without having the schema ahead of time. `buf curl` uses this to invoke any reflection-enabled gRPC or Connect endpoint with no `.proto` files on your disk.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

import "buf/validate/validate.proto";

message User {
  string id = 1 [json_name = "uid"];
  string name = 2;
  uint32 age = 3 [(buf.validate.field).uint32.lt = 150];
  Role role = 4;

  enum Role {
    ROLE_UNSPECIFIED = 0;
    ROLE_USER = 1;
    ROLE_ADMIN = 2;
  }
}
```

>_ DESCRIPTOR OUTPUT (JSON)

```
{
  "file": [
    {
      "name": "input.proto",
      "package": "demo.v1",
      "dependency": ["buf/validate/validate.proto"],
      "messageType": [
        {
          "name": "User",
          "field": [
            {
              "name": "id",
              "number": 1,
              "label": "LABEL_OPTIONAL",
              "type": "TYPE_STRING",
              "jsonName": "uid"
            },
            {
              "name": "name",
```

```
                "number": 2,
                "label": "LABEL_OPTIONAL",
                "type": "TYPE_STRING",
                "jsonName": "name"
              },
              {
                "name": "age",
                "number": 3,
                "label": "LABEL_OPTIONAL",
                "type": "TYPE_UINT32",
                "options": { "[buf.validate.field]": { "uint32": { "lt": 150 }
} },
                "jsonName": "age"
              },
              {
                "name": "role",
                "number": 4,
                "label": "LABEL_OPTIONAL",
                "type": "TYPE_ENUM",
                "typeName": ".demo.v1.User.Role",
                "jsonName": "role"
              }
            ],
            "enumType": [
              {
                "name": "Role",
                "value": [
                  { "name": "ROLE_UNSPECIFIED", "number": 0 },
                  { "name": "ROLE_USER", "number": 1 },
                  { "name": "ROLE_ADMIN", "number": 2 }
                ]
              }
            ]
          }
        ],
        "syntax": "editions",
        "edition": "EDITION_2024"
      }
    ]
  }
```


ON THE WEB

Descriptor explorer: edit a schema and watch its FileDescriptorSet change as you type

protobuf.com/descriptors/

Extending

Protobuf can be extended at compile time and at schema-definition time. This page covers plugins, extensions, and custom options.

Everything on this page works because the schema itself is data. When the compiler parses a `.proto` file, it produces descriptors, Protobuf messages that describe every file, message, field, and service in the schema. Each mechanism on this page reads or extends those descriptors.

Plugins consume descriptors at build time. The compiler parses your schema, serializes the descriptors into a `CodeGeneratorRequest`, and pipes it to a plugin: an ordinary program, in any language, that returns the files it wants to generate. The official code generators work this way, and you can write your own that emits documentation, SQL schemas, or API clients instead.

Extensions widen messages you don't own. A message can reserve a range of field numbers for external use, and other files then declare fields in that range without touching the original definition. The base message's schema doesn't change, but the messages it describes can carry the new fields.

Custom options are built on extensions and descriptors. The descriptor types that hold options, like `FieldOptions` and `MethodOptions`, declare extension ranges of their own, so extending them attaches your metadata to any schema element. That metadata travels inside the descriptors, where plugins and runtime reflection can read it back. protovalidate uses custom options to attach validation rules to fields.

Custom Plugins

`protoc` has generators for a few languages built in, such as C++, Java, and Python. Everything else, Go and TypeScript included, comes from a plugin: the compiler (`protoc` or `buf generate`) parses the `.proto` files and hands the resulting descriptors to a separate program.

A plugin is just an executable that reads a `CodeGeneratorRequest` and writes a `CodeGeneratorResponse`, both defined in `google/protobuf/compiler/plugin.proto`.

```yaml
# buf.gen.yaml
version: v2
plugins:
  - local: ./my-plugin
    out: ./generated
    opt: log_level=debug,other_flag=true
```

stdin, stdout, stderr

The compiler starts the plugin program as a subprocess.

- **stdin:** The compiler passes a binary serialized `CodeGeneratorRequest` message.
- **stdout:** The plugin must return a binary serialized `CodeGeneratorResponse` message. The plugin must *not* modify the filesystem directly; it returns the files to be written in this response.
- **stderr:** For logs and error messages only. Keep logging off by default and turn it on with a flag, so normal runs stay quiet.

Request and response

Options passed via `--<plugin>_opt` arrive in the `parameter` field of the `CodeGeneratorRequest` as a single comma-separated string. The plugin must parse and split it.

The compiler passes many files (including dependencies), but the plugin must only generate code for the files listed in the `file_to_generate` field of the request.

In the `CodeGeneratorResponse`, declare the features your plugin supports. Setting `supported_features` along with `minimum_edition` and `maximum_edition` is required to compile files that use Protobuf Editions. Without them, the compiler refuses to run the plugin on those files, since it can't tell whether the plugin understands their features.



Proto Extensions

Where you can use extensions depends on the syntax:

- `proto2` : Allows extensions on any message (both user-defined messages and standard options).
- `proto3` : Restricts extensions exclusively to option messages (specifically to define custom options; more on that later).
- Editions: Restores the ability to extend any message (bringing back general-purpose extensions) while keeping option definitions standard and native.

To use extensions in `proto2` or `Editions`, you must define an extension range in the base message using the `extensions` keyword. External files can then declare fields targeting that range.

EXTENSION NUMBERS ARE FIELD NUMBERS

Extension numbers are ordinary field numbers, and they share the base message's tag space on the wire. No two extensions of the same message can use the same number, or their data collides and gets misread.

>_ BASE.PROTO (DEFINITION)

```
edition = "2024";

message UserProfile {
  string username = 1;

  // Declare range of tags reserved for third-party extensions
  extensions 100 to 199;
}
```

>_ BILLING.PROTO (EXTENSION)

```
edition = "2024";
import "base.proto";

// Extend the custom UserProfile message directly
extend UserProfile {
  string stripe_customer_id = 100;
}
```

Custom Options

Teams use them for validation rules (protovalidate), for tagging fields that hold PII, and for access control, like the roles an RPC requires.

The compiler records these annotations in the descriptors, so any program that reads your schema can see them: a protoc plugin generating custom code, a service that configures itself at startup, or a tool that loads schemas at runtime through reflection.

A custom option is an extension of one of the built-in option messages, such as `FieldOptions` or `MethodOptions`, declared with the `extend` keyword.

AVAILABLE SCOPES

`File` `Message` `Field` `Oneof` `Enum` `EnumValue` `Service` `Method`

Metadata can be attached to any of these points by extending the respective standard descriptor messages.

For more information, see the Editions Custom Options Guide.

Extension Registries

Every extension of a descriptor message such as `FieldOptions` shares one set of field numbers, so your extension numbers can't overlap with anyone else's.

Google maintains a Global Extension Registry for public projects.

>_ OPTIONS.PROTO (DEFINITION)

```
edition = "2024";
import "google/protobuf/descriptor.proto";

extend google.protobuf.FieldOptions {
  bool is_pii = 50001;
}

extend google.protobuf.MethodOptions {
  string required_role = 50002;
}
```

>_ SERVICE.PROTO (USAGE)

```
edition = "2024";
import "options.proto";

service UserService {
  rpc GetSensitiveData(GetRequest) returns (GetResponse) {
    option (required_role) = "ADMIN";
  }
}

message Profile {
  string ssn = 1 [(is_pii) = true];
}
```

Validation

Generated code already checks your types. A string field always holds a string, and an int32 always holds a number. With protovalidate you can also write range and format rules straight into the schema, next to the fields they constrain.

Type Validation

A schema type can tell you a field is a string, but it cannot tell you the string is a valid email address. It cannot tell you an age is in range, or that two fields satisfy some business rule between them.

Validation rules describe those expectations in the schema itself. You add annotations to the fields, and your services run a validator that checks each message against them. Services in different languages read the same rules from the same schema.

```
message SignupRequest {
  string email = 1;
  uint32 age = 2;
  repeated string roles = 3;
}
```

What a rule looks like

A protovalidate rule is a field option. You write it in the brackets after the field, right where any other option goes. That means the rule lives in the same file as the field it constrains, so it goes through review with the field and ends up in the same generated descriptors. The protovalidate library for each language reads the rules from those descriptors and enforces them at runtime.

A file that uses rules imports them like any other schema, with `import "buf/validate/validate.proto";`. Nothing else about the field changes, and removing the options leaves a plain Protobuf schema.

```
string email = 3 [(buf.validate.field).string.email = true];
```

`(buf.validate.field)`
The option protovalidate defines. The parentheses mark it as an extension, so this is a custom option rather than one built into Protobuf.

`.string.email`
The rule: the field's type, then a rule for that type. A uint32 field takes numeric rules, a repeated field takes list rules.

`= true`
The value. Some rules are toggles; others take a number, a string, or a list of allowed values.

Length and format

>_ ACCOUNT.PROTO

```
message Account {
  string id = 1 [
    (buf.validate.field).string.uuid = true
  ];
  string name = 2 [
    (buf.validate.field).string.min_len = 2,
    (buf.validate.field).string.max_len = 50
  ];
  string email = 3 [
    (buf.validate.field).string.email = true
  ];
}
```

- **`string.uuid`**
  Requires the string to be a well-formed UUID.
- **`string.min_len`**
  Sets a minimum length, in characters.
- **`string.max_len`**
  Sets a maximum length. A field can have several rules, and all apply.
- **`string.email`**
  Common formats are toggles: email, uuid, hostname, ip, uri.

Ranges

>_ MEMBERSHIP.PROTO

```
message Membership {
  uint32 age = 1 [
    (buf.validate.field).uint32 = {
      gte: 18
      lt: 120
    }
  ];
}
```

- **uint32.gte**
  Sets an inclusive lower bound. The braces set both rules at once.
- **uint32.lt**
  Sets an exclusive upper bound, where lte would include it.

Enum values

>_ ASSIGNMENT.PROTO

```
message Assignment {
  Role role = 1 [
    (buf.validate.field).enum = {
      defined_only: true
    }
  ];
  enum Role {
    ROLE_UNSPECIFIED = 0;
    ROLE_USER = 1;
    ROLE_ADMIN = 2;
  }
}
```

- **enum.defined_only**
  Rejects any number the enum does not declare. An open enum would otherwise accept it.

Must be set

>_ LOGIN_EVENT.PROTO

```
message LoginEvent {
  string user_id = 1 [
    (buf.validate.field).string.uuid = true
  ];
  google.protobuf.Timestamp event_time = 2 [
    (buf.validate.field).required = true
  ];
}
```

- **required**
  Makes the field mandatory. New schemas use this in place of legacy required. The message's encoding is unchanged, so old messages still parse.

Lists and maps

>_ TEAM.PROTO

```
message Team {
  repeated string roles = 1 [
    (buf.validate.field).repeated = {
      min_items: 1
      unique: true
      items: {
        string: {min_len: 1}
      }
    }
  ];
  map<string, int32> quotas = 2 [
    (buf.validate.field).map = {
      max_pairs: 10
      values: {
        int32: {gte: 0}
      }
    }
  ];
}
```

- `repeated.min_items`
  Sets the fewest elements the list may hold.
- `repeated.unique`
  Requires every element to be distinct.
- `repeated.items`
  Applies the rules of a plain string field to each element.
- `map.max_pairs`
  Sets the most entries the map may hold.
- `map.values`
  Applies rules to each value. A keys block constrains the map's keys.

Rules that span fields

>_ DATE_RANGE.PROTO

```
message DateRange {
  google.protobuf.Timestamp start = 1 [
    (buf.validate.field).required = true
  ];
  google.protobuf.Timestamp end = 2 [
    (buf.validate.field).required = true
  ];
  option (buf.validate.message).cel = {
    id: "date_range.start_before_end"
    message: "start must be before end"
    expression: "this.start < this.end"
  };
}
```

- `message.cel`
  Goes on the message because the check compares two of its fields.
- `cel.message`
  The error reported when the expression is false.
- `cel.expression`
  A CEL expression over this message. Field rules compile to CEL too.

Running the rules

By default, validation skips unset fields that track presence. In edition 2024 every singular field does, so a message that leaves `email` out passes `string.email` without complaint. Fields without presence, including repeated fields and maps, are still checked: an empty list fails `repeated.min_items`.

`{}`	Passes. The field is unset, so its rules are skipped.
`{ "email": "not-an-email" }`	Fails. The field is set, so the email rule runs.
`{ "email": "hiro@metaverse.com" }`	Passes. The field is set and the value is valid.

When the field has to be there, add `(buf.validate.field).required = true` next to the other rules. Then the empty message fails too.

In TypeScript, running the rules takes a `createValidator()` call and one `validate()` per message. Every protovalidate library has the same shape, and Go servers built on ConnectRPC can skip the manual call entirely: the validate interceptor checks every request and rejects invalid ones before your handler runs.

>_ VALIDATE.TS

```
import { create } from "@bufbuild/protobuf";
import { createValidator } from "@bufbuild/protovalidate";
import { UserSchema } from "./gen/demo/v1/user_pb";

const validator = createValidator();
const user = create(UserSchema, { email: "not-an-email" });

const result = validator.validate(UserSchema, user);
if (result.kind === "invalid") {
  for (const violation of result.violations) {
    console.log(violation.toString());
  }
}
```

Data Validation

Rules like the ones above are written in CEL and read straight out of the schema by protovalidate. There is no generated validation code, and the rules aren't kept separately in each service.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

import "buf/validate/validate.proto";
import "google/protobuf/timestamp.proto";

message User {
  string id = 1 [(buf.validate.field).string.uuid = true];
  string name = 2 [
    (buf.validate.field).string.min_len = 2,
    (buf.validate.field).string.max_len = 50
  ];
  string email = 3 [(buf.validate.field).string.email = true];

  uint32 age = 4 [(buf.validate.field).uint32 = {
    gte: 18
    lt: 120
  }];

  Role role = 7 [(buf.validate.field).enum.defined_only = true];
  Date birth_date = 8;

  enum Role {
    ROLE_UNSPECIFIED = 0;
    ROLE_USER = 1;
    ROLE_ADMIN = 2;
  }
}

message Date {
  int32 year = 1 [(buf.validate.field).int32 = {
    gte: 1900
    lte: 2100
  }];
  int32 month = 2 [(buf.validate.field).int32 = {
```

```
    gte: 1
    lte: 12
  }];
  int32 day = 3 [(buf.validate.field).int32 = {
    gte: 1
    lte: 31
  }];
}

message LoginEvent {
  string user_id = 1 [(buf.validate.field).string.uuid = true];
  google.protobuf.Timestamp event_time = 2 [(buf.validate.field).required =
true];
}
```

>_ VALID

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Hiro Protagonist",
  "email": "hiro@metaverse.com",
  "age": 30,
  "role": 2,
  "birthDate": {
    "year": 1996,
    "month": 1,
    "day": 1
  }
}
```

Validation passed

>_ INVALID_AGE


```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Hiro",
  "email": "hiro@metaverse.com",
  "age": 200,
  "role": 2,
  "birthDate": {
    "year": 1996,
    "month": 1,
    "day": 1
  }
}
```


⚠ AGE
must be greater than or equal to 18 and less than 120

>_ INVALID_DATE

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Hiro",
  "email": "hiro@metaverse.com",
  "age": 30,
  "role": 2,
  "birthDate": {
    "year": 1850,
    "month": 13,
    "day": 32
  }
}
```

⚠ BIRTH_DATE.YEAR
must be greater than or equal to 1900 and less than or equal to 2100

⚠ BIRTH_DATE.MONTH
must be greater than or equal to 1 and less than or equal to 12

⚠ BIRTH_DATE.DAY
must be greater than or equal to 1 and less than or equal to 31

>_ INVALID_EMAIL

```
{
  "id": "550e8400-e29b-41d4-a716-446655440000",
  "name": "Hiro",
  "email": "not-an-email",
  "age": 30,
  "role": 2
}
```

⚠ EMAIL
must be a valid email address

ON THE WEB

Validation lab: edit a message or the rules themselves and watch the violations update

protobuf.com/validation/

Validation Strategy

Putting validation in the schema means every service enforces the exact same rules, in every language. Your TypeScript frontend can check a form against the same rules your Go or Java backend runs when the request arrives.

Learn More

Protovalidate
Protovalidate Playground
Protovalidate in the Buf docs
protovalidate on the BSR

Community

Protocol Buffers started at Google in 2001 and was open-sourced in 2008. Today it is widely used for APIs between services. The Editions page traces that history release by release.

Protobuf in Production

Each of these companies and projects has either published its Protobuf definitions or written about how it uses them. Many more companies use Protobuf for internal APIs, and most of those schemas stay in private repositories.

Anthropic

Anthropic built and open-sourced a zero-copy Protobuf runtime and a ConnectRPC implementation for Rust, both already running in production there.

buf.build/blog/connect-rust-joins-the-connect-project

NVIDIA

NVIDIA's Triton Inference Server defines its inference, model metadata, and repository APIs as a Protobuf gRPC service.

github.com/triton-inference-server/common/blob/main/protobuf/grpc_service.proto

Netflix

Netflix uses gRPC between its backend services, and its engineers have written about using Protobuf's FieldMask to shape requests and responses.

netflixtechblog.com/practical-api-design-at-netflix-part-1-using-protobuf-fieldmask-35cfdc606518

Spotify

Spotify moved thousands of backend services off its in-house messaging protocol and onto gRPC with Protobuf service definitions.

leaddev.com/technical-direction/updated-story-why-we-migrate-grpc-and-how-we-go-about-it

DoorDash

DoorDash uses gRPC as the standard protocol between its microservices, with reliability settings declared as options in the service definitions.

careersatdoordash.com/blog/building-a-grpc-client-standard-with-open-source-to-boost-reliability-and-velocity/

Square

Square built and maintains Wire, its own Protobuf implementation for Android, Kotlin, Java, and Swift.

github.com/square/wire

Bluesky

Bluesky's app server reads from its data plane over ConnectRPC, with the internal API defined in Protobuf.

github.com/bluesky-social/atproto/blob/main/packages/bsky/proto/bsky.proto

Grafana Labs

Grafana Pyroscope, the continuous profiling database, defines its APIs in Protobuf and serves them with ConnectRPC.

github.com/grafana/pyroscope/tree/main/api

CockroachDB

CockroachDB's internal key-value API is a Protobuf gRPC service. SQL queries are translated into calls to it.

github.com/cockroachdb/cockroach/blob/master/pkg/kv/kvpb/api.proto

Google

Google's API design guidance is built around resource-oriented APIs with Protobuf service definitions and HTTP/JSON transcoding.

cloud.google.com/apis/docs/http

Kubernetes

Kubernetes documents Protobuf as an alternate API encoding for built-in resources, alongside JSON.

kubernetes.io/docs/reference/using-api/api-concepts/#protobuf-encoding

Envoy Proxy

Envoy's xDS APIs and configuration resources are published as versioned Protobuf API references.

www.envoyproxy.io/docs/envoy/latest/api/api

Docker / Swarm

SwarmKit keeps its control-plane API definitions in Protobuf schemas in the project source.

`github.com/moby/swarmkit/tree/master/api`

Vitess

Vitess publishes Protobuf definitions for its query, tablet, topology, and automation APIs.

`github.com/vitessio/vitess/tree/main/proto`

Etcd

Etcd exposes a gRPC API with a gRPC-gateway that maps HTTP/JSON onto the same API surface.

`etcd.io/docs/v3.5/dev-guide/api_grpc_gateway/`

TiDB

The TiDB/TiKV ecosystem maintains shared Protobuf definitions for distributed storage APIs in kvproto.

`github.com/pingcap/kvproto`

OpenTelemetry

OpenTelemetry defines its cross-language telemetry data protocol in shared Protobuf schemas.

`github.com/open-telemetry/opentelemetry-proto`

Temporal

Temporal publishes the API definitions for its workflow platform as Protobuf schemas.

`github.com/temporalio/api`

Community

Official Resources

Official Blog
https://protobuf.dev/news/

Third-Party Implementations
https://github.com/protocolbuffers/protobuf/blob/main/docs/third_party.md

GitHub Releases
https://github.com/protocolbuffers/protobuf/releases

Specifications

Encoding Specification
https://protobuf.dev/programming-guides/encoding/

Proto3 Language Guide
https://protobuf.dev/programming-guides/proto3/

JSON Mapping
https://protobuf.dev/programming-guides/proto3/#json

Support & Forums

Google Group (Official)
https://groups.google.com/g/protobuf

Buf Slack
https://buf.build/links/slack

Awesome gRPC
https://github.com/grpc-ecosystem/awesome-grpc

Stack Overflow
https://stackoverflow.com/questions/tagged/protobuf

Conclusion

This book covered how a `.proto` file defines a contract, how it becomes generated code and wire bytes, and how descriptors let you build tooling on top.

To try it yourself, take a schema of your own and add a field. Regenerate the code, encode a message, and look at the bytes in the Binary Explorer at protobuf.com/binary/explorer/. Then change the number of an existing field and run `buf breaking` against the previous version to see the change reported before it ships.

The appendices that follow collect the reference material from the site: the type tables and the full quiz bank with its answers.

Where to go next

There is a lot more to Protobuf than this guide covers. These are good places to start.

Language Spec

The full grammar for .proto files, from declarations and field types to options and reserved names.
`https://buf.build/docs/reference/protobuf-language-spec/`

Buf CLI

One binary that compiles, lints, formats, generates code, and catches breaking changes before they ship.
`https://buf.build/docs/introduction`

Buf Schema Registry

Publish schemas as versioned modules instead of vendoring .proto files, and depend on them from buf.yaml, with hosted docs and generated SDKs.
`https://buf.build/docs/bsr/`

ConnectRPC

An RPC framework for gRPC-compatible APIs that also work natively in the browser and on mobile.
`https://connectrpc.com/`

Ecosystem

Explore Protobuf libraries, runtimes, plugins, and tools from across the community.
`https://protobuf.com/ecosystem/`

Practice

Apply the concepts by fixing broken schemas in the interactive exercises.
`https://protobuf.com/practice/`

Types

Every field in a schema needs a type. This reference lists the types available. Scalar types map directly to primitives your language already has. Well-known types cover common structures like timestamps. Every type, scalar or well-known, gets a defined JSON representation too, covered in the ProtoJSON mapping below.

Language mappings follow the generated code of each official runtime, with TypeScript from Protobuf-ES and Python from protobuf-py.

Non-Numeric Types

`string`

Always UTF-8 encoded text. Limited to 2GB.

	TYPE	LITERAL
Go	`string`	`"hi"`
TypeScript	`string`	`"hi"`
Java	`String`	`"hi"`
Python	`str`	`"hi"`
C++	`std::string`	`"hi"`
ProtoJSON	`string`	`"hi"`

`bytes`

Raw bytes, no encoding assumed.

	TYPE	LITERAL
Go	`[]byte`	`[]byte("hi")`
TypeScript	`Uint8Array`	`new Uint8Array([104, 105])`
Java	`ByteString`	`ByteString.copyFromUtf8("hi")`
Python	`bytes`	`b"hi"`
C++	`std::string`	`std::string("hi")`
ProtoJSON	`base64 string`	`"aGk="`

`bool`

Encoded as a varint 0 or 1.

	TYPE	LITERAL
Go	`bool`	`true`
TypeScript	`boolean`	`true`
Java	`boolean`	`true`
Python	`bool`	`True`
C++	`bool`	`true`
ProtoJSON	`true / false`	`true`

`enum`

Predefined set of named integers. Defaults to 0.

	TYPE	LITERAL
Go	`named int32`	`pb.Status_ACTIVE`
TypeScript	`enum`	`Status.ACTIVE`
Java	`enum`	`Status.ACTIVE`
Python	`IntEnum`	`Status.ACTIVE`
C++	`enum`	`ACTIVE`
ProtoJSON	`string (the value name)`	`"ACTIVE"`

Numeric Types

int32

Signed 32-bit integer with variable-length (varint) encoding. Negative values always take 10 bytes, so prefer sint32 when they are common.

	TYPE	LITERAL
Go	int32	42
TypeScript	number	42
Java	int	42
Python	int	42
C++	int32_t	42
ProtoJSON	number	42

int64

Signed 64-bit integer with varint encoding. Use it for whole numbers unless you have a reason to pick another type.

	TYPE	LITERAL
Go	int64	42
TypeScript	bigint	42n
Java	long	42L
Python	int	42
C++	int64_t	42
ProtoJSON	string	"42"

uint32

Unsigned 32-bit integer with varint encoding. Ideal for counts and sizes that can never be negative.

	TYPE	LITERAL
Go	uint32	42
TypeScript	number	42
Java	int	42
Python	int	42
C++	uint32_t	42
ProtoJSON	number	42

uint64

Unsigned 64-bit integer with varint encoding. Ideal for large counts, sizes, and identifiers.

	TYPE	LITERAL
Go	uint64	42
TypeScript	bigint	42n
Java	long	42L
Python	int	42
C++	uint64_t	42
ProtoJSON	string	"42"

sint32

Signed 32-bit integer that uses ZigZag encoding to keep small negative numbers compact.

	TYPE	LITERAL
Go	int32	42
TypeScript	number	42
Java	int	42
Python	int	42
C++	int32_t	42
ProtoJSON	number	42

sint64

Signed 64-bit integer with ZigZag encoding. Use when values are frequently negative.

	TYPE	LITERAL
Go	int64	42
TypeScript	bigint	42n
Java	long	42L
Python	int	42
C++	int64_t	42
ProtoJSON	string	"42"

fixed32

Always 4 bytes, unsigned. Beats a varint only for values consistently above 2^{28}, like hashes.

	TYPE	LITERAL
Go	`uint32`	42
TypeScript	`number`	42
Java	`int`	42
Python	`int`	42
C++	`uint32_t`	42
ProtoJSON	`number`	42

fixed64

Always 8 bytes, unsigned. Beats a varint only for values consistently above 2^{56}, like hashes and large constants.

	TYPE	LITERAL
Go	`uint64`	42
TypeScript	`bigint`	42n
Java	`long`	42L
Python	`int`	42
C++	`uint64_t`	42
ProtoJSON	`string`	"42"

sfixed32

Always 4 bytes, signed. The fixed-width counterpart of int32.

	TYPE	LITERAL
Go	`int32`	42
TypeScript	`number`	42
Java	`int`	42
Python	`int`	42
C++	`int32_t`	42
ProtoJSON	`number`	42

sfixed64

Always 8 bytes, signed. The fixed-width counterpart of int64.

	TYPE	LITERAL
Go	`int64`	42
TypeScript	`bigint`	42n
Java	`long`	42L
Python	`int`	42
C++	`int64_t`	42
ProtoJSON	`string`	"42"

float

32-bit IEEE 754 floating point number.

	TYPE	LITERAL
Go	`float32`	3.14
TypeScript	`number`	3.14
Java	`float`	3.14f
Python	`float`	3.14
C++	`float`	3.14f
ProtoJSON	`number, or "NaN" / "Infinity"`	3.14

double

64-bit IEEE 754 floating point number.

	TYPE	LITERAL
Go	`float64`	3.14
TypeScript	`number`	3.14
Java	`double`	3.14
Python	`float`	3.14
C++	`double`	3.14
ProtoJSON	`number, or "NaN" / "Infinity"`	3.14



Well-Known Types

Every compiler ships a set of schemas already written for you, called well-known types, or WKTs for short. Google standardizes them, so you import `google.protobuf.Timestamp` or any other one without declaring a dependency yourself. Each WKT also gets its own JSON mapping, so a `Timestamp`, for example, appears in JSON as an RFC 3339 string. Full definitions are in the google.protobuf reference.

General

google.protobuf.Any

Holds an arbitrary serialized message and a type URL that identifies it. The payload can vary at runtime.

	TYPE	LITERAL
Go	`*anypb.Any`	`anypb.New(msg)`
TypeScript	`Any`	`anyPack(UserSchema, user)`
Java	`com.google.protobuf.Any`	`Any.pack(msg)`
Python	`protobuf.wkt.Any`	`Any.pack(msg)`
C++	`google::protobuf::Any`	`any.PackFrom(msg)`
ProtoJSON	`object with "@type"`	`{"@type": "...", ...}`

google.protobuf.Timestamp

A point in time, independent of timezone. Maps to RFC 3339 in JSON.

	TYPE	LITERAL
Go	`*timestamppb.Timestamp`	`timestamppb.Now()`
TypeScript	`Timestamp`	`timestampNow()`
Java	`com.google.protobuf.Timestamp`	`Timestamps.fromMillis(ms)`
Python	`protobuf.wkt.Timestamp`	`Timestamp.now()`
C++	`google::protobuf::Timestamp`	`TimeUtil::GetCurrentTime()`
ProtoJSON	`RFC 3339 string`	`"2026-01-15T10:00:00Z"`

google.protobuf.Duration

A span of time. Maps to a string ending in 's' in JSON (e.g. '1.5s').

	TYPE	LITERAL
Go	`*durationpb.Duration`	`durationpb.New(30 * time.Second)`
TypeScript	`Duration`	`create(DurationSchema, { seconds: 30n })`
Java	`com.google.protobuf.Duration`	`Durations.fromSeconds(30)`
Python	`protobuf.wkt.Duration`	`Duration.from_seconds(30)`
C++	`google::protobuf::Duration`	`TimeUtil::SecondsToDuration(30)`
ProtoJSON	`string ending in "s"`	`"30s"`

google.protobuf.Empty

Marks an API that takes no parameters, or returns nothing.

	TYPE	LITERAL
Go	`*emptypb.Empty`	`&emptypb.Empty{}`
TypeScript	`Empty`	`create(EmptySchema)`
Java	`com.google.protobuf.Empty`	`Empty.getDefaultInstance()`
Python	`protobuf.wkt.Empty`	`Empty()`
C++	`google::protobuf::Empty`	`Empty()`
ProtoJSON	`{}`	`{}`

google.protobuf.FieldMask

A set of symbolic field paths. They mark which fields a read or update should touch.

	TYPE	LITERAL
Go	`*fieldmaskpb.FieldMask`	`fieldmaskpb.New(msg, "name")`
TypeScript	`FieldMask`	`create(FieldMaskSchema, { paths: ["name"] })`
Java	`com.google.protobuf.FieldMask`	`FieldMaskUtil.fromString("name")`
Python	`protobuf.wkt.FieldMask`	`FieldMask(paths=["name"])`
C++	`google::protobuf::FieldMask`	`FieldMaskUtil::FromString("name", &mask)`
ProtoJSON	`comma-separated string`	`"name"`

{ } Dynamic JSON

google.protobuf.Struct

Maps directly to a free-form JSON object.

	TYPE	LITERAL
Go	`*structpb.Struct`	`structpb.NewStruct(m)`
TypeScript	`Struct`	`fromJson(StructSchema, { k: "v" })`
Java	`com.google.protobuf.Struct`	`Struct.newBuilder().putFields(…)`
Python	`protobuf.wkt.Struct`	`Struct.from_python({"k": "v"})`
C++	`google::protobuf::Struct`	`(*s.mutable_fields())["k"]`
ProtoJSON	`object`	`{"k": "v"}`

google.protobuf.Value

Represents a dynamically typed value, equivalent to any JSON type.

	TYPE	LITERAL
Go	`*structpb.Value`	`structpb.NewStringValue("x")`
TypeScript	`Value`	`fromJson(ValueSchema, "x")`
Java	`com.google.protobuf.Value`	`Values.of("x")`
Python	`protobuf.wkt.Value`	`Value.from_python("x")`
C++	`google::protobuf::Value`	`v.set_string_value("x")`
ProtoJSON	`any JSON value`	`"x"`

Like every well-known type, these are ordinary messages, defined in plain `.proto` files. `struct.proto` defines `Struct` and `Value`. `wrappers.proto` defines the wrapper types, though you rarely need those now that scalar fields track presence by default.

One more family of WKTs describes APIs and types themselves. They power runtime reflection and API tooling, and application code seldom uses them directly. The main ones are `Method` in `api.proto`, `Type` in `type.proto`, and `SourceContext` in `source_context.proto`.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

import "google/protobuf/timestamp.proto";
import "google/protobuf/duration.proto";

message Job {
  google.protobuf.Timestamp scheduled_at = 1;
  google.protobuf.Duration timeout = 2;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "scheduledAt": "2026-01-15T10:00:00Z",
  "timeout": "30s"
}
```

{} ProtoJSON Mapping

Protobuf's wire format is binary, but it also defines a standard ProtoJSON mapping: the rules for how each type appears in JSON, so JSON written by one library can be read by any other. It does not pin down the exact output. Whitespace, field order, and serializer options can differ between libraries and versions, so don't hash, sign, or byte-compare ProtoJSON.

PROTOBUF TYPE	JSON TYPE(S)	JSON VALUE EXAMPLE	NOTES

`message`

JSON Object

EXAMPLE `{"userName": "hiro"}`

Serialized as a JSON object. Field names are mapped to lowerCamelCase by default, or the json_name option if set.

`repeated`

JSON Array

EXAMPLE `["a", "b"]`

Serialized as a JSON array.

`map<K, V>`

JSON Object

EXAMPLE `{"k": "v"}`

Serialized as a JSON object.

`int32, uint32`

JSON Number

EXAMPLE `42`

Standard JSON numbers.

`float, double`

JSON Number

EXAMPLE `123.45`

Standard JSON numbers.

`bool`

JSON `Boolean`

EXAMPLE `true`

Standard JSON booleans.

`int64, uint64`

JSON `String`

EXAMPLE `"9007199254740993"`

Strings prevent precision loss in JS.

`enum`

JSON `String`

EXAMPLE `"ROLE_ADMIN"`

Uses the string name of the enum value.

`bytes`

JSON `String`

EXAMPLE `"NDI="`

Base64 encoded string.

`google.protobuf.Timestamp`

JSON `String`

EXAMPLE `"2023-10-01T12:00:00Z"`

RFC 3339 formatted timestamp string.

`google.protobuf.Duration`

JSON `String`

EXAMPLE `"1.000340012s"`

Seconds with up to 9 fractional digits.

`google.protobuf.FieldMask`

JSON `String`

EXAMPLE `"f.a,f.b"`

Comma-separated paths as a single string.

`google.protobuf.Struct`

JSON Object

EXAMPLE `{"foo": "bar"}`

Standard representation for a generic JSON object.

`google.protobuf.Value`

JSON Any

EXAMPLE `"foo" or 123`

Can be any valid JSON value (null, number, string, boolean, struct, or list).

`google.protobuf.NullValue`

JSON null

EXAMPLE `null`

The JSON null value.

`google.protobuf.Empty`

JSON Object

EXAMPLE `{}`

An empty JSON object.

64-bit Precision

Every number in JavaScript is a 64-bit float, which can only represent integers exactly up to 2^{53} - 1. Past that, values get rounded. Parsing `9007199254740993` as a JSON number rounds it to `9007199254740992`.

That is why 64-bit integer types (`int64`, `fixed64`, `uint64`, `sint64`, and `sfixed64`) are encoded as strings in JSON, not numbers.

>_ SCHEMA.PROTO

```
edition = "2024";

package demo.v1;

message Record {
  int64 big_number = 1;
  bytes raw = 2;
  double ratio = 3;
}
```

>_ EXAMPLE MESSAGE (JSON)

```
{
  "bigNumber": "9007199254740993"
}
```

Further reading: Types

Protobuf Tip #10: Choosing the right integer type
https://buf.build/blog/choosing-the-right-protobuf-integer-type

int32, sint64, fixed32: which integer type to reach for, and what each costs as a varint.

APPENDIX B

Quiz

The full question bank behind the site's quiz. On the site each run draws ten questions at random from a level; here every question is listed, with the answers and their explanations on their own pages at the end of each level.

Normal level

Fields, types, defaults, and how schemas change over time.

1 What identifies a field in Protobuf's binary format?

A Its field name

B Its declared type

C Its line order in the `.proto` file

D Its field number

2 What must be true of field numbers within a single message?

A Fields must be listed in ascending numeric order

B Nested messages must continue the parent's numbering

C Every field must have a unique number

D Numbers must be consecutive with no gaps

3 What does this `service` block define?

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

A An RPC contract that code generators turn into client and server stubs

B An embedded message containing a request field and a response field

C An HTTP `GET /GetUser` route bound to URL query parameters

D The server handler that queries the database and returns a user record

4 Why does this enum start with `STATUS_UNSPECIFIED = 0` instead of `STATUS_ACTIVE = 0`?

```
edition = "2024";

enum Status {
  STATUS_UNSPECIFIED = 0;
  STATUS_ACTIVE = 1;
  STATUS_SUSPENDED = 2;
}
```

A The compiler requires the first constant name to end in `_UNSPECIFIED`

B The zero entry is the default value when an enum field is unset

C Zero is reserved for unknown values received from newer schemas

D Enum constants must be numbered sequentially starting from zero

5 A binary payload was written before `username` was renamed to `handle`. What happens when code compiled from the new schema parses it?

```
// Before
string username = 1;

// After
string handle = 1;
```

A Parsing fails with a field name mismatch error

B The value decodes into `handle` normally

C The value is stored in the message's unknown field set

D The field reads as `""` unless a migration alias is declared

6 Your code sets `email` to `"a@example.com"` and then sets `phone` to `"+15550100"`. What is the state of `contact`?

```
oneof contact {
  string email = 1;
  string phone = 2;
}
```

A Both `email` and `phone` remain set on the message

B Only `phone` is set; setting it clears `email`

C Only `email` is set; the first assignment wins

D Setting a second member throws a runtime error

7 When removing a field from a message, how do you prevent future edits from reusing its field number or name?

```
message Order {
  reserved 4, 9 to 11;
  reserved coupon_code, legacy_discount;
  string id = 1;
}
```

A Prefix the removed field's name with an underscore `_`

B Add the removed field's number and name to `reserved` statements

C Leave a comment with the old field number so the compiler skips it

D Keep the old field in the message with `[deprecated = true]`

8 Which well-known type represents an instant in time such as `created_at`?

A `google.protobuf.DateTime`

B `google.protobuf.Instant`

C `google.protobuf.Duration`

D `google.protobuf.Timestamp`

9 When this message is serialized with ProtoJSON, what key name is emitted for `first_name`?

```
message User {
  string first_name = 1;
  bool verified = 2;
}
```

A `"User.first_name"` (qualified by message name)

B `"first_name"` (kept in `snake_case`)

C `"1"` (keyed by field number)

D `"firstName"` (converted to `lowerCamelCase`)

10 Which declaration lets `tags` hold an ordered list of zero or more strings?

A `repeated string tags = 1;`

B `string[] tags = 1;`

C `map<int32, string> tags = 1;`

D `list<string> tags = 1;`

11 What does the `package` declaration in this file control?

```
edition = "2024";

package acme.billing.v1;

message Invoice {
  string id = 1;
}
```

A It selects the remote registry where the schema is published

B It forces the `.proto` file to be stored at `acme/billing/v1/`

C It locks code generation to a single target language

D It namespaces the file's Protobuf types as `acme.billing.v1.*`

12 Where must `edition = "2024";` appear in an Edition 2024 `.proto` file?

A Inside an `option (google.protobuf.edition) = "2024";` statement

B Immediately after the `package` declaration at the top of the file

C Inside each `message` block that uses Edition 2024 features

D As the first non-comment statement, before `package` or `import`

13 What happens if a `.proto` file omits both `edition` and `syntax` declarations?

A The compiler defaults to legacy `proto2` syntax

B The compiler defaults to `proto3` syntax

C Compilation fails immediately with a syntax error

D The compiler defaults to the latest supported edition

14 Why is it good practice to assign field numbers `1` through `15` to the most frequently populated fields?

A Field numbers above `15` cannot use packed repeated encoding

B Parsers decode fields `1` through `15` before reading higher numbers

C Their field tags encode in a single byte instead of two or more

D Only fields `1` through `15` can be used in `map` or `repeated` fields

15 In this message, `retries` and `label` are never set. What do their getters return?

```
edition = "2024";

message JobConfig {
  int32 retries = 1;
  string label = 2;
}
```

A `0` for `retries` and `""` for `label`

B An uninitialized-field exception on access

C `-1` for `retries` and `""` for `label`

D `null` for both `retries` and `label`

16 What is required for `RUNNING` and `IN_PROGRESS` to share the enum value `1`?

```
enum TaskState {
  TASK_STATE_UNSPECIFIED = 0;
  TASK_STATE_RUNNING = 1;
  TASK_STATE_IN_PROGRESS = 1;
}
```

A Mark `TASK_STATE_IN_PROGRESS` with `[deprecated = true]`

B Add `option allow_alias = true;` inside the `TaskState` enum body

C Nothing; enum constants can share numeric values by default

D Declare `reserved 1;` at the top of the `TaskState` enum body

17 Which of these `map` declarations will fail to compile?

A `map<int64, string> by_id = 1;`

B `map<EnumStatus, string> by_status = 1;`

C `map<string, User> by_name = 1;`

D `map<bool, int32> by_flag = 1;`

18 How do you represent a list of maps when `repeated map<string, string>` is not valid syntax?

A Nest two maps directly as `map<int32, map<string, string>>`

B Declare `map<string, repeated string>` to store a list per key

C Wrap the `map` in a message and mark that message field `repeated`

D Enable `option allow_repeated_maps = true;` at the file level

19 Which field declaration is illegal inside a `oneof` block?

```
message Filter {
  oneof criterion {
    string query = 1;
    // Which line below is invalid?
  }
}
```

A `repeated string tags = 2;`

B `int64 min_score = 2;`

C `DateRange created_between = 2;`

D `bytes raw_token = 2;`

20 File `order.proto` uses `acme.common.v1.Money`, defined in `acme/common/v1/money.proto`. What is needed in `order.proto`?

A Add `include "acme/common/v1/money.proto";` inside the `Order` message

B Put `order.proto` in the `acme.common.v1` package so it auto-imports

C Nothing; the compiler resolves all `.proto` files in the directory tree

D Add `import "acme/common/v1/money.proto";` near the top of `order.proto`

21 You move `User` from `types.proto` into `user.proto`, and want existing files that `import "types.proto";` to keep compiling. What should `types.proto` contain?

A `import public "user.proto";`

B `import "user.proto";`

C `option moved_to = "user.proto";`

D `export "user.proto";`

22 How does `ShippingOrder` reference the `Address` message declared inside `Customer`?

```
message Customer {
  message Address {
    string street = 1;
  }
  string id = 1;
}
```

A `Address` cannot be referenced outside `Customer`

B `Customer::Address destination = 1;`

C `Customer/Address destination = 1;`

D `Customer.Address destination = 1;`

23 When should you choose `bytes` over `string` for a field in a `.proto` schema?

A When the payload holds raw binary data that is not valid UTF-8 text

B Whenever the field should be omitted from ProtoJSON serialization

C Whenever a text value might exceed 64 KB in serialized length

D When the text contains emojis or non-ASCII Unicode characters

24 Which well-known type is designed for a signed span of time such as `request_timeout`?

A `google.protobuf.Timestamp`

B `google.protobuf.Duration`

C `google.protobuf.Interval`

D `google.protobuf.TimeSpan`

25 What problem does `google.protobuf.Any` solve in a message schema?

```
import "google/protobuf/any.proto";

message AuditEvent {
  string event_id = 1;
  google.protobuf.Any detail = 2;
}
```

A It stores a dynamically typed scalar (`int32` , `string` , or `bool`) without a wrapper

B It acts as an abstract base message that other messages inherit from

C It embeds a serialized Protobuf message alongside a `type_url` identifying its schema

D It accepts arbitrary unvalidated JSON objects without converting them to Protobuf

26 When should you reach for `google.protobuf.Struct` or `google.protobuf.Value` instead of `google.protobuf.Any` ?

A When you need to store raw binary blobs such as compressed archives

B When modeling dynamic, schema-less JSON data that has no `.proto` definition

C When you want the smallest binary representation for strongly typed messages

D When defining RPC envelopes that carry typed Protobuf domain events

27 What is the effect of adding `[deprecated = true]` to a field?

```
message User {
  string id = 1;
  string legacy_fax = 2 [deprecated = true];
}
```

A The compiler rejects any application code that reads or writes the field

B Generated accessors are marked deprecated, while wire and JSON behavior stay unchanged

C Serializers stop writing the field on the wire, and `protoc` reserves its number

D Parsers silently drop the field when decoding incoming binary payloads

28 An existing REST API expects the JSON key `"user_id"` in `snake_case` instead of `"userId"`. How can a Protobuf schema match that contract?

```
message Account {
  string user_id = 1 [json_name = "user_id"];
}
```

A Set `[json_name = "user_id"]` on the field so ProtoJSON emits `"user_id"`

B Name the field `USER_ID` in all caps to disable camelCase conversion

C ProtoJSON always emits `lowerCamelCase` and cannot be overridden per field

D Change the field number to match the JSON property position

29 In Edition 2024, how do you declare a scalar field that does NOT track explicit presence (matching `proto3`'s default behavior)?

```
edition = "2024";

message Counter {
  int32 value = 1 [features.field_presence = IMPLICIT];
}
```

A Annotate the field with the `[optional = false]` option

B Place the scalar field inside a single-field `oneof` block

C Set `features.field_presence = IMPLICIT` on the field or file

D Omit the `optional` keyword when declaring the field

30 You are reading an older `syntax = "proto3";` file and see `optional int32 timeout_ms = 3;`. Why was `optional` written there?

A It omitted the field's tag byte on the wire for non-zero values

B It opted the scalar field into explicit presence tracking (`has_timeout_ms`)

C Without `optional`, `proto3` treated every scalar field as required

D It allowed the field to hold multiple values like a `repeated` field

31 Why is `features.field_presence = LEGACY_REQUIRED` (the old `required` label from `proto2`) strongly discouraged?

A Required fields cannot be assigned field numbers between `1` and `15`

B Missing required fields cause parse failures, so the field can never be safely stopped

C Required fields force every integer in the message to use 8-byte fixed encoding

D Required fields cannot be serialized to ProtoJSON and only work in binary mode

32 What does this `buf.validate.field` annotation do?

```
import "buf/validate/validate.proto";

message CreateUserRequest {
  string email = 1 [(buf.validate.field).string.email = true];
  int32 age = 2 [(buf.validate.field).int32.gte = 18];
}
```

A It alters the binary wire format so invalid strings cannot be serialized

B It generates SQL `CHECK` constraints directly inside the `protoc` compiler

C It makes `protoc` reject the `.proto` file if default values break the rules

D It attaches schema validation rules that `protovalidate` checks at runtime

33 What kind of RPC method does this declaration define?

```
service TelemetryService {
  rpc RecordMetrics(stream MetricPoint) returns (IngestSummary);
}
```

A Unary batching: the client sends a single repeated array of `MetricPoint` items

B Bidirectional streaming: both client and server send independent message streams

C Client-streaming: the client streams `MetricPoint` messages and gets one `IngestSummary`

D Server-streaming: the client sends one `MetricPoint` and reads a stream of summaries

34 Why should an RPC method use a dedicated `GetBookRequest` message instead of a shared or primitive parameter?

```
service LibraryService {
  rpc GetBook(GetBookRequest) returns (GetBookResponse);
}
```

A Reusing a request message across two RPCs is a compile error in `protoc`

B Only messages ending in `Request` and `Response` are accepted by `service` blocks

C You can add new request fields later without breaking the method signature

D Bare scalars like `rpc GetBook(string)` are valid, but wrapper messages compress better

35 How does ProtoJSON serialize an `int64` or `uint64` field by default?

```
message Transaction {
  int64 amount_micros = 1;
}
```

A As a base64-encoded 8-byte binary string

B As an unquoted JSON number, such as `9007199254740993`

C As a quoted decimal string, such as `"9007199254740993"`

D As a two-element JSON array `[high32, low32]`

36 When `Status.STATUS_ACTIVE` (`1`) is serialized with ProtoJSON, how is the value represented by default?

```
enum Status {
  STATUS_UNSPECIFIED = 0;
  STATUS_ACTIVE = 1;
}
```

A As an object `{"name": "STATUS_ACTIVE", "number": 1}`

B As the enum constant name string `"STATUS_ACTIVE"`

C As the lowercase string `"active"` with the prefix stripped

D As the integer number `1` in the JSON field

37 Why does this file fail to compile when both enums are declared at the top level of the same package?

```
edition = "2024";
package acme.store.v1;

enum OrderStatus {
  UNSPECIFIED = 0;
  ACTIVE = 1;
}

enum SubscriptionStatus {
  UNSPECIFIED = 0;
  ACTIVE = 1;
}
```

A A `.proto` file can only declare one top-level `enum` definition per file

B Sibling enum values share the enclosing scope, so `UNSPECIFIED` and `ACTIVE` collide

C Every enum constant must include a numeric version suffix such as `_V1`

D Two enums in the same package cannot reuse the integer numbers `0` and `1`

38 You want to replace `int32 price_cents = 3;` with `string price_display` in a deployed schema. What is the safest way to make this change?

A Put both `int32 price_cents = 3;` and `string price_display = 3;` inside a `oneof`

B Reserve or deprecate field `3`, and add `string price_display = 4;` with a new number

C Change field `3` in place from `int32` to `string`; parsers convert it automatically

D Change field `3` to `string` and add `[deprecated = true]` to ignore wire mismatches

39 In Protobuf, does a singular message field (like `Address billing_address = 2;`) track field presence?

A Yes; singular message fields always track presence across all syntax versions

B Only when `features.field_presence = EXPLICIT` is set on the field

C No; an unset message field is identical to an empty `Address{}` instance

D Only when the message field is declared inside a `oneof` block

40 Can a `repeated` or `map` field distinguish between 'never set' and 'set to an empty list/map'?

A Yes; in Edition 2024 `repeated` fields expose a `has_*` presence check by default

B Yes; annotating the field with `features.field_presence = EXPLICIT` enables it

C No; an empty `repeated` or `map` field writes zero bytes and looks identical to unset

D Only in ProtoJSON (`[]` vs omitted), not in the binary wire format

41 If you insert keys `"b"`, `"a"`, and `"c"` into a `map<string, int32>` field and serialize the message, what order does a decoder see?

A Map iteration order is undefined; callers cannot rely on any specific order

B Insertion order (`"b"`, `"a"`, `"c"`) is preserved by the wire format

C Lexicographical key order (`"a"`, `"b"`, `"c"`) is guaranteed by all parsers

D Descending value order, from largest `int32` value to smallest

42 How do tools like `protoc` and `buf generate` support code generation for languages outside the compiler's built-in targets?

A They transpile generated C++ classes into the target language using LLVM

B They require each runtime to parse `.proto` text files at application startup

C They execute JavaScript generator scripts embedded inside `option` blocks

D They invoke `protoc-gen-<name>` plugin binaries, passing descriptors over `stdin`

43 What is `google.protobuf.FieldMask` commonly used for in `Update*` or `Get*` RPCs?

```
import "google/protobuf/field_mask.proto";

message UpdateUserRequest {
  User user = 1;
  google.protobuf.FieldMask update_mask = 2;
}
```

A Specifying which field paths (like `"display_name"`) the call reads or updates

B Mapping old field numbers to new field numbers during a schema migration

C Filtering repeated lists on the server using SQL-like `WHERE` expressions

D Encrypting sensitive fields in `User` before writing the message to the wire

44 Why do API style guides (including `buf lint`) recommend returning a dedicated `DeleteUserResponse {}` message instead of `google.protobuf.Empty`?

A `google.protobuf.Empty` is deprecated in Edition 2024 and rejected by `protoc`

B `google.protobuf.Empty` cannot be serialized to ProtoJSON by standard runtimes

C You can add fields to `DeleteUserResponse` later without changing the RPC return type

D `google.protobuf.Empty` takes 4 bytes on the wire, while `DeleteUserResponse {}` takes 0

45 What is the effect of file-level options such as `option go_package` or `option java_package`?

```
edition = "2024";
package acme.ledger.v1;

option go_package = "github.com/acme/ledger/gen/acme/ledger/v1;ledgerv1";
option java_package = "com.acme.ledger.v1";
```

A They restrict the `.proto` file so it only compiles for those two languages

B They override the Protobuf type name used inside `google.protobuf.Any` URLs

C They prefix every binary message payload with the language package string

D They set the generated code's package path without changing the wire format

46 In this Edition 2024 file, which fields use `IMPLICIT` field presence?

```
edition = "2024";
option features.field_presence = IMPLICIT;

message Metric {
  string name = 1;
  int64 value = 2;
  string unit = 3 [features.field_presence = EXPLICIT];
}
```

A `name` and `value` use `IMPLICIT`; `unit` overrides it to `EXPLICIT`

B All three fields use `IMPLICIT` because field-level overrides are not allowed

C All three fields use `EXPLICIT` because file-level features are ignored

D Only `name` uses `IMPLICIT`; `value` and `unit` use `EXPLICIT`

47 A `repeated string roles = 1;` field is populated with `["admin", "user", "admin"]`. What does the receiver get after decoding?

A `["admin", "user", "admin"]` in the exact same order, with duplicates intact

B A parse error because `repeated` fields require unique elements

C `["admin", "user"]` with duplicate values automatically deduplicated

D `["admin", "admin", "user"]` sorted alphabetically by the encoder

48 If you explicitly set `fixed_discount_cents = 0` on this `oneof`, what does the `discount` case check report?

```
message Coupon {
  oneof discount {
    int32 fixed_discount_cents = 1;
    int32 percent_off = 2;
  }
}
```

A `fixed_discount_cents` is active and serialized on the wire even though its value is `0`

B Both `fixed_discount_cents` and `percent_off` report as active with `0`

C `discount` is reported as unset because `0` is the default value for `int32`

D Setting a `oneof` member to `0` clears the `oneof` and omits it from the wire

49 What is a practical advantage of using `google.protobuf.Timestamp` over `int64 created_at` for timestamps?

A It encodes in 2 bytes on the wire instead of up to 10 bytes for `int64`

B It avoids unit ambiguity (seconds vs. ms) and formats as RFC 3339 in ProtoJSON

C It can be used as a `map` key, whereas `int64` cannot be a `map` key

D It automatically converts the timestamp to the client's local timezone on parse

50 You add a new field `string display_name = 4;` to `User`. What happens when a server running the new schema parses an older `User` payload that only had fields `1` – `3`?

A The message is rejected unless `display_name` is marked `[deprecated = true]`

B The parser populates `display_name` with the string value of field `1`

C Parsing fails because field `4` is missing from the incoming byte stream

D The message parses normally, `has_display_name` is `false`, and its getter returns `""`

51 A backend adds a new field `string tier = 5;` to a response message and serializes it as ProtoJSON. How do standard ProtoJSON parsers on older clients behave by default?

A They map unknown JSON keys into the first `string` field in the message

B They always ignore unknown JSON keys without any parser configuration

C They reject unknown JSON keys by default unless configured to ignore unknown fields

D They preserve unknown JSON keys in the binary unknown field set automatically

52 In this `Order` message, how are `billing_address` and `shipping_address` distinguished on the wire?

```
message Order {
  Address billing_address = 1;
  Address shipping_address = 2;
}
```

A Two fields in the same message cannot share the same embedded message type

B By shifting `shipping_address` 's internal `Address` field numbers up by 100

C By embedding `"billing_address"` or `"shipping_address"` inside each `Address`

D By their outer field numbers (`1` and `2`) on `Order` , while reusing `Address`

53 Which statement about numbering constants inside an `enum` is true?

```
enum Priority {
  PRIORITY_UNSPECIFIED = 0;
  PRIORITY_LOW = 10;
  PRIORITY_HIGH = 100;
}
```

A Enum constants must be powers of two (`0` , `1` , `2` , `4` , `8`) for bitmasking

B An enum with value `100` pads the binary message with 100 bytes on the wire

C Enum numbers must be contiguous (`0` , `1` , `2`), so `10` and `100` fail to compile

D Gaps between enum numbers (like `0` , `10` , `100`) are valid and cost no extra space

54 What does a schema breaking-change detector like `buf breaking` compare?

A Application source code for calls to methods marked `[deprecated = true]`

B Binary serialization throughput between two different versions of `protoc`

C Descriptors from the current and baseline schemas to catch wire, JSON, or code breaks

D Your service's unit test suite against live production traffic in a staging cluster

55 When should you use `uint32` or `uint64` instead of `int32` or `int64`?

A When the field is used as a `map` key, since `int32` is not allowed as a key

B Whenever the field uses ZigZag encoding for small negative numbers

C When values are non-negative and you want the full `2^32 - 1` or `2^64 - 1` range

D Whenever you want every integer to occupy a fixed 4 or 8 bytes on the wire

Answers: Normal level

1 D Each encoded field starts with a tag containing its field number and wire type. The field name is never written to the binary payload. (Basics > Fields)

2 C No two fields in the same message can share a number. Gaps and declaration order do not affect encoding, and nested messages have their own number space. (Basics > Fields)

3 A A `service` declares RPC methods with their request and response message types. RPC plugins such as Connect and gRPC generate typed clients and server interfaces from it. (Basics > Services)

4 B The first constant in an enum must have value `0`, which serves as the default value. Using an `_UNSPECIFIED` sentinel prevents an unset field from looking like an intentional `STATUS_ACTIVE`. (Basics > Enums)

5 B Binary payloads identify fields by number, not by name, so field `1` decodes into `handle`. Note that renaming a field can still break generated code and ProtoJSON payloads. (Advanced > Schema Evolution)

6 B A `oneof` holds at most one member at a time. Setting any member automatically clears whichever member was previously set. (Basics > Oneof)

7 B `reserved` makes the compiler reject any new field that reuses those numbers or names, protecting both binary payloads and ProtoJSON from silent corruption. (Advanced > Schema Evolution)

8 D `google.protobuf.Timestamp` represents a UTC instant as `seconds` and `nanos` relative to the Unix epoch, and serializes as an RFC 3339 string in ProtoJSON. (Types)

9 D ProtoJSON converts `snake_case` field names to `lowerCamelCase` when serializing. Compliant parsers accept both `"firstName"` and `"first_name"` on input. (Types)

10 A The `repeated` cardinality label turns a field into a dynamically sized, ordered list of elements of the given type. (Basics > Repeated Fields)

11 D `package` places the file's declarations into a Protobuf namespace (`acme.billing.v1.Invoice`), preventing name collisions when files import one another. (Basics > Packages)

12 D The `edition` statement (or `syntax` in older files) must be the very first non-comment line of a `.proto` file. (Advanced > Editions)

13 A For historical compatibility, a file without an `edition` or `syntax` header is parsed as `proto2`. Modern linters such as `buf lint` flag missing headers. (Advanced > Editions)

14 C A one-byte varint tag uses 1 continuation bit and 3 wire-type bits, leaving 4 bits for the field number (`1` to `15`). Starting at `16`, the tag takes two bytes. (Basics > Fields)

15 A Every scalar type has a deterministic default value: `0` for numeric types, `false` for `bool`, `""` for `string`, and an empty byte sequence for `bytes`. (Basics > Fields)

16 B Duplicate enum numbers are rejected unless `option allow_alias = true;` is set on the enum. (Basics > Enums)

17 B Map keys can be any integer type, `bool`, or `string`. Enums, floating-point types (`float`, `double`), `bytes`, and messages cannot be used as map keys. (Basics > Maps)

18 C A `map` cannot be `repeated`, and a `map` value cannot be another `map` or a `repeated` field. Wrapping the collection in a message and repeating that message is the standard pattern. (Basics > Maps)

19 A Fields inside a `oneof` cannot be `repeated` or `map` fields. If one branch of a `oneof` needs a list, wrap the `repeated` field in a separate message. (Basics > Oneof)

20 D Every `.proto` file must explicitly `import` the files whose definitions it references, even when both files share the same `package`. (Advanced > Imports)

21 A `import public` transitively re-exports the imported file's declarations to any file that imports `types.proto`, allowing schemas to be split or moved without breaking callers. (Advanced > Imports)

22 D Nested messages and enums can be referenced outside their enclosing message using dot notation (`Customer.Address`). (Basics > Composition)

23 A `string` fields must contain valid UTF-8 text (which includes all Unicode and emojis). Arbitrary binary payloads such as hashes, ciphertext, or images must use `bytes`. (Types)

24 B `google.protobuf.Duration` stores a signed span of time as `int64 seconds` and `int32 nanos`. (Types)

25 C `google.protobuf.Any` pairs a `string type_url` with `bytes value`, letting a message carry another Protobuf message whose concrete type isn't fixed in `AuditEvent`'s schema. (Advanced > The Any Type)

26 B `google.protobuf.Struct` and `google.protobuf.Value` directly represent arbitrary JSON structures (objects, arrays, strings, numbers, booleans, and `null`) without requiring a `.proto` schema for the payload. (Advanced > The Value Type)

27 B `[deprecated = true]` is a source-level signal: it emits deprecation annotations (`@Deprecated`, `#[deprecated]`, etc.) in generated code without altering wire or JSON behavior. (Advanced > Standard Options)

28 A The `json_name` field option overrides the default `lowerCamelCase` key emitted by ProtoJSON serializers. (Advanced > Standard Options)

29 C Edition 2024 defaults singular fields to `EXPLICIT` presence and removes the `optional` keyword. Setting `features.field_presence = IMPLICIT` opts into implicit presence. (Advanced > Field Presence)

30 B In `proto3`, plain scalar fields had implicit presence (unable to distinguish `0` from unset). `optional` was re-added in `proto3` to restore explicit `has_*` presence checks. (Advanced > Field Presence)

31 B Because parsers reject any message missing a `LEGACY_REQUIRED` field, a required field can never be safely deprecated or omitted, even by intermediate proxies. Runtime validation (`protovalidate`) is preferred instead. (Advanced > Required Fields)

32 D Protovalidate rules are custom field options stored in descriptors. Runtime libraries read those rules and validate message instances consistently across Go, Java, Python, TypeScript, and C++. (Validation)

33 C Placing `stream` before the request type only declares a client-streaming RPC. Placing `stream` on the response makes it server-streaming, and on both makes it bidirectional. (Basics > Services)

34 C Every Protobuf RPC takes one request message and returns one response message. Giving each RPC its own dedicated `FooRequest` and `FooResponse` lets you add fields over time without affecting other methods. (Basics > Services)

35 C Standard JSON parsers (such as `JSON.parse` in JavaScript) use IEEE 754 doubles, which lose integer precision above `2^53 - 1`. ProtoJSON quotes 64-bit integers as strings to prevent silent rounding. (Types)

36 B ProtoJSON emits the enum constant's name as a string by default, while parsers accept either the constant name (`"STATUS_ACTIVE"`) or the numeric value (`1`). (Types)

37 B Protobuf uses C++ scoping rules for enum values: sibling enums share their parent scope. Prefixing constants with the enum name (`ORDER_STATUS_UNSPECIFIED` , `SUBSCRIPTION_STATUS_UNSPECIFIED`) avoids collisions. (Basics > Enums)

38 B `int32` (wire type 0, varint) and `string` (wire type 2, length-delimited) are wire-incompatible. You should add a new field with a new number and deprecate or reserve the old one. (Advanced > Schema Evolution)

39 A Singular message fields always have explicit presence across `proto2` , `proto3` , and editions: you can always distinguish an unset `billing_address` from one set to an empty `Address` message. (Advanced > Field Presence)

40 C `repeated` and `map` fields never track presence. Whether you leave the field untouched or assign an empty list, nothing is written to the wire. If you must distinguish 'no list' from 'empty list', wrap the repeated field in a sub-message. (Advanced > Field Presence)

41 A Protobuf maps are unordered associative containers. Different languages and runtimes may iterate map entries in arbitrary order. (Basics > Maps)

42 D The compiler parses `.proto` files into descriptors and pipes a `CodeGeneratorRequest` to `protoc-gen-*` plugins, which return the generated source files in a `CodeGeneratorResponse` . (Basics > Generating Code)

43 A `FieldMask` holds a list of dot-separated field paths (serialized as a comma-separated camelCase string in ProtoJSON) so partial updates and projections can specify exactly which fields apply. (Types)

44 C On the wire, any empty message serializes to zero bytes. Starting with a dedicated `DeleteUserResponse {}` message costs nothing on the wire and lets you add fields (such as `deleted_at`) later without breaking the RPC signature. (Basics > Services)

45 D Language-specific file options (`go_package` , `java_package` , `csharp_namespace` , etc.) only tell that language's code generator how to organize generated files and namespaces. The wire format and Protobuf `package` remain unchanged. (Advanced > Standard Options)

46 A Features cascade down the lexical hierarchy (file → message → field). Setting `option features.field_presence = IMPLICIT;` at the file level applies to `name` and `value` , while `unit` overrides it back to `EXPLICIT` . (Advanced > Editions)

47 A A `repeated` field is an ordered list, not a set: element order and duplicate values are preserved on the wire. (Basics > Repeated Fields)

48 A `oneof` members always have explicit presence. Setting `fixed_discount_cents = 0` records that `fixed_discount_cents` is the active case and writes field `1` with value `0` to the wire. (Basics > Oneof)

49 B A bare `int64` leaves callers guessing whether it holds seconds, milliseconds, or nanoseconds, and serializes as a numeric string in JSON. `google.protobuf.Timestamp` has unambiguous `seconds` + `nanos` fields and an ISO 8601 / RFC 3339 ProtoJSON representation. (Types > Well-Known Types)

50 D Adding a new field is forward- and backward-compatible: payloads written before field `4` existed simply leave `display_name` unset, and its getter returns the default empty string `""`. (Advanced > Schema Evolution)

51 C Unlike the binary parser (which retains unknown fields by default), the ProtoJSON spec requires parsers to reject unknown field names by default unless the caller enables the `ignoreUnknownFields` option. (Advanced > Schema Evolution)

52 D Each field on `Order` has its own tag (`1` for `billing_address` , `2` for `shipping_address`), and each length-delimited payload inside encodes an `Address` using `Address` 's own field numbers. (Basics > Composition)

53 D After the required `0` first entry, enum constants can use any 32-bit integer values in any order, with arbitrary gaps. (Basics > Enums)

54 C `buf breaking` compares the compiled descriptors of your `.proto` files against a git branch, tag, or registry commit, catching deleted fields, changed types, or renamed JSON keys before they ship. (Advanced > Schema Evolution)

55 C `uint32` and `uint64` use standard varint encoding for non-negative integers up to `2^32 - 1` and `2^64 - 1` . If a field uses fixed 4/8-byte encoding, use `fixed32` / `fixed64` instead. (Types)

Hard level

The binary encoding, and what a decoder does with it.

1 How many bytes does the unsigned varint encoding of `300` occupy, and what are they?

A 2 bytes: `0x01 0x2C`

B 2 bytes: `0xAC 0x02`

C 1 byte: `0x2C`

D 4 bytes: `0x2C 0x01 0x00 0x00`

2 A serialized message starts with the tag byte `0x08`. What field number and wire type does it encode?

A Field `0`, wire type `8`

B Field `1`, wire type `2` (length-delimited)

C Field `1`, wire type `0` (varint)

D Field `8`, wire type `0` (varint)

3 For a field that frequently holds small negative numbers like `-1` or `-5`, why is `sint32` smaller on the wire than `int32`?

A `int32` rejects negative numbers at serialization time

B `sint32` uses a fixed 2-byte integer layout for all values

C `int32` uses 10 bytes for negatives; `sint32` ZigZag-encodes them

D `sint32` supports a wider numeric range than `int32`

4 Which wire type is used to encode an embedded message field?

A `0` (varint)

B `5` (32-bit fixed)

C `2` (length-delimited)

D `1` (64-bit fixed)

5 With this Edition 2024 schema, how is `count` encoded when a caller explicitly sets `count = 0` versus leaving `count` unset?

```
edition = "2024";

message Stats {
  int32 count = 1;
}
```

A Both write zero bytes to the wire; `0` and unset are identical

B Explicit `0` writes `0x08 0x00` (2 bytes); unset writes 0 bytes

C Explicit `0` writes 0 bytes; unset writes a null tag `0x08 0xFF`

D Both write `0x08 0x00` because Edition 2024 serializes all fields

6 What are the exact bytes on the wire for `Batch` when `ids` is set to `[1, 2, 3]`?

```
edition = "2024";

message Batch {
  repeated int32 ids = 1;
}
```

A `0A 03 01 02 03` (tag `0x0A`, byte length `0x03`, then `01 02 03`)

B `0A 0C 01 00 00 00 02 00 00 00 03 00 00 00` (fixed 32-bit ints)

C `08 01 08 02 08 03` (tag `0x08` repeated before each varint)

D `08 03 01 02 03` (tag `0x08`, element count `0x03`, then `01 02 03`)

7 Service A (new schema) sends a message with field `5` to Service B (old schema, fields `1`–`4`). Service B updates field `1` and re-serializes. What happens to field `5`?

A Service B moves field `5` into field `4` during re-serialization

B Service B drops field `5` when re-serializing the message

C Service B keeps field `5` as an unknown field and re-serializes it

D Service B fails to parse the incoming message

8 In an Edition 2024 file with default features, what happens when a parser decodes an enum value (`99`) that is not declared in the enum?

A The unrecognized number `99` is stored directly in the enum field

B The parser overwrites the unrecognized value with `0`

C The value `99` is stripped and moved to the unknown field set

D Deserialization fails with an invalid enum value error

9 A binary payload contains two records for the same singular `int32` field `1`: `08 05 08 09`. What value does the parser produce for field `1`?

A `5` (the first occurrence wins)

B `14` (the parser sums duplicate varints)

C `9` (the last occurrence wins)

D Parsing fails because duplicate scalar tags are malformed

10 Which of these field numbers will `protoc` reject as reserved for the Protobuf implementation?

A `100000`

B `15`

C `19500`

D `536870911`

11 What is the tag byte in hexadecimal for field number `2` with wire type `2` (length-delimited)?

A `0x0A`

B `0x12`

C `0x22`

D `0x10`

12 How is the tag for field number `16` with wire type `0` (varint) encoded on the wire?

A `0x80 0x01`

B `0x01 0x80`

C `0x10`

D `0x80 0x10`

13 Why does `-1` in an `int32` field always serialize as 10 varint bytes (`FF FF FF FF FF FF FF FF FF 01`)?

A Negative `int32` values are sign-extended to 64 bits before varint encoding

B A 32-bit two's-complement negative integer requires 10 groups of 3 bits

C Wire type `0` reserves the first 6 bytes for an overflow header on signed fields

D The encoder pads negative numbers to 10 bytes to distinguish them from `sint32`

14 A schema changes `int32 delta = 1;` to `sint32 delta = 1;`. An old payload with `delta = 2` (`08 02`) is parsed by the new code. What value is decoded?

A `2` (positive integers decode identically in both types)

B `1` (ZigZag decodes unsigned `2` as `+1`)

C Parsing fails with a wire type mismatch error

D `-1` (ZigZag maps even numbers to negative values)

15 How does ProtoJSON format a `google.protobuf.Any` that packs a regular message `acme.v1.Order` versus one that packs `google.protobuf.Timestamp`?

A Both place their serialized binary bytes as Base64 under `"value"` with `"typeUrl"`

B `Order` is Base64-encoded under `"value"`; only well-known types expand inline

C Both nest their JSON representation inside a `"payload"` object alongside `"@type"`

D `Order` merges its fields inline with `"@type"`; `Timestamp` nests under a `"value"` key

16 How is `map<string, int32> scores = 3;` represented in the binary wire format?

A As a dedicated wire type `6` dictionary record sorted by UTF-8 key bytes

B As two parallel packed repeated fields at field `3` (keys) and field `4` (values)

C As a single wire-type-`2` block at field `3` of alternating strings and varints

D As repeated sub-messages at field `3`, each with `key = 1` and `value = 2`

17 What does a binary parser do if a `map<string, int32>` field contains two entry records with the key `"alice"` (`10` then `25`), or an entry missing its `value` field?

A Duplicate keys keep the last entry (`25`); a missing `value` defaults to `0`

B Duplicate keys fail parsing; a missing `value` omits the key from the map

C Duplicate keys keep the first entry (`10`); a missing `value` fails parsing

D Duplicate keys sum their values (`35`); a missing `value` defaults to `-1`

18 In a binary payload, a singular message field `Profile profile = 1;` appears twice: first with `name = "Ada"`, `role = "user"`, and second with `role = "admin"`. What is decoded?

A `name = "Ada"` and `role = "admin"` (the two occurrences are merged)

B Parsing fails because singular message fields cannot appear twice

C `name = ""` and `role = "admin"` (the second occurrence replaces the first)

D `name = "Ada"` and `role = "user"` (later occurrences are ignored)

19 Why is the maximum valid field number in Protobuf `536,870,911` (`2^29 - 1`)?

A A 32-bit unsigned tag reserves 3 bits for the wire type, leaving 29 bits

B `FileDescriptorProto` stores field numbers in a 29-bit bitfield on disk

C Signed 32-bit field numbers reserve the top 2 bits for `oneof` and `map` flags

D A 5-byte varint stores at most 29 payload bits alongside 6 header bits

20 Field `int32 score = 1` uses `features.field_presence = IMPLICIT`. A sender sets `score = 42`, then sets `score = 0`, and serializes the message. What is written for `score`?

A Two bytes (`0x08 0x00`) in binary, but the field is omitted in ProtoJSON

B Zero bytes; an implicit-presence field holding `0` is omitted from the wire

C Two bytes (`0x08 0x00`), because `score` was mutated before serialization

D One byte (`0x08`) with no value byte, signaling a reset to default

21 How does `protoc` (or `buf generate`) pass the parsed schema to a `protoc-gen-*` plugin binary?

A It pipes the raw `.proto` source text over `stdin` and reads generated source files from `stdout`

B It writes a temporary JSON AST file to disk and passes its path via `--plugin_ast_out`

C It writes a binary `CodeGeneratorRequest` to `stdin` and reads `CodeGeneratorResponse` from `stdout`

D It loads the plugin as a C shared library (`.so` / `.dylib`) and passes `DescriptorPool*` pointers

22 Which bitwise expression ZigZag-encodes a signed 32-bit integer `n` (using arithmetic right shift)?

A `(n >> 1) ^ (n << 31)`

B `~(n << 1) + (n >> 31)`

C `(n << 1) | (n >>> 31)`

D `(n << 1) ^ (n >> 31)`

23 A `sint32` field on the wire contains the varint byte `0x03`. What signed integer value does it represent?

A `-2`

B `-1`

C `-3`

D `3`

24 How is a custom option (such as `[(acme.audit_redacted) = true]`) implemented in Protobuf?

```
import "google/protobuf/descriptor.proto";

extend google.protobuf.FieldOptions {
  bool audit_redacted = 50001;
}
```

A An extension field defined on one of the `google.protobuf.*Options` messages

B A wire-format header prepended to each field that uses the option

C A string key-value entry stored in a `map<string, string>` on every message

D A compiler pragma that is stripped once `protoc` finishes generating code

25 How is a `google.protobuf.Duration` of 1 second and 500,000,000 nanoseconds serialized in ProtoJSON?

A `"PT1.5S"` (an ISO 8601 duration string)

B `"1.500s"` (decimal seconds ending in `s`)

C `1500` (total duration in integer milliseconds)

D `{"seconds": "1", "nanos": 500000000}`

26 Which wire types and byte orders are used for `fixed32` / `float` and `fixed64` / `double`?

A Wire type `5` (4 bytes) and wire type `1` (8 bytes), both big-endian

B Wire type `5` (4 bytes) and wire type `1` (8 bytes), both little-endian

C Wire type `4` (4 bytes) and wire type `8` (8 bytes), both big-endian

D Wire type `2` (`LEN`) with a 1-byte length prefix of `0x04` or `0x08`

27 Why must a Protobuf binary parser immediately abort if it encounters a tag with wire type `6`?

A Wire type `6` requires a schema checksum that Edition 2024 parsers do not compute

B The parser does not abort; it skips 8 bytes and stores it as an unknown field

C Wire type `6` is an explicit error sentinel emitted when serialization fails

D Wire type `6` has no defined framing, so the parser cannot know how many bytes to skip

28 Given `oneof contact { string email = 1; string phone = 2; }`, what happens if a crafted binary payload contains field `1` followed by field `2`?

A The parser rejects the payload because two `oneof` members are present

B The first member (`email`) wins and `phone` is moved to unknown fields

C Both `email` and `phone` stay set in memory until re-serialization

D The parser accepts the payload and clears `email` when `phone` is read

29 A binary payload has wire type `2` bytes `0xFF 0xFE` (invalid UTF-8) for a `string` field. What happens during parsing?

A The parser automatically switches the field's runtime type from `string` to `bytes`

B The field is moved into the message's unknown field set without failing

C Compliant parsers reject the payload because `string` fields require valid UTF-8

D The parser Base64-encodes the invalid byte sequence into an ASCII string

30 How does `google.protobuf.StringValue name = 1;` differ on the wire from an explicit-presence `string name = 1;` when set to `"abc"`?

A `StringValue` uses wire type `0` when null and wire type `2` when populated

B Both compile down to the exact same 5 bytes (`0A 03 61 62 63`) on the wire

C `StringValue` wraps `"abc"` in a sub-message (`0A 05 0A 03 61 62 63`), adding 2 extra bytes

D `StringValue` prepends an 8-byte type hash before the string payload bytes

31 Are `string` and `bytes` wire-compatible when evolving a schema?

A No, because `string` appends a trailing `0x00` null terminator on the wire

B Yes, if the payload is valid UTF-8, since both use wire type `2` identically

C Yes, even for arbitrary non-UTF-8 binary data when read as a `string`

D No, because `bytes` is Base64-encoded on the binary wire while `string` is raw

32 A field is changed from `int32 id = 1;` to `repeated int32 id = 1;`. Why can the new parser still decode old payloads that encoded `id` with wire type `0` instead of packed wire type `2`?

A It cannot unless `features.repeated_field_encoding = EXPANDED` is set on the field

B A singular `int32` field already uses wire type `2` (length-delimited) on the wire

C Parsers for `repeated` scalars must accept both packed (`2`) and unpacked (`0`) encodings

D The parser stores wire-type- `0` records as unknown fields and converts them lazily

33 What are wire types `3` (`SGROUP`) and `4` (`EGROUP`), and how does Edition 2024 express that encoding?

A Deprecated wire types that cause an immediate parse error in Edition 2024

B Streaming frame markers used by gRPC over HTTP/2 to batch messages

C 16-bit and 128-bit fixed integer wire types that were removed in `proto3`

D Start/end tags for delimited messages (`features.message_encoding = DELIMITED`)

34 How is `google.protobuf.Struct` defined in `google/protobuf/struct.proto`?

A A compiler-intrinsic scalar type that has no `.proto` message definition

B A message with `repeated google.protobuf.Any properties = 1;` keyed by field number

C A message with a single `bytes raw_json = 1;` field that stores UTF-8 JSON text

D A message with `map<string, Value> fields = 1;`, where `Value` is a `oneof` of JSON types

35 How is `google.protobuf.NullValue` defined in `google/protobuf/struct.proto`?

A As an `enum` with a single constant `NULL_VALUE = 0`

B As an empty `message NullValue {}` with zero fields

C As a `bool` field with a custom option `[is_null = true]`

D As a reserved wire type `0` varint value of `-1`

36 How do message `extensions` and `extend` blocks work in Edition 2024?

```
edition = "2024";

message Event {
  string id = 1;
  extensions 100 to 199;
}
```

A Other files can `extend Event` with fields in `100`–`199`, which encode like regular fields

B Extension fields are packed into a nested `google.protobuf.Any` field at tag `100`

C Edition 2024 only permits `extend` on `*Options` messages and rejects `extensions` here

D `extensions` allows another message to inherit `Event`'s fields `1` through `99`

37 For an explicit-presence `int32 count = 1;` field, how does ProtoJSON behave when serializing an unset field versus parsing `{"count": null}`?

A Serializing emits `"count": null`; parsing `{"count": null}` sets `count` to `0`

B Serializing omits `"count"`; parsing `{"count": null}` fails with a type error

C Serializing omits `"count"`; parsing `{"count": null}` leaves `count` unset

D Serializing emits `"count": 0`; parsing `{"count": null}` leaves `count` unset

38 A field uses a closed enum (`features.enum_type = CLOSED`). What happens when the binary parser reads an unrecognized enum number `7` for that field?

A The value `7` is discarded and replaced with `0` when re-serialized

B The field stores `7` directly and `has_*` returns `true`, matching open enums

C The binary parser aborts immediately with an invalid enum value error

D The field stays unset, and `7` is stored in the message's unknown field set

39 What are the default recursion depth limit and maximum serialized message size in the C++ and Java Protobuf implementations?

A 100 levels of nesting depth and 2 GiB (`2^31 - 1` bytes) maximum size

B 1,000 levels of nesting depth and 4 GiB (`2^32 - 1` bytes) maximum size

C 64 levels of nesting depth and no size ceiling beyond available RAM

D 32 levels of nesting depth and 64 MiB maximum serialized message size

40 When generating a `FileDescriptorSet`, where are the `.proto` source comments and line/column numbers stored?

A In `FileDescriptorProto.source_code_info`, keyed by numeric `path` slices

B They are always stripped by `protoc` and never appear in binary descriptors

C In a separate `.proto.map` sidecar file emitted beside the descriptor set

D In `FieldDescriptorProto.options` under a built-in `leading_comments` field

41 What happens when you compile this message with `protoc` or `buf`?

```
edition = "2024";

message Item {
  string user_id = 1;
  string userId = 2;
}
```

A It compiles unless `option json_strict = true;` is set on the file

B It compiles cleanly; field `2` overrides field `1` only in JSON output

C Compilation fails because both fields map to the JSON name `"userId"`

D It compiles cleanly; ProtoJSON appends `_2` to the second field's key

42 How does ProtoJSON serialize `map<int32, bool> flags` when it contains the entry `42 -> true`?

A `[{"key": 42, "value": true}]` (non-string maps emit entry arrays)

B `{"42": true}` (integer and boolean keys are formatted as JSON strings)

C `{42: true}` (integer map keys are emitted as unquoted JSON numbers)

D ProtoJSON rejects `map` fields whose key type is not `string`

43 When packing an `acme.orders.v1.OrderCreated` message into a `google.protobuf.Any`, what fields does the `Any` message hold on the wire?

A `string type_url = 1` and `google.protobuf.Struct value = 2`

B `string type_url = 1` (`"type.googleapis.com/..."`) and `bytes value = 2`

C `int32 type_id = 1` (a 32-bit type hash) and `bytes value = 2`

D `FileDescriptorProto schema = 1` and `bytes value = 2`

44 At what value does `fixed64` become smaller on the wire than `uint64`?

A Only above `2^63`, where a varint first needs 10 bytes

B When values need 57+ bits (`>= 2^56`), where a varint takes 9–10 bytes vs. 8 fixed

C `fixed64` is never smaller; both varints and `fixed64` top out at 8 bytes

D Whenever a value exceeds `2^32 - 1`, since 64-bit varints jump to 10 bytes there

45 Why should you never test two Protobuf binary payloads for logical equality by comparing or hashing their raw byte slices?

A Varint continuation bits are randomized by encoders to prevent side-channel leaks

B Two equal messages always produce identical bytes in Edition 2024; only `proto2` varied

C The wire format is non-canonical: field order, map order, and unknown fields can vary

D Serializers prepend a random 4-byte nonce to the start of every binary payload

46 What is the largest unsigned integer that fits in a 1-byte varint, and how is the next integer encoded?

A `127` (`0x7F`); `128` takes 2 bytes (`0x80 0x01`)

B `128` (`0x80`); `129` takes 2 bytes (`0x81 0x01`)

C `63` (`0x3F`); `64` takes 2 bytes (`0x40 0x01`)

D `255` (`0xFF`); `256` takes 2 bytes (`0x00 0x01`)

47 You have two serialized binary payloads `bytesA` and `bytesB` of the same message type. What happens if you concatenate them (`bytesA || bytesB`) and parse the combined byte slice?

A Parsing fails at the boundary between `bytesA` and `bytesB` with an EOF error

B It parses cleanly and produces the same result as merging `bytesB` into `bytesA`

C The second payload `bytesB` completely wipes all fields decoded from `bytesA`

D The parser stops at the end of `bytesA` and ignores all bytes in `bytesB`

48 How does ProtoJSON encode a `bytes` field, and what must a compliant ProtoJSON parser accept?

A It emits standard Base64, and parsers accept both standard and URL-safe Base64

B It emits a JSON array of unsigned byte integers (`[0..255]`), and rejects Base64

C It emits lowercase hexadecimal strings (`"deadbeef"`), and parsers also accept Base64

D It emits raw UTF-8 with `\u00XX` escapes, and rejects unpadded Base64 strings

49 A binary payload contains two separate packed wire-type-`2` records for the same `repeated int32 vals = 1;` field: `0A 02 01 02` followed later by `0A 02 03 04`. What is the decoded `vals` list?

A `[3, 4]` (the second length-delimited record overwrites the first)

B `[1, 2]` (only the first packed record for a repeated field is read)

C Parsing fails because a packed repeated field can only appear once

D `[1, 2, 3, 4]` (elements from both packed records are concatenated in order)

50 In the two-byte varint `0xAC 0x02` (which encodes `300`), which byte holds the least significant 7 bits of the number?

A The second byte (`0x02`), because varints use network big-endian byte order

B The first byte (`0xAC`), because varints order 7-bit groups least-significant first

C Neither; the first byte holds the bit length and the second holds the value

D It depends on the CPU endianness of the machine that serialized the message

51 Decode the 4-byte binary payload `0A 02 08 2A` using this schema. What is `inner.id`?

```
edition = "2024";

message Outer {
  Inner inner = 1;
}
message Inner {
  int32 id = 1;
}
```

A `42` (`0x0A` is field 1 `LEN`, `0x02` is length 2, `0x08` is field 1 varint, `0x2A` is 42)

B `2` (`0x0A` is field 1 `LEN`, and `0x02` is the value of `inner.id`)

C `21` (`0x2A` is `42`, which is ZigZag-decoded to `21` for `int32`)

D `10` (`0x0A` is `id = 10`, followed by unknown trailing bytes `02 08 2A`)

52 How does ProtoJSON represent IEEE 754 `NaN`, `+Infinity`, and `-Infinity` in `float` or `double` fields?

A As `null`, because standard JSON has no syntax for non-finite numbers

B As unquoted tokens `NaN`, `Infinity`, and `-Infinity` in the JSON output

C Serialization fails with a range error when a float is non-finite

D As the quoted strings `"NaN"`, `"Infinity"`, and `"-Infinity"`

53 How does ProtoJSON serialize a `google.protobuf.FieldMask` whose `paths` are `["user.first_name", "user.email_address"]`?

A As a nested object: `{"user": {"firstName": true, "emailAddress": true}}`

B As a JSON object: `{"paths": ["user.first_name", "user.email_address"]}`

C As a single string: `"user.firstName,user.emailAddress"`

D As a JSON array: `["user.firstName", "user.emailAddress"]`

54 How is `bool active = 1;` encoded on the wire when set to `true`?

A 2 bytes (`0x08 0x01`): tag for field `1` varint, followed by varint `1`

B 1 byte (`0x09`): field `1` with the low bit set to `1` for `true`

C 5 bytes (`0x0D 01 00 00 00`): field `1` with wire type `5` (`I32`)

D 2 bytes (`0x08 0xFF`): tag for field `1` varint, followed by all-ones `0xFF`

55 Can code generated from an Edition 2024 schema decode a binary payload written years earlier under `proto2` or `proto3`?

A Only if the Edition 2024 file sets `option legacy_wire_format = true;`

B No; Edition 2024 binaries prepend a 2-byte edition header to every payload

C Yes; `proto2`, `proto3`, and editions all share the exact same binary wire format

D No; `proto2` used wire types `3` and `4` exclusively, which editions removed

Answers: Hard level

1 B `300` is `0b100101100` (9 bits). A varint stores 7 bits per byte in little-endian group order with the high bit as a continuation flag: low 7 bits `0101100` + continuation bit `1` = `0xAC`, followed by high bits `0000010` = `0x02`. (Binary > Base-128 Varints)

2 C A tag packs `(field_number << 3) | wire_type`. `0x08` is `0b00001000`: shifting right by 3 gives field `1`, and the low 3 bits are `000` (wire type `0`, varint). (Binary > The Field Tag)

3 C Negative `int32` values are sign-extended to 64 bits before varint encoding, so `-1` always costs 10 bytes. `sint32` applies ZigZag encoding first, mapping `-1` to `1` and `-2` to `3`, which fit in a single byte. (Binary > ZigZag)

4 C Embedded messages, `string`, `bytes`, and packed repeated fields all use wire type `2` (`LEN`): a varint byte length followed by that many payload bytes. (Binary > Wire Types)

5 B Edition 2024 defaults singular scalars to explicit presence (`features.field_presence = EXPLICIT`). When `count` is explicitly set to `0`, its tag and value (`0x08 0x00`) are written to the wire; when unset, nothing is written. (Binary > Presence)

6 A Repeated scalar numeric fields are packed by default: a single wire-type-`2` tag (`(1 << 3) | 2 = 0x0A`), the byte length of the packed payload (`0x03`), and the concatenated varints (`01 02 03`). (Basics > Repeated Fields)

7 C The wire type tells Service B how long field `5` is, so it can keep the field's data in the message's unknown field set without knowing its schema, and include it when it re-serializes. The data is preserved, but the bytes are not guaranteed to be identical: C++, for example, stores an unknown varint as a decoded integer and re-encodes it. (Advanced > Unknown Fields)

8 A Edition 2024 enums are `OPEN` by default: unknown numeric values stay in the field and are returned by the getter. Only `CLOSED` enums (`features.enum_type = CLOSED`) divert unrecognized values to the unknown field set. (Basics > Enums)

9 C For singular scalar fields, the parser processes tags sequentially and each occurrence overwrites the previous value, so the last occurrence (`9`) wins. (Binary > Parse Loop)

10 C Field numbers `19000` through `19999` (`FieldDescriptorProto::kFirstReservedNumber` to `kLastReservedNumber`) are reserved for internal Protobuf use. Valid user field numbers are `1` to `536,870,911` outside that range. (Basics > Fields)

11 B The tag is `(field_number << 3) | wire_type` = `(2 << 3) | 2` = `16 | 2` = `18`, which is `0x12` in hexadecimal. (Binary > The Field Tag)

12 A `(16 << 3) | 0` = `128` (`0b10000000`). Because a single varint byte holds only 7 bits (`0`–`127`), `128` requires two varint bytes: low 7 bits `0000000` with continuation bit `1` (`0x80`), followed by `0x01`. (Binary > The Field Tag)

13 A So that an `int32` field can be widened to `int64` without breaking wire compatibility, negative `int32` values are sign-extended to 64 bits, setting bit 63 and requiring `ceil(64 / 7) = 10` varint bytes. (Binary > ZigZag)

14 B `int32` and `sint32` both use wire type `0`, so the parse succeeds, but ZigZag maps `0 -> 0`, `1 -> -1`, `2 -> 1`, `3 -> -2`. A raw varint `2` is silently decoded as `1`. (Binary > ZigZag)

15 D In ProtoJSON, `Any` expands ordinary messages inline into the same JSON object as `"@type"`. When the packed type is a well-known type with a special JSON mapping (like `Timestamp`, `Duration`, `FieldMask`, or `Struct`), ProtoJSON wraps that representation in `{"@type": "...", "value": ...}`. `Struct` gets the wrapper too, even though it maps to a JSON object. (Advanced > The Any Type)

16 D Maps are wire-equivalent to `repeated MapEntry scores = 3;` where `MapEntry` has `key = 1` and `value = 2`. There is no separate map wire type. (Binary > Map Encoding)

17 A Because each map entry is parsed as an embedded message with `key = 1` and `value = 2`, a missing `value` defaults to the type's default (`0`), and inserting a key that already exists overwrites the earlier entry. (Binary > Map Encoding)

18 A Unlike singular scalars (which overwrite), multiple occurrences of a singular embedded message are merged by the parser: scalar fields inside take the later value and repeated fields inside are concatenated. (Binary > Parse Loop)

19 A In the wire format, the tag is a 32-bit integer (`(field_number << 3) | wire_type`). Shifting out the 3 wire-type bits leaves 29 bits (`1` through `2^29 - 1 = 536,870,911`). (Binary > The Field Tag)

20 B Implicit presence tracks no separate 'is set' bit: at serialization time, if the current value equals the default (`0`), nothing is emitted on the wire. (Binary > Presence)

21 C The Protobuf plugin protocol is pure Protobuf over stdio: the compiler sends a binary `CodeGeneratorRequest` containing `FileDescriptorProto`s on `stdin`, and the plugin writes a binary `CodeGeneratorResponse` to `stdout`. (Descriptors)

22 D `n << 1` places the magnitude in bits `1..31`, and arithmetic `n >> 31` is all `0`s for non-negative `n` and all `1`s for negative `n`. XORing them flips the bits of negative numbers and puts the sign bit in bit `0`. (Binary > ZigZag)

23 A ZigZag maps `0 -> 0`, `-1 -> 1`, `1 -> 2`, `-2 -> 3`, `2 -> 4`. Decoding unsigned `u` uses `(u >>> 1) ^ -(u & 1)`, so `3` decodes to `-2`. (Binary > ZigZag)

24 A Custom options extend the descriptor option messages (`FileOptions`, `MessageOptions`, `FieldOptions`, etc.) from `google/protobuf/descriptor.proto`. Their values are stored as regular fields on the compiled descriptor. (Extending Protobuf)

25 B Serializers write `Duration` as a string of decimal seconds with 0, 3, 6, or 9 fractional digits followed by `"s"`, so the output is `"1.500s"`. Parsers accept any number of fractional digits, so `"1.5s"` is also valid input. (Types)

26 B `fixed32`, `sfixed32`, and `float` use wire type `5` (`I32`, 4 little-endian bytes). `fixed64`, `sfixed64`, and `double` use wire type `1` (`I64`, 8 little-endian bytes). (Binary > Wire Types)

27 D Skipping an unknown field relies on knowing its byte length from the wire type alone (`0` varint, `1` 8 bytes, `2` length-prefixed, `3`/`4` group pair, `5` 4 bytes). Because wire types `6` and `7` are undefined, the parser cannot find the next tag boundary. (Binary > Wire Types)

28 D On the wire, `oneof` members are indistinguishable from ordinary fields. When a decoder encounters a second member of the same `oneof` later in the byte stream, it clears the earlier member, so the last member on the wire wins. (Basics > Oneof)

29 C Protobuf requires `string` fields to contain valid UTF-8 (enforced strictly in `proto3` and Edition 2024, where `features.utf8_validation` defaults to `VERIFY`). Payloads with non-UTF-8 bytes fail parsing. (Types)

30 C `google.protobuf.StringValue` is a message `{ string value = 1; }`, so on the wire it has an outer tag+length (`0A 05`) wrapping the inner `value` field's tag+length (`0A 03`). In Edition 2024, plain `string name = 1;` already has explicit presence without that overhead. (Advanced > Field Presence)

31 B Both `string` and `bytes` encode as wire type `2` (a varint byte length followed by raw bytes). They are wire-compatible as long as any data read by a `string` parser is valid UTF-8. (Advanced > Schema Evolution)

32 C Even when a `repeated` scalar field defaults to packed encoding (wire type `2`), the Protobuf spec requires parsers to accept unpacked elements (wire type `0` / `1` / `5`) as well, so new readers decode old payloads. The reverse does not hold: an old singular reader sees the new packed record as wire type `2` , which does not match `int32` , so it keeps the record as an unknown field and `id` reads as unset. (Advanced > Schema Evolution)

33 D Instead of prefixing a nested message with its byte length (wire type `2`), delimited encoding wraps the fields between a `StartGroup` (wire type `3`) and `EndGroup` (wire type `4`) tag. In Edition 2024, you declare a normal message field with `features.message_encoding = DELIMITED` . (Binary > Wire Types)

34 D `Struct` is defined in standard Protobuf as `map<string, Value> fields = 1;` , where `Value` has a `oneof kind` covering `NullValue` , `double` , `string` , `bool` , `Struct` , and `ListValue` . (Advanced > The Value Type)

35 A `NullValue` is an enum with one constant (`NULL_VALUE = 0`), used inside `google.protobuf.Value` 's `oneof` to represent an explicit JSON `null` . (Advanced > The Value Type)

36 A While `proto3` restricted `extend` to custom options, Edition 2024 supports user-message extensions again. An extension field serializes on the wire using its assigned field number (`100` – `199`) identically to a regular field. (Extending Protobuf)

37 C In ProtoJSON, unset fields are omitted when serializing, and `null` on input leaves a field unset. The exception is `google.protobuf.NullValue` , where `null` is a present value, which lets `google.protobuf.Value` and `Struct` carry a JSON `null` . (Types)

38 D For a `CLOSED` enum, an unrecognized value cannot be stored in the typed field; instead, the parser routes the tag and varint to the message's unknown field set so it is preserved on re-serialization. (Advanced > Schema Evolution)

39 A C++ and Java cap recursion depth at 100 to prevent stack-overflow DoS from deeply nested payloads; Go's default is much higher, at 10,000. The 2 GiB ceiling applies to every implementation: a serialized message must be smaller than 2 GiB. (Advanced > Size Limits)

40 A When compiled with `--include_source_info` (or `buf build`), `FileDescriptorProto.source_code_info` holds `Location` records whose `path` arrays (sequences of descriptor field numbers and list indices) map AST nodes to their line/column spans and comments. (Descriptors)

41 C The compiler checks that all JSON field names within a message are distinct. Because `user_id` and `userId` both derive the camelCase JSON name `"userId"` , the compiler rejects the schema. (Advanced > Schema Evolution)

42 B Because JSON object keys must be strings, ProtoJSON formats integer and `bool` map keys as decimal or `"true"` / `"false"` strings when serializing, and parses them back to their scalar type on input. (Types)

43 B `google.protobuf.Any` has only two fields: `string type_url = 1;` (defaulting to the `type.googleapis.com/<full.type.Name>` prefix) and `bytes value = 2;` (the standard binary serialization of the packed message). (Advanced > The Any Type)

44 B Each varint byte carries 7 payload bits, so an 8-byte varint holds up to `8 * 7 = 56` bits. Values needing 57–64 bits (such as random 64-bit IDs or hashes) take 9 or 10 bytes as a varint, making 8-byte `fixed64` smaller. `sint64` crosses over sooner, at a magnitude of about `2^55`, because ZigZag doubles each value before encoding it. (Binary > Base-128 Varints)

45 C Multiple distinct byte sequences can decode to the exact same logical message (different map iteration order, concatenated payloads, non-minimal varints, or preserved unknown fields). (Binary > Parse Loop)

46 A Because the highest bit of each varint byte is reserved for the continuation bit, a single byte carries 7 value bits (`0` through `127`, or `0x00` – `0x7F`). `128` (`0b10000000`) splits into `0x80 0x01`. (Binary > Base-128 Varints)

47 B A top-level Protobuf message has no outer length prefix or end-of-message marker; it is simply a stream of tagged fields. Concatenating two serialized messages is equivalent to parsing the first and merging the second. (Binary > Parse Loop)

48 A ProtoJSON serializes `bytes` fields as standard Base64 strings, and requires parsers to accept both standard Base64 and URL-safe Base64 (`-` and `_`), with or without padding (`=`). (Types)

49 D When a message is merged or concatenated, a packed repeated field can appear in multiple wire-type-`2` chunks. The parser unpacks each chunk and appends its elements to the existing list. (Basics > Repeated Fields)

50 B Protobuf varints are little-endian at the 7-bit group level: the first byte on the wire (`0xAC` → low 7 bits `0b0101100` = `44`) holds bits `0..6`, and the second byte (`0x02` → `2 << 7 = 256`) holds bits `7..13`. (Binary > Base-128 Varints)

51 A `0x0A` = `(1 << 3) | 2` (field `1`, length-delimited), `0x02` = 2-byte sub-message payload. Inside `Inner`, `0x08` = `(1 << 3) | 0` (field `1`, varint), and `0x2A` = `42`. (Binary > Binary Explorer)

52 D RFC 8259 JSON forbids unquoted `NaN` and `Infinity` literals, so ProtoJSON serializes them as the exact strings `"NaN"`, `"Infinity"`, and `"-Infinity"`. (Types)

53 C ProtoJSON converts each snake_case path segment to `lowerCamelCase` and joins all paths into a single comma-separated string. (Types)

54 A `bool` uses wire type `0` (varint): `false` encodes as `0x00` and `true` encodes as `0x01`, preceded by the tag byte `0x08` (`(1 << 3) | 0`). (Binary > Wire Types)

55 C The binary wire format has remained unchanged across `proto2`, `proto3`, and editions. Editions unify source syntax and feature defaults without changing how fields are encoded on the wire. (Advanced > Editions)

APPENDIX C

On the web

Everything in this book is on protobuf.com, and some of it only works there. These are the parts of the site that need a browser.

Playground

https://protobuf.com/playground/

Edit a schema and see a message that fits it, as JSON and as annotated bytes, with examples for every feature the guide covers.

Binary Explorer

https://protobuf.com/binary/explorer/

Encode a message against a schema and inspect every tag, length prefix, and payload in the result.

Descriptor explorer

https://protobuf.com/descriptors/

Edit a schema and watch the FileDescriptorSet the compiler produces for it.

Validation lab

https://protobuf.com/validation/

Run protovalidate rules against a message, and change the rules themselves.

Size comparison

https://protobuf.com/efficiency/

Weigh a message's JSON against its binary encoding, raw and compressed.

Topic labs

https://protobuf.com/basics/messages/

Most topic pages let you edit the topic's example message and re-encode it with the real protobuf-es runtime.

Practice

https://protobuf.com/practice/

Short schema exercises, with the compiler checking your schema as you type.

Quiz

https://protobuf.com/quiz/

Ten questions at a time from the bank in Appendix B, with your best score saved in the browser.

Ecosystem directory

https://protobuf.com/ecosystem/

Tools, plugins, services, and libraries from across the community, with search, filters, and activity data refreshed daily.