**ProtocolBuffers**

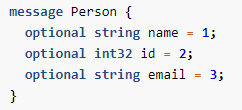
**https://protobuf.dev/**

* language-neutral, platform-neutral extensible mechanism for serializing structured data
* It’s like JSON, except it’s smaller and faster, and it generates native language bindings
* combination of the definition language (created in .proto files), the code that the proto compiler generates to interface with data, language-specific runtime libraries, the serialization format for data that is written to a file (or sent across a network connection), and the serialized data.

**What Problems do Protocol Buffers Solve?**

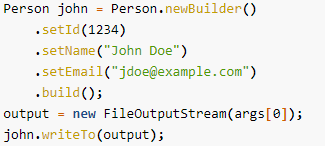
1. serialization format for packets of typed, structured data that are up to a few megabytes in size - suitable for both ephemeral network traffic and long-term data storage
2. Can be extended with new information without invalidating existing data or requiring code to be updated.

Eg



The proto compiler is invoked at build time on .proto files to generate code in various programming languages to manipulate the corresponding protocol buffer.

Each generated class contains simple accessors for each field and methods to serialize and parse the whole structure to and from raw bytes. The following shows you an example that uses those generated methods:



1. Where backward compatibility crucial. Protocol buffers allow for the seamless support of changes, including the addition of new fields and the deletion of existing fields, to any protocol buffer without breaking existing services.

**What are the Benefits of Using Protocol Buffers?**

* Ideal for any situation in which you need to serialize structured, record-like, typed data in a language-neutral, platform-neutral, extensible manner
* most often used for defining communications protocols (together with gRPC) and for data storage.
* Compact data storage
* Fast parsing
* Availability in many programming languages
* Optimized functionality through automatically-generated classes
* Cross-project support

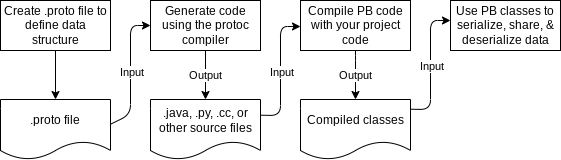
**Updating Proto Definitions Without Updating Code**

It’s standard for software products to be backward compatible, but it is less common for them to be forward compatible.

**When are Protocol Buffers not a Good Fit?**

1. Protocol buffers tend to assume that entire messages can be loaded into memory at once and are not larger than an object graph. For data that exceeds a few megabytes, consider a different solution; when working with larger data, you may effectively end up with several copies of the data due to serialized copies, which can cause surprising spikes in memory usage.
2. When protocol buffers are serialized, the same data can have many different binary serializations. You cannot compare two messages for equality without fully parsing them.
3. Messages are not compressed. While messages can be zipped or gzipped like any other file, special-purpose compression algorithms like the ones used by JPEG and PNG will produce much smaller files for data of the appropriate type.
4. Protocol buffer messages are less than maximally efficient in both size and speed for many scientific and engineering uses that involve large, multi-dimensional arrays of floating point numbers. For these applications, FITS and similar formats have less overhead.
5. Protocol buffers are not well supported in non-object-oriented languages popular in scientific computing, such as Fortran and IDL.
6. Protocol buffer messages don’t inherently self-describe their data, but they have a fully reflective schema that you can use to implement self-description. That is, you cannot fully interpret one without access to its corresponding .proto file.
7. Protocol buffers are not a formal standard of any organization. This makes them unsuitable for use in environments with legal or other requirements to build on top of standards.

**How do Protocol Buffers Work?**

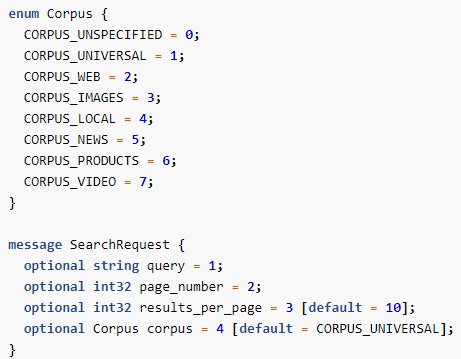


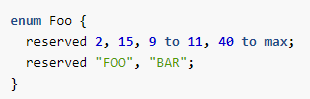
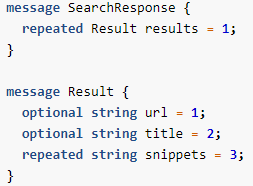
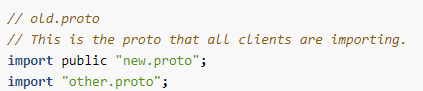
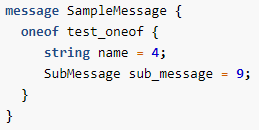
**Protocol Buffers Definition Syntax**

* A field is either optional or repeated. Or leave it set to the default.
* Specify the datatype.
  + message: can nest parts of definition
  + enum: a set of values to choose from
  + oneof: many optional fields and atmost one filed will be set at the same time
  + map: key-value pairs

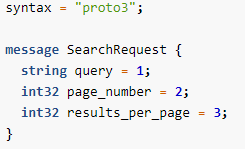
**Proto 2**

* + Scalar value type
    - Double, float, int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64, bool, string, bytes
  + Optional Fields and Default Values
  + Enumerations
  + Reserved Values



* + Reserved Values
  + 
  + Using Other Message Types
  + 
  + 
  + Nested Types
  + 
  + Groups
    - Deprecated
  + Extensions
    - a field defined outside of its container message; usually in a .proto file separate from the container message’s .proto file
    - 
  + Using One of
    - 
  + Maps
    - 
* *Skiped further sections*

**Proto 3**

****

* Field numbers from 1 to 536,870,911, must be unique among all fields
* 19000-19999 reserved for protocol buffers implementation
* The number cannot be changed once the message type is in use
* Well-formed Messages – bytes serialized or deserialised
* protoc parser validates that a given proto definition file is parseable
* Deleting Fields
  + Must reserve the field number (reservefields)
  + Delete the field definition from the message
  + Also reserve the field names – for json text format encoding to continue to parse
* For Kotlin, in addition to the Java generated code, the compiler generates a .kt file for each message type with an improved Kotlin API. This includes a DSL that simplifies creating message instances, a nullable field accessor, and a copy function.
* Default values
  + For strings, the default value is the empty string.
  + For bytes, the default value is empty bytes.
  + For bools, the default value is false.
  + For numeric types, the default value is zero.
  + For enums, the default value is the first defined enum value, which must be 0.
  + For message fields, the field is not set. Its exact value is language-dependent. See the generated code guide for details

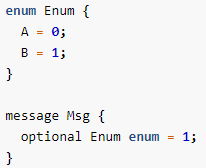
Skipped

to add Style Guide <https://protobuf.dev/programming-guides/style/>.

<https://protobuf.dev/programming-guides/enum/>

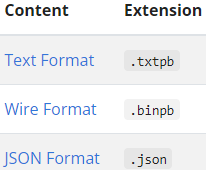
**Enums**

Two distinct flavors – open and closed

* Closed Enum
* 

**Design patterns**

1. Common Filename Suffixes



1. Streaming Multiple Messages

to write multiple messages to a single file or stream, it is up to you to keep track of where one message ends and the next begins. The Protocol Buffer wire format is not self-delimiting, so protocol buffer parsers cannot determine where a message ends on their own. The easiest way to solve this problem is to write the size of each message before you write the message itself. When you read the messages back in, you read the size, then read the bytes into a separate buffer, then parse from that buffer.

1. Large Data Sets

Not designed to handle large messages

Great for handling individual messages within a large data set

1. Self-describing Messages

Protocol Buffers do not contain descriptions of their own types. Thus, given only a raw message without the corresponding .proto file defining its type, it is difficult to extract any useful data.

However, the contents of a .proto file can itself be represented using protocol buffers. The file src/google/protobuf/descriptor.proto in the source code package defines the message types involved. protoc can output a FileDescriptorSet—which represents a set of .proto files—using the --descriptor\_set\_out option. With this, you can define a self-describing protocol message like so:



*Many skipped areas*

**Proto Serialization Is Not Canonical**

* Cannot with usecases Use cases include:
  + using a serialized proto as a key in a hash table
  + taking a fingerprint or checksum of a serialized proto
  + comparing serialized payloads as a way of checking message equality
  + **Deterministic is not Canonical**
    - The protobuf schema changes in anyway
    - The application being build changes in any way
    - The binary is built with different flags (eg opt vs debug)
    - The protobuf library is updated

**Proto Best Practices**

* **Don’t** reuse a tag number
* **Do** reserve tag numbers for deleted fields
* **Do** reserve numbers for deleted enum values
* **Don’t** change the type of a field
* **Don’t** add a required field
* **Don’t** make a message with lots of fields
* **Do** include an unspecified value in an enum
* **Do** use well known types and common types

Using the following common, shared types is strongly encouraged. E.g., do not use int32 timestamp\_seconds\_since\_epoch or int64 timeout\_millis in your code when a perfectly suitable common type already exists!

duration is a signed, fixed-length span of time (for example, 42s).

timestamp is a point in time independent of any time zone or calendar (for example, 2017-01-15T01:30:15.01Z).

interval is a time interval independent of time zone or calendar (for example, 2017-01-15T01:30:15.01Z - 2017-01-16T02:30:15.01Z).

date is a whole calendar date (for example, 2005-09-19).

month is a month of year (for example, April).

dayofweek is a day of week (for example, Monday).

timeofday is a time of day (for example, 10:42:23).

field\_mask is a set of symbolic field paths (for example, f.b.d).

postal\_address is a postal address (for example, 1600 Amphitheatre Parkway Mountain View, CA 94043 USA).

money is an amount of money with its currency type (for example, 42 USD).

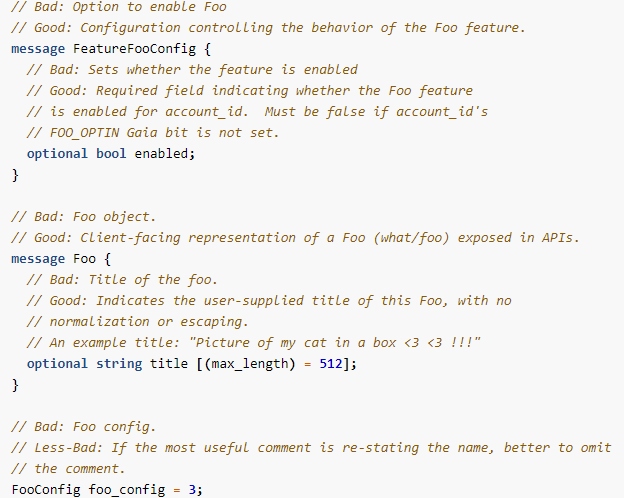
latlng is a latitude/longitude pair (for example, 37.386051 latitude and -122.083855 longitude).

color is a color in the RGBA color space

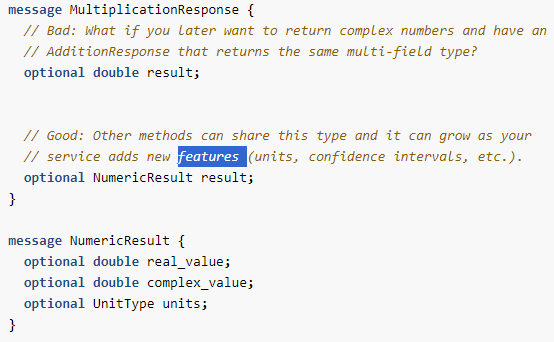
* Don’t Change the Default Value of a Field
* Don’t Go from Repeated to Scalar
* Don’t use Text Format Messages for Interchange
* Never Rely on Serialization Stability Across Builds
* Don’t Generate Java Protos in the Same Java Package as Other Code
* Avoid Using Language Keywords for Field Names

**API Best Practices**

1. Precisely, Concisely Document Most Fields and Messages



1. Use Different Messages for Wire and Storage
2. For Mutations, Support Partial Updates or Append-Only Updates, Not Full Replaces
3. Don’t Include Primitive Types in a Top-level Request or Response Proto



1. Never Use Booleans for Something That Has Two States Now, but Might Have More Later
2. Rarely Use an Integer Field for an ID
3. Don’t Encode Data in a String That You Expect a Client to Construct or Parse
4. Returning HTML in a Front-End Proto
5. Encode Opaque Data in Strings by Web-Safe Encoding Binary Proto Serialization
6. Don’t Include Fields that Your Clients Can’t Possibly Have a Use for
7. Rarely Define a Pagination API Without a Continuation Token
8. Include a Field Read Mask in Read Requests
9. Include a Version Field to Allow for Consistent Reads
10. Use Consistent Request Options for RPCs that Return the Same Data Type
11. Batch/multi-phase Requests
12. Create Methods that Return or Manipulate Small Bits of Data and Expect Clients to Compose UIs from Batching Multiple Such Requests
13. Make a One-off RPC when the Alternative is Serial Round-trips on Mobile or Web
14. Make Repeated Fields Messages, Not Scalars or Enums
15. Use Proto Maps
16. Prefer Idempotency - A simple way to avoid duplicate writes is to allow clients to specify a client-created request ID that your server dedupes on (for example, hash of content or UUID)
17. Be Mindful of Your Service Name, and Make it Globally Unique
18. Ensure Every RPC Specifies and Enforces a (Permissive) Deadline

RPC clients should still set a deadline on outgoing RPCs and will typically do so by default when they use standard frameworks. A deadline may and typically will be overwritten by a shorter deadline attached to a request.

Setting the deadline option clearly communicates the RPC deadline to your clients, and is respected and enforced by standard frameworks:



1. Bound Request and Response Sizes
2. Propagate Status Codes Carefully
3. Create Unique Protos per Method

Protobuf Editions

Replace proto2 and proto3 designations

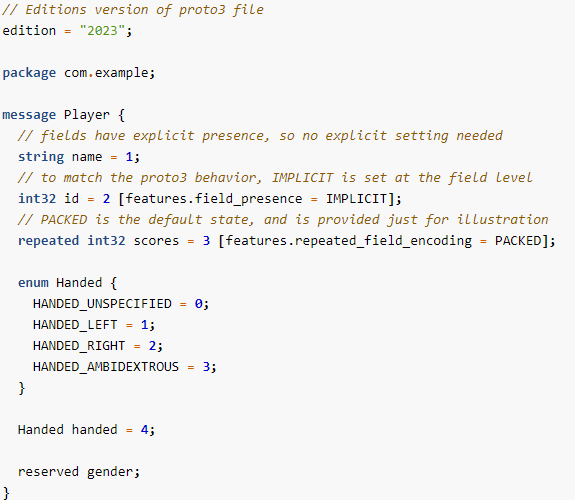
Instead of the hardcoded behaviors that older versions have had, editions represent a collection of features with a default value (behavior) per feature. Features are options on a file, message, field, enum, and so on, that specify the behavior of protoc, the code generators, and protobuf runtimes.

Instead of adding syntax = "proto2" or syntax = "proto3" at the top of proto definition files, you use an edition number, such as edition = "2023", to specify the default behaviors your file will have. Editions enable the language to evolve incrementally over time.

Proto3 syntax



Editions syntax



Prototiller

A migration guide and migration tooling that ease the migration to and between editions. The tool, called Prototiller, will enable you to:

* convert proto2 and proto3 definition files to the new editions syntax, at scale
* migrate files from one edition to another
* manipulate proto files in other ways

Backward Compatibility

We are building Protobuf Editions to be as minimally disruptive as possible. For example, you can import proto2 and proto3 definitions into editions-based definition files, and vice versa:

