Intro

Definition

- A method of serializing structured data
- Useful in developing programs to communicate with each other over a network or for storing data

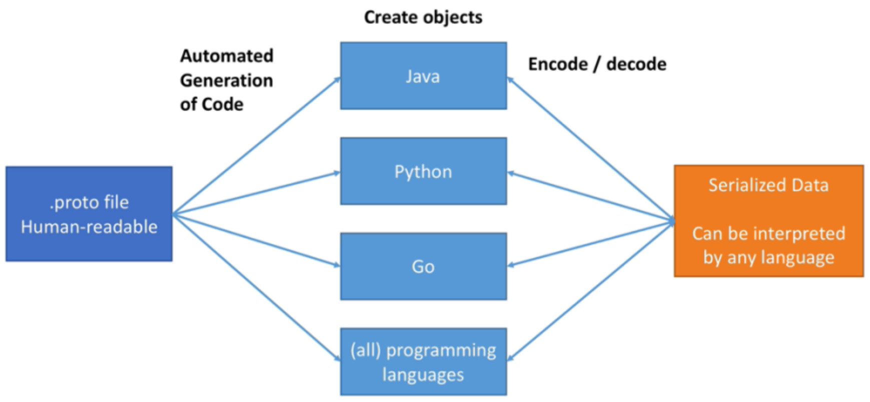
Advantages

- Data is fully typed
- Data is compressed automatically
- Schema is needed to generate code and read the data
- Documentation can be embedded in the schema
- Data can be read across any language
- Schema can evolve over time in a safe manner
- Smaller and faster than XML
- Code is generated for you automatically

Disadvantages

- Protobuf support for some languages might be lacking
- Can't "open" with a text editor

How is Protocol buffer used

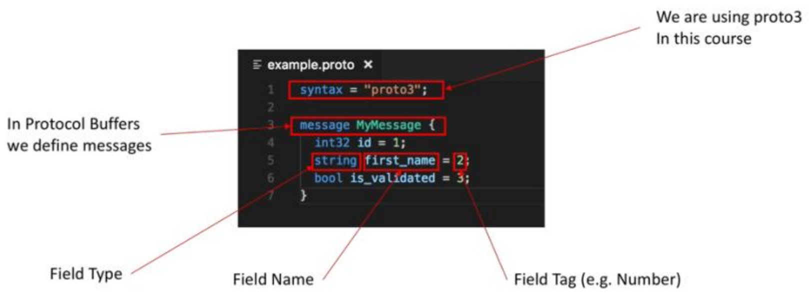

Create objects
Automated Generation of Code
Encode / decode
.proto file
Human-readable
Java
Python
Go
(all) programming languages
Serialized Data
Can be interpreted by any language

Proto2 vs Proto3

- Mid 2016, Google release the 3rd iteration of Protocol Buffers, named proto3

Protocol Buffers Basics

First Message



```
syntax = "proto3";

message MyMessage {
  int32 id = 1;
  string first_name = 2;
  bool is_validated = 3;
}
```

Scalar Types Number

- Integer: int32

- Foating: float, double
- Boolearn: Bool
- String
- Bytes: bytes
    - Small image

Tags

- Smallest tag: 1
- Largest tag: 2**29
- Cannot use 19000-19999
- Tags numbered from 1 to 15 use 1 byte in space
    - Use them for frequently populated fields
- Tags numbered from 16 to 2047 use 2 bytes

Repeated Fields

- Make a list or an array

Comments

- //
- /* */

Default Values for field

- All fields, if not specified or unknown, will take a default value

Enums

- The first value of an enum is the default value
- Enum must start by the tag 0

Defining multiple Messages in the same .proto file

Nesting Messages

- Possible to define types within types

Importing Types

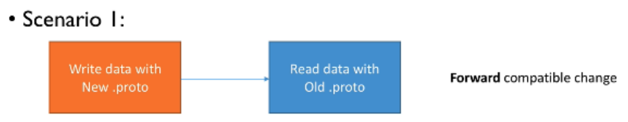
- Can have different types in different .proto files

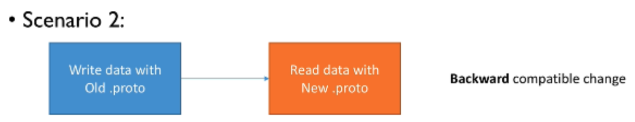
Packages

- Define the packages in which your protocol buffer message live
    - When code gets compiled, it will be placed at the package you indicated

Data Evolution

The need for updating the protocols


• Scenario 1:
Write data with New .proto
Read data with Old .proto
Forward compatible change

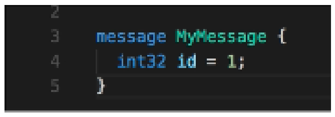

• Scenario 2:
Write data with Old .proto
Read data with New .proto
Backward compatible change

Updating Protocol Rules

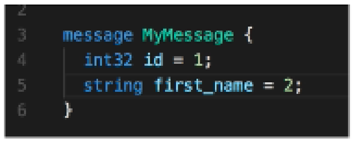
- Don't change the numeric tags for any existing fields .
- You can add new fields, and old code will just ignore them.
- If the oldnew code reads unknows data, the default will take place

- Fields can be removed, as long as the. Tag number is not used again in your updated message type.
    - If renaming the field, adding the prefix “OBSOLETE_”, or make the tag reserved.

Adding fields



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



```
message MyMessage {
  int32 id = 1;
  string first_name = 2;
}
```

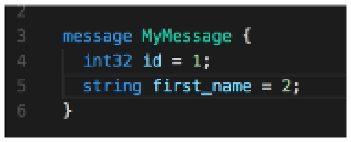
- If that field is sent to old code, the old code will not know what that tag number corresponds to and the field will be ignored (or dropped)
- Oppositely, if we read old data with the new code, the new field will not be found and the default value will be assumed (empty string)

- Default values should always be interpreted with care

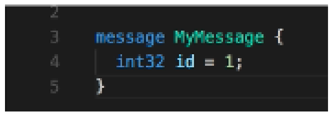
Renaming Fields

- Only the tag number is important for Protobuf

Removing Fields

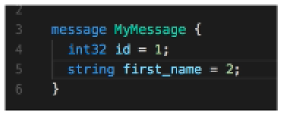


```
message MyMessage {
  int32 id = 1;
  string first_name = 2;
}
```

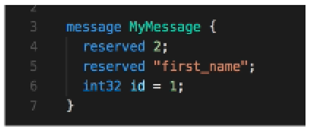


```
message MyMessage {
  int32 id = 1;
}
```

- If old code doesn’t find the field anymore, the default value will be used
- If we read old data with the new code, the deleted field will just be dropped
- Default values should always be interpreted with care
- When removing a field, you should ALWAYS reserve the tag and the name



```
message MyMessage {
  int32 id = 1;
  string first_name = 2;
}
```



```
message MyMessage {
  reserved 2;
  reserved "first_name";
  int32 id = 1;
}
```

    - This prevents the tag to be re-used and this prevents the name to be re-used
    - Necessary to prevent conflicts in the codebase
- Alternative: rename it to OBSOLETE_field_name

Reserved Keywords

- Can’t mix TAGS AND FIELDS NAMES in the same reserved statement
- Reserve TAGS to prevent new fields from re-using tags
- Don’t ever remove any reserved tags

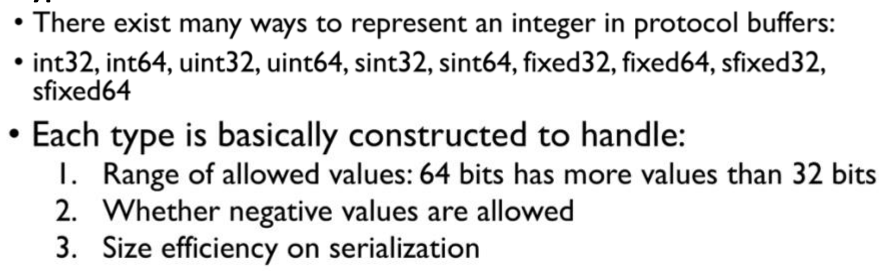
Defaults

- A field will always have a non-null values
- You cannot differentiate from a missing field or if a value equal to the default was set.
- Solution
    - Make sure the default value doesn’t have meaning for your business
    - Deal with default values in your code if needed
        - Use if statements

Evolving Enumerations

- Make the first value “UNKNOWN = 0”

Integer Types

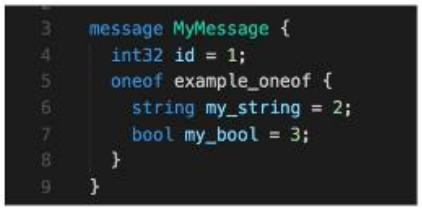


- There exist many ways to represent an integer in protocol buffers:
- int32, int64, uint32, uint64, sint32, sint64, fixed32, fixed64, sfixed32, sfixed64
- Each type is basically constructed to handle:
  1. Range of allowed values: 64 bits has more values than 32 bits
  2. Whether negative values are allowed
  3. Size efficiency on serialization

Advanced Types

one of

- Only one field can have a value

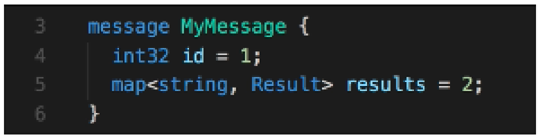


```
message MyMessage {
  int32 id = 1;
  oneof example_oneof {
    string my_string = 2;
    bool my_bool = 3;
  }
}
```

- Can't be repeated
- Evolving schemas using one of is complicated
- On read, all fields will be null except the last one that was set at write

Maps

- Maps scalars to values of any type

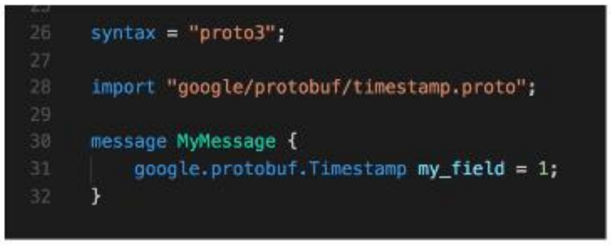


```
message MyMessage {
  int32 id = 1;
  map<string, Result> results = 2;
}
```

- Cannot be repeated
- No ordering for map

Well Know Types

- Ex: Timestamps
  - o Have to sue the import statement

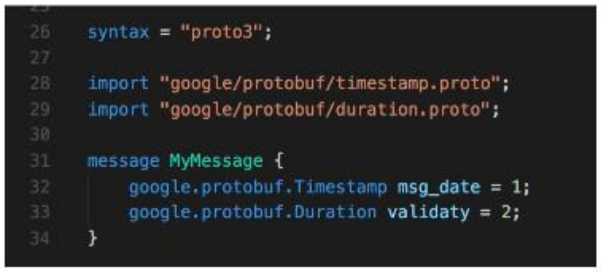


```
syntax = "proto3";

import "google/protobuf/timestamp.proto";

message MyMessage {
    google.protobuf.Timestamp my_field = 1;
}
```

- Duration
  - o Represents the time span between two timestamps



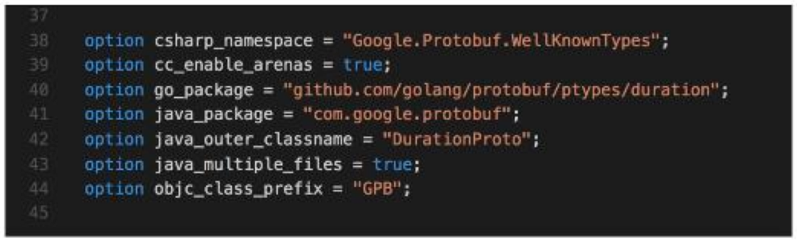
```
syntax = "proto3";

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

message MyMessage {
    google.protobuf.Timestamp msg_date = 1;
    google.protobuf.Duration validaty = 2;
}
```

Options

- Allow to alter the behavior of the protoc compiler when generating code for specific languages



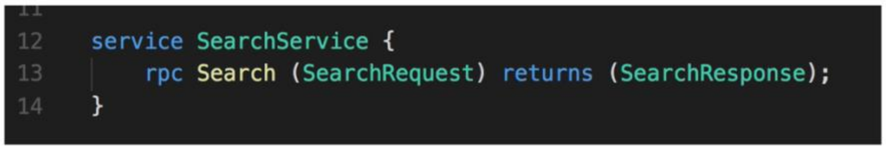
```
option csharp_namespace = "Google.Protobuf.WellKnownTypes";
option cc_enable_arenas = true;
option go_package = "github.com/golang/protobuf/ptypes/duration";
option java_package = "com.google.protobuf";
option java_outer_classname = "DurationProto";
option java_multiple_files = true;
option objc_class_prefix = "GPB";
```

Naming Convention From the doc

- https://developers.google.com/protocol-buffers/docs/style

Protocol Buffer Services

- A set of endpoints your application can be accessible from



```
service SearchService {
    rpc Search (SearchRequest) returns (SearchResponse);
}
```

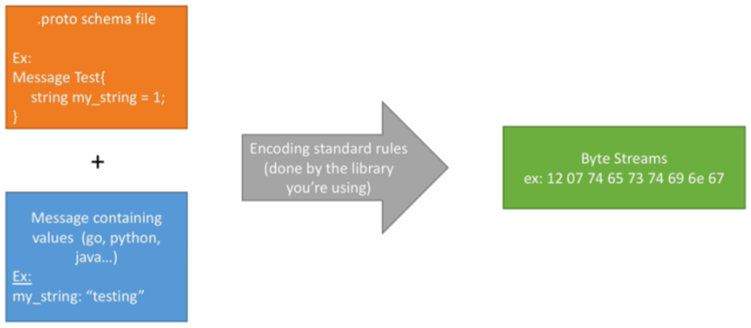
- Services need to be interpreted by a framework to generate associated code

Protocol Buffers Internals

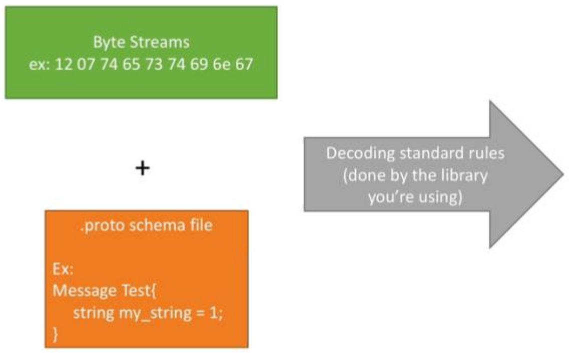
Protocol Buffers Encoding

- The magic is protocol buffers is to have the same serialization and deserialization for al the languages
- Serialization means transforming an object into bytes and deserialization means taking bytes and getting an object out of it

High level understanding encoding


.proto schema file
Ex:
Message Test{
string my_string = 1;
}
+
Message containing values (go, python, java…)
Ex:
my_string: "testing"
Encoding standard rules (done by the library you're using)
Byte Streams
ex: 12 07 74 65 73 74 69 6e 67

High level understanding decoding


Byte Streams
ex: 12 07 74 65 73 74 69 6e 67
+
.proto schema file
Ex:
Message Test{
string my_string = 1;
}
Decoding standard rules (done by the library you're using)
Message containing values (go, python, java...)
Ex:
my_string: "testing"

Decoding Rules for VarInts

- A number of ariable length when encoded
-