Protobuf Serialization Mechnism and Alternatives

Azat Satklycov

Agenda

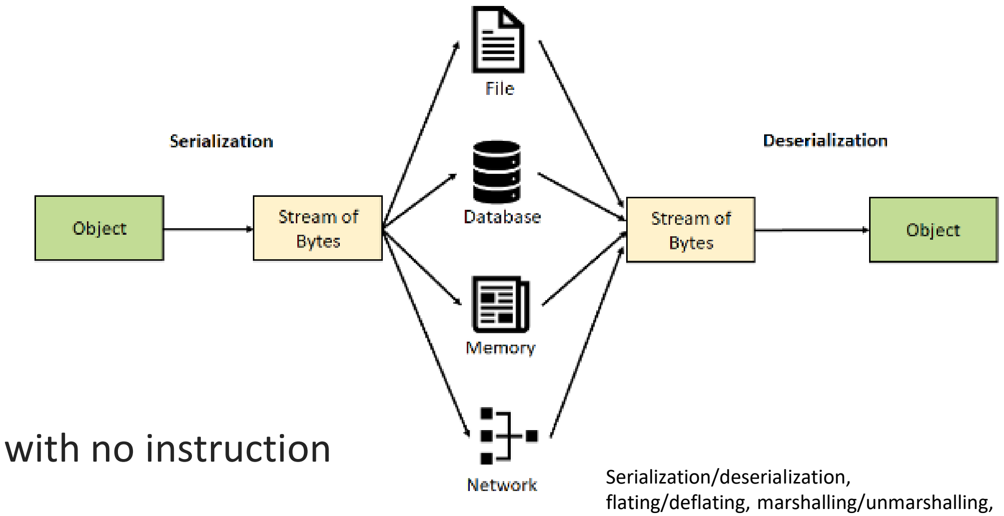
- Java Serialization Mechanism
- Alternative Serialization Mechanisms
- FlatBuffers
- Cap’n Proto
- JSON and CSV, XML, YAML etc
- Message Pack
- Comparison Matrix
- Protocol Buffers - Best Practices (Don’t & Dos)

Java Serialization Mechanism

APIs: java.io.**Serializable|Externalizable** interfaces, ObjectOutputstream, ObjectInputStream, NotSerializableException, and in Java 9 ObjectInputFilter. **Usages:** persist objects, deep cloning, communication (RMI, ApacheHadoop, Spark, etc.), long-term data storage …

Caveats / Performance Overhead

- Only object marked Serializable can be persisted
- Resource leak if you forget release I/O streams
- Process is recursive – whole graph of objects included (heavy object)
- If superclass is not serializable no parent constructor called
- SerialVersionUID – InvalidClassException thrown if not same on both objects
- Not work well if you share data with apps written in C++, Python



Serialization/deserialization, flating/deflating, marshalling/unmarshalling, etc. persisting

Security concerns

- deserialization from untrusted sources, e.g. readObject may have exec-code with no instruction
- Deserialized object graph can consume huge memory, open to DoD attack
- Sensitive data exposures

Security Libraries

Apache Commons Lang SerializationUtils, & update Java for security patches

Serialization Alternatives and Future

- Custom Java NIO **ByteBuffer**, XML, **JSON**, **Protocol Buffers**, **Cap'n Proto, FlatBuffers**, **Message Pack**, Apache Thrift, Avro, Kryo etc.
- Removal of Java Serialization, Project Loom, Project Amber, Record pattern influence on sterilization and deserialization

Protocol Buffers

Protobuf serializes data in language agnostic way. E.g. gPRC(2015), GoogleCloud, EnvoyProxy uses it.

- ✓ **Simple to use, Fast** (recommended message size < 1-2 MB) Serialization & Deserialization.
- ✓ Backward compatible: add/delete new fields existing/new fields without breaking
- ✓ **Strongly Typed, has Rich Types**: scalar(bool, int32..) & non-scalar
- ✓ Boilerplate codes (**auto generated**) are **Immutable** (thread safe, has service methods).
- ✓ Converting JSON, XML messages from/to protobuf is straightforward.
- ✓ **IDL based RPC frameworks**: gRPC, Apache Thrift, Ion, Bond - gPRC service [CORBA, DCOM, EJB,SOAP,REST]
- ✓ **Cross-project** support, e.g. of proto definitions are timestamp.proto and status.proto
- ✓ **Cross-language** compatibility: Different systems can have different lang. but can exchange
- ✓ data integration with messaging systems, e.g. Kafka and others

Protocol Buffers

Developer(s)	Google
Initial release	Early 2001 (internal)[1] July 7, 2008 (public)
Stable release	25.2 / 9 January 2024; 10 days ago[2]
Repository	github.com/protocolbuffers/protobuf
Written in	C++, C#, Java, Python, JavaScript, Ruby, Go, PHP, Dart
Operating system	Any
Platform	Cross-platform
Type	serialization format and library, IDL compiler
License	BSD
Website	protobuf.dev

Characteristics or Limitations

- Protobuf2 to Protobuf3 – migration needed, e.g. tags, required fields (protobuf2), etc.
- messages are serialized into a binary wire format which is compact, forward- and backward-compatible, but not self-describing
- Protobufs are not designed for messages > 64MB, recommended split into multiple chunks 1-2 MB. Protocol buffers tend to assume that entire messages can be loaded into memory at once and are not larger than an object graph.
- Not good for the purposes of storing something e.g. a text document, or a database dump.

FlatBuffers

Zero-copy serialization format – designed for shorter serialization time and use less memory than protobuf.

FlatBuffers (compiler here) is an efficient cross platform serialization **Google** library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust and Swift. Used by Google,Tabnine, Facebook, etc.

- Define a schema as **.fbs** files, and generate a code for favorite language
- Memory efficiency and speed - Deserialization speed
- Much efficient for large messages
- Strongly typed
- Cross platform code with no dependencies
- Access to serialized data without unpacking (e.g. just to get a snippet on big data)
- FlatBuffers are also ideal for usage with *mmap* (or streaming)

- FlexBuffers - The schema-less version of FlatBuffers have their own encoding.
  - should not be used in cases where we have to change the data.
  - Usage is less practical than protobuf
  - Documentation is less explanative
  - More used in C/C++ e.g. gRPC

FlatBuffers

Original author(s)	Wouter van Oortmerssen
Developer(s)	Derek Bailey
Initial release	June 17, 2014; 9 years ago[1]
Stable release	23.3.3 / March 3, 2023; 10 months ago[2]
Repository	github.com/google/flatbuffers
Written in	C++
Operating system	Android, Microsoft Windows, Mac OS X, Linux
Type	serialization format and library, IDL compiler
License	Apache License 2.0
Website	flatbuffers.dev

Cap’n Proto

[Cap’n Proto](#) is an insanely fast [data interchange format](#) and capability-based RPC system.

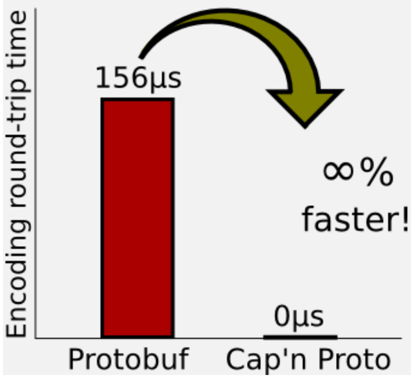
- **Random access** - can read just one field of a message no need to parse whole
- **mmap** - Read a large Cap’n Proto file by memory-mapping it.
- Inter-language communication: Two languages can easily operate on the **same in-memory data structure**. E.g. calling C++ code from, Java or Python.
- Inter-process communication - processes running on the same machine can share a Cap’n Proto message via shared memory. No need to pipe data
- **Tiny generated code** - Protobuf generates a lot whereas Cap’n code smaller
- **Tiny runtime library:** Runtime library is much smaller die to Cap’n format.
- **Incremental reads** - easy to start processing message before receiving all of **Time-traveling RPC:** Cap’n Proto features an RPC system that implements [time travel](#) ([Promise Pipelining](#)) such that call results are returned to the client before the request even arrives at the server!

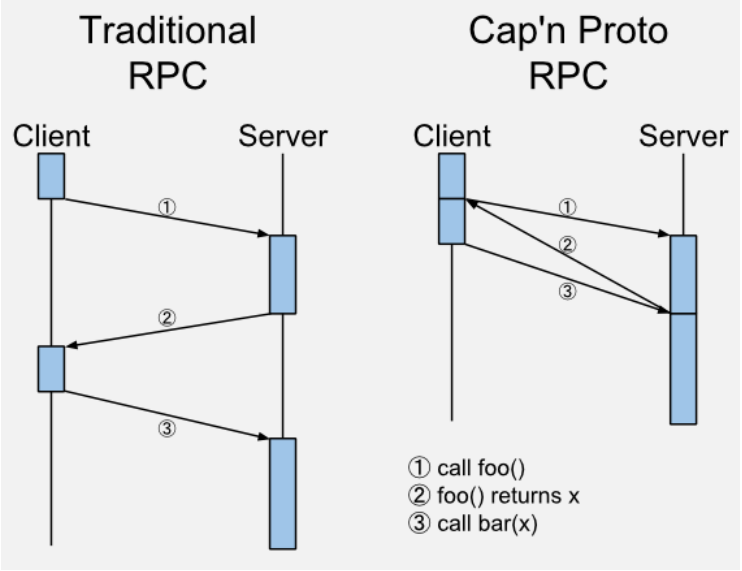
- Alternative to gRPC but no community you have to implement it
- [Message sizes](#) bigger comparing to Protobuf
- Symantec removed the installation win32 ;)

Protocol Bu

Cap'n Proto

Original author(s)	Kenton Varda
Stable release	0.10.4 / April 13, 2023; 9 months ago
Repository	github.com/capnproto/capnproto
Written in	C++
Type	Remote procedure call framework, serialization format and library, IDL compiler
License	MIT License
Website	capnproto.org





SBE

Simple Binary Encoding (SBE)

SBE is a binary representation for encoding/decoding messages to support **low-latency streaming**.
Ref-implementation FIX-SBE. It is an **OSI layer 6** presentation for encoding and decoding binary application messages **for low-latency financial applications** – e.g. NASDAQ, Trading, Stocks, …

The Message Structure
In order to preserve streaming semantics, **a message must be capable of being read or written sequentially, with no backtrack**. This eliminates extra operations — like dereferencing, handling location pointers, managing additional states, etc. – and utilizes hardware support better to keep maximum performance and efficiency.


```xml
<?xml version="1.0" encoding="UTF-8"?>
<sbe:messageSchema xmlns:sbe="http://fixprotocol.io/2016/sbe"
  package="com.baeldung.sbe.stub" id="1" version="0" semantic
  description="A schema represents stock market data.">
    <types>
        <composite name="messageHeader"
          description="Message identifiers and length of mess
            <type name="blockLength" primitiveType="uint16"/>
            <type name="templateId" primitiveType="uint16"/>
            <type name="schemaId" primitiveType="uint16"/>
            <type name="version" primitiveType="uint16"/>
        </composite>
        <enum name="Market" encodingType="uint8">
            <validValue name="NYSE" description="New York Sto
            <validValue name="NASDAQ"
              description="National Association of Securities
        </enum>
        <type name="Symbol" primitiveType="char" length="4" c
```

- SBE is well suited for fixed-size data like numbers, bitsets, enums, and arrays.
- SBE provides the ability to define our schemas via **XML / XSD**.
- Enables on-the-fly decoding of messages
- Produces a binary encoding suitable for low-latency financial trading

- SBE isn't well suited for variable-length data types like string **and blob**.

JSON and CSV, XML, YAML etc

All of these formats define a set of rules to represent and transmit data across apps., servers, operating systems, etc.
JSON format characteristics: Easy to manipulate, mostly used REST resource, human readable (self describing), better schema support. Contains basic elements: Objects {}, Object members, Arrays [], Values (string, object, array, or literal).
3rd party JSON libs for JAVA: Jackson, Gson, json-io, Genson, Json-P


```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "courses": ["Math",
"History", "Science"]
}
```


Limitations:

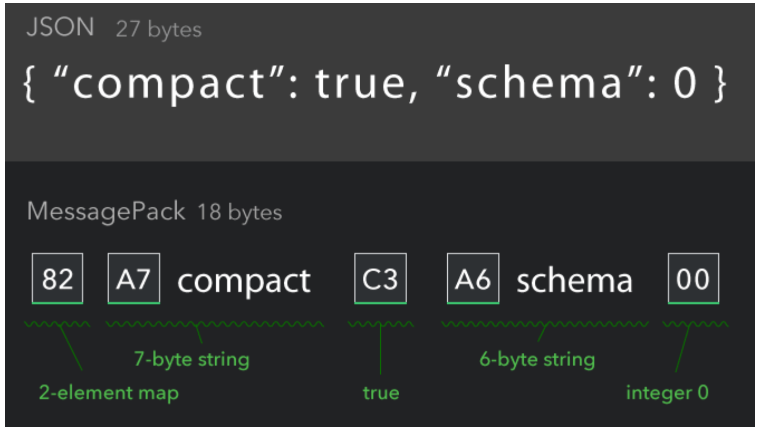
- No ability to add comments(can't document) or attribute tags to JSON
- No rich data types – e.g. date type: anomalies in a string representation
- Slow performance comparing to binary protocols
- Inherits JS sloppy architecture on types
- JSON doesn’t do everything that XML does, e.g. No industry-wide equivalent to XLST (transforms), SOAP WS-Security
- JSON data has no fixed structure. You can not verify the consistency of the data stored in the JSON field without parsing
- JSON is more difficult to produce than code
- Use more efficient BSON (Binary JSON), or **MessagePack** to fulfill some limitations


```
\x16\x00\x00\x00
\x02hello\x00\x06\x00\x00\x00world\x00
\x00
```


JSON is a great choice when transferring small amounts of data that is short-lived, not complex, and verified correctness is not a concern

Message Pack

Zero-copy serialization format



MessagePack is a computer data interchange format. It is a binary form for representing simple data structures like arrays and associative arrays.

- MessagePack aims to be as compact and simple as possible.
- The official implementation is available for: C, C++, C#, D, Erlang, Go, Haskell, Java, JavaScript (NodeJS), Lua, OCaml, Perl, PHP, Python, Ruby, Rust, Scala, Smalltalk, and Swift
- Data types: nil, bool, int, float, str, bin, array, map, ext, timestamp

Limitations

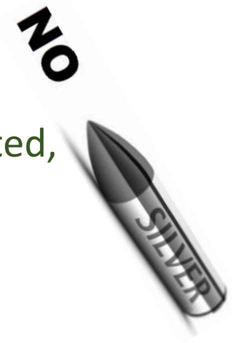
- MessagePack is more compact than JSON, but imposes limitations on array and integer sizes. On the other hand, it allows binary data and non-UTF-8 encoded strings.
- Schema less format is lack of flexibility, less practical
- Compared to BSON, MessagePack is more space-efficient. BSON is designed for fast in-memory manipulation, whereas MessagePack is designed for efficient transmission over the wire.
- The Protocol Buffers format provides a significantly more compact transmission format than MessagePack because it doesn't transmit field names.

Comparison Matrix

Feature	Protobuf	JSON
Data types, Schemas, Boiler plate code generation	define message format, rules and schemas to define messages. Rich types [ANY, ONE OF, ENUM, etc], gRPC etc. Less boilerplate - proto generated classes	Only message format, schemaless. No rich types, no classes, functions Rely on hand-written ad-hoc boilerplate code to handle the encoding and decoding
Support for evolving schema	Since there is a proto schema, things are less error prone. Improved communication between services and better data consistency	No strict contract between clients/API providers, errors due to field type ambiguity (int vs long, date etc.).
Usage	Protobuf is much focused towards performance benefits. Protobuf is more favorable in intersystem communication (interoperability), IoT, IoV. e.g. DAP server <-> InterTest server, IoT devices	JSON is more on easy of use and generic adoption. API class directly from the browser (web-apps), or expansive datasets, or config-files then JSON is the right fit
Service frameworks	gRPC (since 2015) uses the Protobuf message format. It uses HTTP/2 which faster. Using TCP connection, HTTP/2 supports multiple data streaming from the server alongside the traditional request-response. Challenge needs a middleware to work with browser. Protobuf use in **REST** via ProtobufHttpMessageConverter	REST receives messages using the JSON (& etc) format. REST uses HTTP/1.1. REST also lacks more on multiplex, only traditional request-response. Also **JSON-RPC**
Serialization speed and message size	Fastest for small packs Smaller messages → less data to transfer → potentially faster transfer [MAX data packet is 1.5 KB) (See MTU)[Network layer of OSI model L3]	Slower, much bandwidth → data (textual nature) capacity raises concern in network transmissions.

Protobuf boasts impressive speed and size efficiency, schema evolution, courtesy of its binary serialization. Moreover backward compatible, has rich types and gRPC (less adopted, future oriented) services support. **Language agnostic**. Not good to store data in DB, files systems.

Protobuf provides more compact messages than MessagePack



JSON is human readable, has browser support and globally accepted (adoption). Used in REST(widely adopted), JSON-RPC services. Need **spec-libraries** per language. Good to store data.
MessagePack is like JSON but in binary form, small and faster. Space efficient than BSON. Schemaless protocol, returns dynamic-data structure **no automatic structure check**

Comparison Matrix

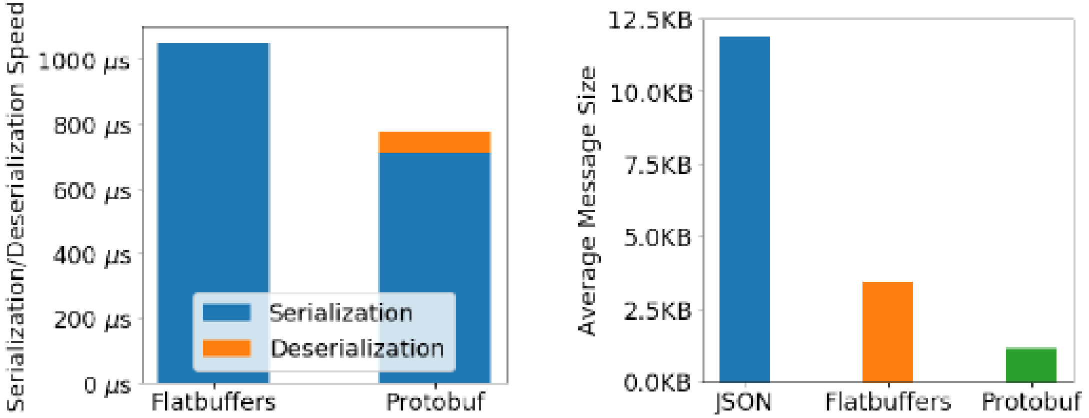
Simple Binary Encoding

Feature	Protobuf	Cap'n Proto	SBE	FlatBuffers
Schema evolution	yes	yes	caveats	yes
Zero-copy	no	yes	yes	yes
Random-access reads	no	yes	no	yes
Safe against malicious input	yes	yes	yes	opt-in upfront
Reflection / generic algorithms	yes	yes	yes	yes
Initialization order	any	any	preorder	bottom-up
Unknown field retention	removed in proto3	yes	no	no

PUBLIC research works / books / blogs:

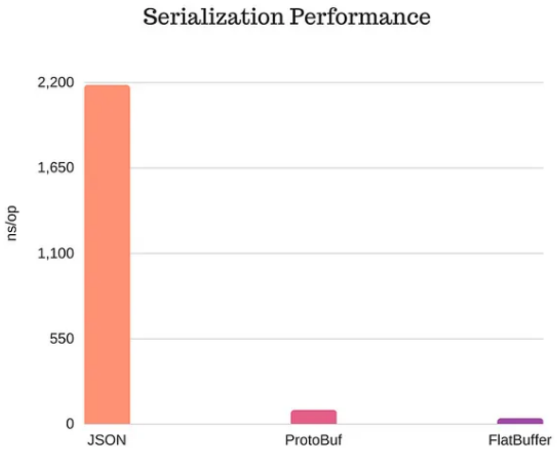
1. Networking conference: Performance Comparison of Messaging Protocols and Serialization Formats (**Research work - pdf**) Protobuf beats Flatbuffers, as it typically achieves three times smaller serialized message size and has faster serialization speed ….

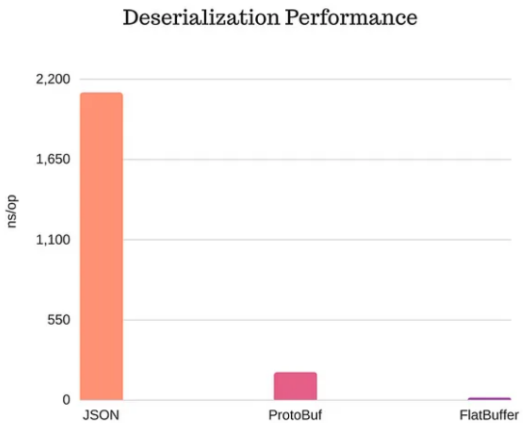
2. JAVA Expect: Joshua Bloch [Designer of JAVA Collections, java.math api] – Note in Effective Java (3rd) – 2018 book: In summary, serialization is dangerous and should be avoided. If you are designing a system from scratch, use a cross-platform structured-data representation such **as JSON or protobuf** instead.
FYI: Converting json-messages to protobuf is easy with Protobuf.

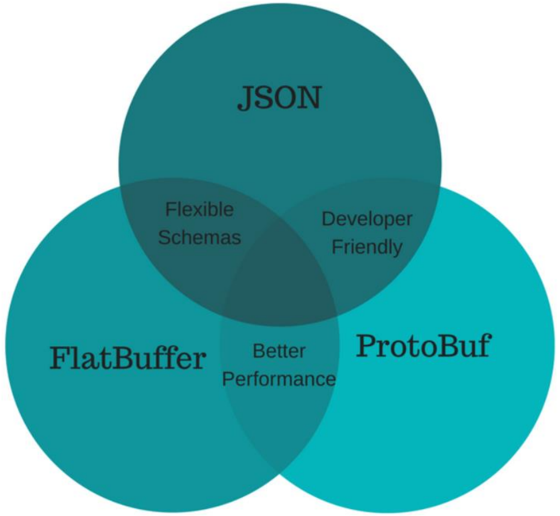


(a) Serialization speed. (b) Serialized message size.

3. Research work - medium blog







Protobuf has the best performance (speed) for smaller messages (especially < 1-2 MB). Actions can be done via **wrapper** or gPRC solution in Protobuf. Mostly used data-format at Google. It has **bigger community.**

FlatBuffer (from Google – efficient for bigger messages) and **Cap'n Proto** (former Protobuf maintainers) wins over Protobuf only on big messages. Deserialization is always faster.
Cap'n Proto is new and claims to be best, but **less community**, and users are mainly C/C++, less usages for other languages so far.

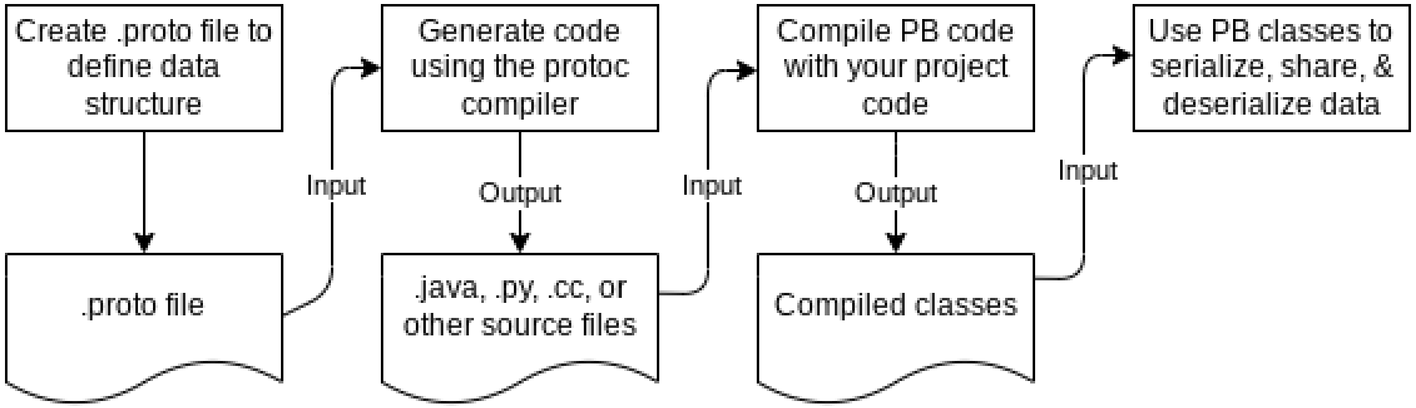
Protobuf buffer workflow

1. Download your OS relevant protobuf-compiler & set env-variable for it
2. Define maven dependencies and plugins, see <GitHub-link>

```
C:\Users\as892333>protoc --version
libprotoc 25.2
```

```xml
<dependency>
   <groupId>com.google.protobuf</groupId>
   <artifactId>protobuf-java</artifactId>
   <version>${protobuf.version}</version>
</dependency>
```

```xml
<plugins>
   <plugin>
      <groupId>org.xolstice.maven.plugins</groupId>
      <artifactId>protobuf-maven-plugin</artifactId>
      <version>0.6.1</version>
      <extensions>true</extensions>
```



Protobuf message definition

3. Define your data-structure (message) using protobuf format, e.g. ?.proto, for style guide look here

When defining .proto files, specify field optional (default) or repeated (proto2 has also required) in proto3

A field type: Scalar types: double, float, int32|64, uint32|64, sint32|64, fixed32|64, sfixed32|64, bool, string, bytes[ByteString Provides conversions to and from byte[], String, ByteBuffer, InputStream, OutputStream, CodedInputStream] .
Non-scalar types: Map, List (with repeated keyword), enum, neste classes, and compound types (oneOf, Anyof) See more detailed
More about updating message type , unknown fields

```
syntax = "proto3";  import "google/protobuf/any.proto";
package myCompundDT;
option java_package =  "net.sahet.tt.point.compound";
option java_multiple_files = true;//false – all in one file
option java_generic_services = true; //generates gPRC interface
/* I am a multiline Comment
will be in generated class
*/
message Divadlo {
 string name = 1;
 repeated AnimalType type = 2;
 repeated google.protobuf.Any people_inside = 3;
 oneof availableEmployees {
  int32 count = 4;
  string experts = 5;
 }
map<int32, string> errors_detailed_by_id = 7;
bytes bytes_dat = 8; //sequence of 8 bytes, can store 2 GB data
int32 err_code = 1;
enum AnimalType {
 CAT = 0;
 DOG = 1;
}

message Employee {
reserved 2, 15, 9 to 11;
reserved "foo", "bar";
string name = 1;
}
message Viewer {
 string name = 1;
 int32 age = 2;
}
service AnInterface {
 rpc someMethod (SomeRequest) returns (SomeResponse) { }
}
message SomeRequest {  }
}
```

Assigning Field Numbers

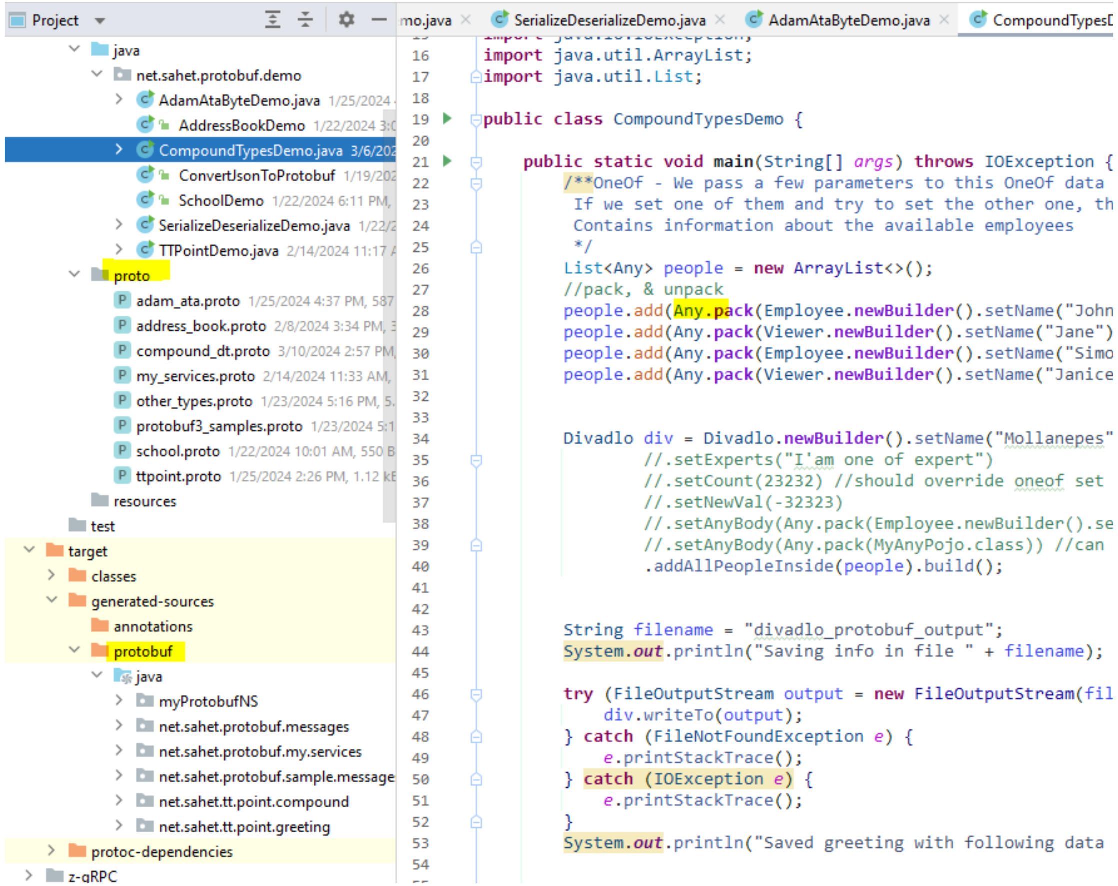
Tag numbers range 1 and 536,870,911 with below restrictions:
1) Filed tag number must be unique
2) 19,000 to 19,999 are reserved for the Protobuf implementation
3) You can not use previously used tags (even deleted one)
4) First 1-15 has one byte, 16-2047 two bytes, see more encoding

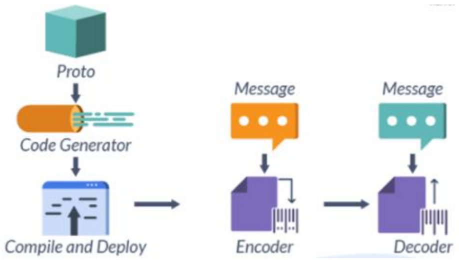
Generated boilerplate code, serialization and deserialization

4. Generate[compile] messages (classes) from Protobuf file using 'protoc' compiler

```
>protoc -I=src/main/proto --java_out=src/main/java person.proto //or --python_out, cpp_out
>mvn protobuf:compile -DprotocExecutable="C:/apps/protoc-25.2-win64/bin/protoc.exe“
```

5. Start using boilerplate code **(immutable classes)** has all utils, builders, parsers, encoders, etc…





See:

- byte[] **toByteArray**();

//parses a message from the given byte array.

- **parseFrom(byte**[] data);:

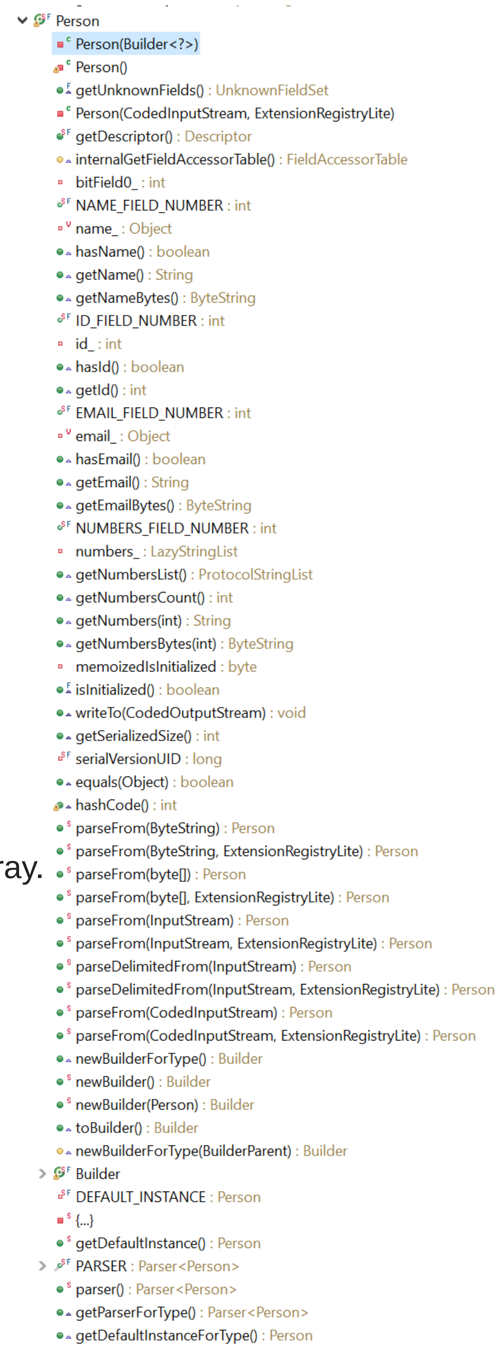
//serializes message

- void **writeTo**(OutputStream output);

//deserialize

- **parseFrom**(InputStream input);

The <CODE/> SHOW



Proto Dos and Don't e.g. Use Well-Known Types and Common Types API Best Practices see Java

- *Use **PascalCase** (with an initial capital) for message names – for example, MyServerRequest*
- *Use **lower_snake_case** for field names (including one of field and extension names) – e.g. , **title_name**. JAVA or other lang. spec. conventions used during the code generation, e.g. java.lang.String **getTitleName**();*
- *Do not re-use TAG(serial) numbers even deleted one, better make them **reserved** by tag_name or field_name, e.g.* reserved **2, 15, 9 to 11;** reserved "foo", "bar"; so consequences of re-using field numbers
- *First TAG numbers 1-15, encoded with 1 byte, then 2 bytes, so consider keeping fields optimized*
- *Combining messages (e.g. message, enum, map, and service) in single large .proto file, may lead to dependency bloat*

- *Add comments via //, or /*…*/ syntax*
- *Removing enum values is a breaking change for persisted protos, better mark it:* PHONE_TYPE_WORK = 3 [deprecated=true];
- *Extensions: reuse definitions via importing e.g.* import "school.proto";

Data Type	Default value
Int32 / Int64	0
Float/double	0.0
String	Empty string
Boolean	False
Enum	First Enum item, that is the one with "index=0"
Repeated type	Empty list
Map	Empty Map
Nested Class	null

Encoding	Sample types	Length
varint	int32, uint32, int64	Variable length
fixed	fixed32, float, double	Fixed 32-bit or 64-bit length
byte sequence	string, bytes	Sequence length

Advanced Usage

Services - Protocol Buffers are similar to the Apache Thrift, Ion, and Microsoft Bond protocols, offers gPRC
Reflection - You can iterate over the fields of a message and manipulate their values without writing your code, and use it
e.g.1 Converting protocol messages to and from other encodings, such as XML or JSON.
e.g.2 More advanced use of reflection to find differences between two messages of the same type
Reflection is provided as part of the Message and Message.Builder interfaces.

Extending a Protocol Buffer
If you want to "improve" the protocol buffer's definition and want your new buffers to be backwards-compatible, and your old buffers to be forward-compatible – then there are some rules you need to follow.

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.

```
syntax = "proto3";

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

message SelfDescribingMessage {
// FileDescriptorProtos which describe the type and its dependencies.
google.protobuf.FileDescriptorSet descriptor_set = 1;
 // The message and its type, encoded as an Any message.
google.protobuf.Any message = 2;
}
```