Services Evolution:
Required Is Forever

Natalia Pryntsova

Services evolution: required is forever

ACCU 2021
March 11, 2021

Natalia Pryntsova
Team Leader, Portfolio Enterprise Infrastructure Engineering

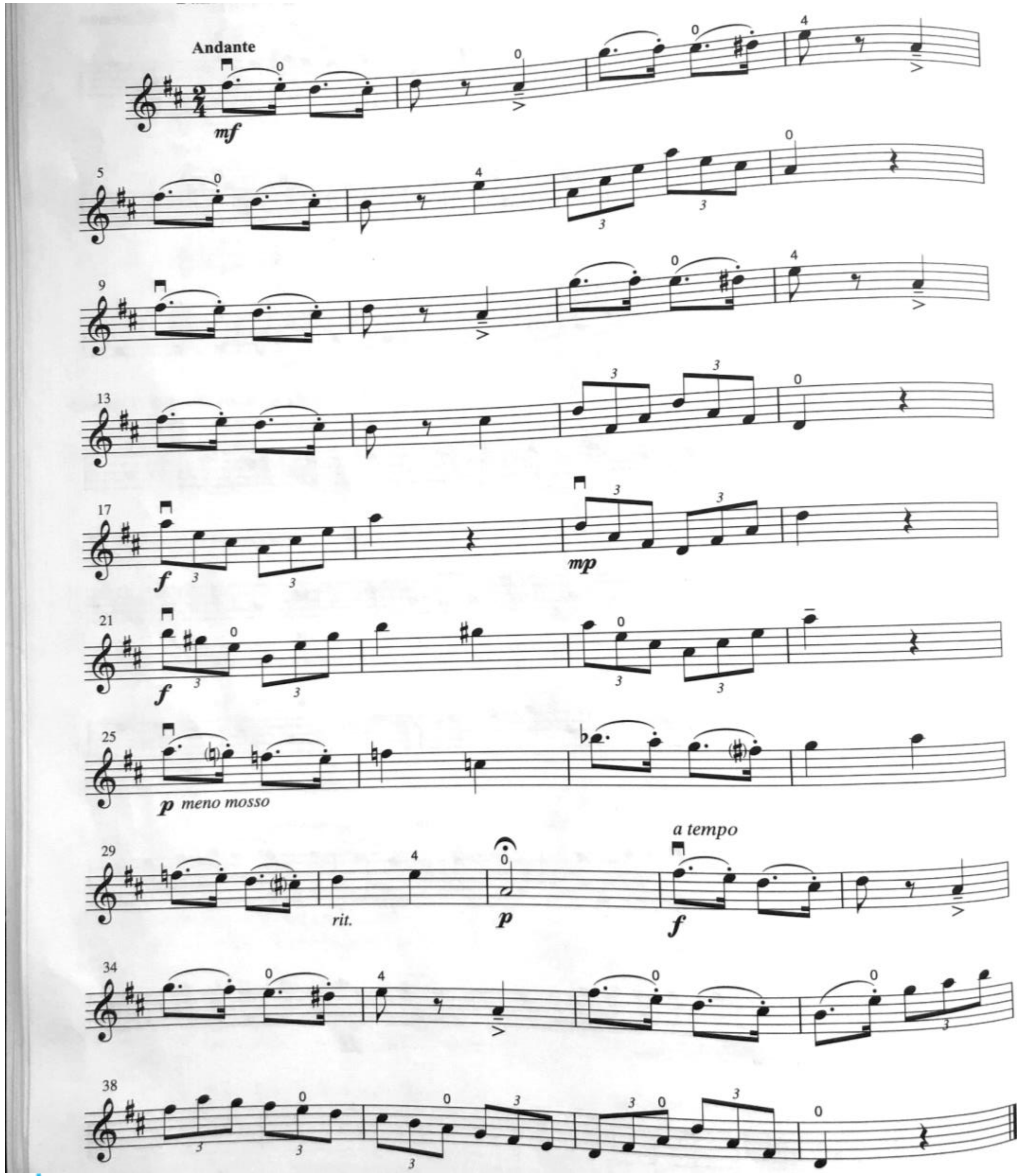
TechAtBloomberg.com

MIDI

- No significant changes since 1983

  Note On 9<ch> <note> <velocity>
  92 60 96 Ch.3 Note On C4, forte “ff”

- MIDI 2.0 released Dec 2020

- Backwards compatibility was the main requirement

TechAtBloomberg.com

Backward/forward compatibility

“Backward compatibility is a property of a system, product, or technology that allows for interoperability with an older legacy system, or with input designed for such a system, especially in telecommunications and computing.”

Wikipedia

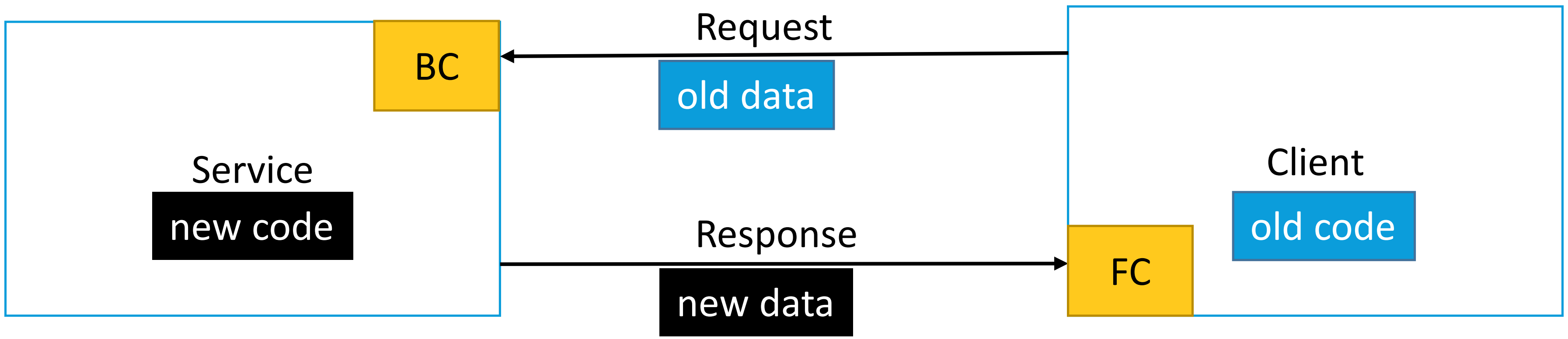
Backward compatibility: new code - old data

- Can be verified

Forward compatibility: old code - new data

- “Best efforts” basis

Backward/forward compatibility



Assuming Service POV:

- Backward compatible on request
- Forward compatible on response

Backward compatibility in practice

Compatibility is driven by serialization methods

Schema Resolution

A reader of Avro data, whether from an RPC or a file, can always parse that data because the original schema must be provided along with the data. However, the reader may be programmed to read data into a different schema. For example, if the data was written with a different version of the software than it is read, then fields may have been added or removed from records. This section specifies how such schema differences should be resolved.

We refer to the schema used to write the data as the *writer's* schema, and the schema that the application expects the *reader's* schema. Differences between these should be resolved as follows:

- It is an error if the two schemas do not *match*.
  To match, one of the following must hold:
  - both schemas are arrays whose item types match
  - both schemas are maps whose value types match
  - both schemas are enums whose (unqualified) names match
  - both schemas are fixed whose sizes and (unqualified) names match
  - both schemas are records with the same (unqualified) name
  - either schema is a union
  - both schemas have same primitive type
  - the writer's schema may be *promoted* to the reader's as follows:
    - int is promotable to long, float, or double
    - long is promotable to float or double
    - float is promotable to double
    - string is promotable to bytes
    - bytes is promotable to string
- **if both are records:**
  - the ordering of fields may be different: fields are matched by name.
  - schemas for fields with the same name in both records are resolved recursively.
  - if the writer's record contains a field with a name not present in the reader's record, the writer's value for that field is ignored.
  - if the reader's record schema has a field that contains a default value, and writer's schema does not have a field with the same name, then the reader should use the default value from its field.
  - if the reader's record schema has a field with no default value, and writer's schema does not have a field with the same name, an error is signalled.
- **if both are enums:**
  if the writer's symbol is not present in the reader's enum and the reader has a `default` value, then that value is used, otherwise an error is signalled.
- **if both are arrays:**
  This resolution algorithm is applied recursively to the reader's and writer's array item schemas.
- **if both are maps:**
  This resolution algorithm is applied recursively to the reader's and writer's value schemas.
- **if both are unions:**
  The first schema in the reader's union that matches the selected writer's union schema is recursively resolved against it. if none match, an error is signalled.
- **if reader's is a union, but writer's is not**
  The first schema in the reader's union that matches the writer's schema is recursively resolved against it. If none match, an error is signalled.
- **if writer's is a union, but reader's is not**
  If the reader's schema matches the selected writer's schema, it is recursively resolved against it. If they do not match, an error is signalled.

A schema's "doc" fields are ignored for the purposes of schema resolution. Hence, the "doc" portion of a schema may be dropped at serialization.

Protocol Buffers

Search

Overview

Developer Guide
Language Guide (proto2)
Language Guide (proto3)
Style Guide
Encoding
Techniques
Add-ons

Tutorials
Tutorials Overview
Basics: C++
Basics: C#
Basics: Dart
Basics: Go
Basics: Java
Basics: Python

Related Guides
gRPC

Updating A Message Type

If an existing message type no longer meets all your needs – for example, you'd like the message format to have an extra field – but you'd still like to use code created with the old format, don't worry! It's very simple to update message types without breaking any of your existing code. Just remember the following rules:

- Don't change the field numbers for any existing fields.
- If you add new fields, any messages serialized by code using your "old" message format can still be parsed by your new generated code. You should keep in mind the default values for these elements so that new code can properly interact with messages generated by old code. Similarly, messages created by your new code can be parsed by your old code: old binaries simply ignore the new field when parsing. See the Unknown Fields section for details.
- Fields can be removed, as long as the field number is not used again in your updated message type. You may want to rename the field instead, perhaps adding the prefix "OBSOLETE_", or make the field number reserved, so that future users of your `.proto` can't accidentally reuse the number.
- `int32`, `uint32`, `int64`, `uint64`, and `bool` are all compatible – this means you can change a field from one of these types to another without breaking forwards- or backwards-compatibility. If a number is parsed from the wire which doesn't fit in the corresponding type, you will get the same effect as if you had cast the number to that type in C++ (e.g. if a 64-bit number is read as an int32, it will be truncated to 32 bits).
- `sint32` and `sint64` are compatible with each other but are *not* compatible with the other integer types.
- `string` and `bytes` are compatible as long as the bytes are valid UTF-8.
- Embedded messages are compatible with `bytes` if the bytes contain an encoded version of the message.
- `fixed32` is compatible with `sfixed32`, and `fixed64` with `sfixed64`.
- For `string`, `bytes`, and message fields, `optional` is compatible with `repeated`. Given serialized data of a repeated field as input, clients that expect this field to be `optional` will take the last input value if it's a primitive type field or merge all input elements if it's a message type field. Note that this is **not** generally safe for numeric types, including bools and enums. Repeated fields of numeric types can be serialized in the packed format, which will not be parsed correctly when an `optional` field is expected.
- `enum` is compatible with `int32`, `uint32`, `int64`, and `uint64` in terms of wire format (note that values will be truncated if they don't fit). However be aware that client code may treat them differently when the message is deserialized: for example, unrecognized proto3 `enum` types will be preserved in the message, but how this is represented when the message is deserialized is language-dependent. Int fields always just preserve their value.
- Changing a single value into a member of a **new** `oneof` is safe and binary compatible. Moving multiple fields into a new `oneof` may be safe if you are sure that no code sets more than one at a time. Moving any fields into an existing `oneof` is not safe.

TechAtBloomberg.com

Backward compatibility in practice

In practice, it is still challenging to meaningfully evolve service APIs while maintaining compatibility:

- Growth via addition of optional fields; some of them are not really optional
- Code branching for different major versions
- Following elaborate compatibility rules

```python
class GetUserRequest:
    id: int
    user_name: str
    first_name: str
    last_name: str
    description: str = None
    #.... x 10 optional fields
    some_other_flag: bool = None
```

```python
def entrypoint_v1()
    pass

def entrypoint_v2():
    pass
```

Apache Avro

No tags, no field names or field ids in serialized data, how is this possible?

Schema

{} avro_example.avsc > ...

```
{
    "type": "record",
    "name": "test",
    "fields": [
        {
            "name": "a",
            "type": "long"
        },
        {
            "name": "b",
            "type": "string"
        }
    ]
}
```

Data

```
{ "a": 127, "b": "foo" }
```

Serialized data with schema


```
00000000  4f 62 6a 01 04 14 61 76  72 6f 2e 63 6f 64 65 63  |Obj...av
00000010  08 6e 75 6c 6c 16 61 76  72 6f 2e 73 63 68 65 6d  |.null.av
00000020  61 dc 01 7b 22 74 79 70  65 22 3a 20 22 72 65 63  |a..{"typ
00000030  6f 72 64 22 2c 20 22 6e  61 6d 65 22 3a 20 22 74  |ord", "n
00000040  65 73 74 22 2c 20 22 66  69 65 6c 64 73 22 3a 20  |est", "f
00000050  5b 7b 22 74 79 70 65 22  3a 20 22 6c 6f 6e 67 22  |[{"type"
00000060  2c 20 22 6e 61 6d 65 22  3a 20 22 61 22 7d 2c 20  |, "name"
00000070  7b 22 74 79 70 65 22 3a  20 22 73 74 72 69 6e 67  |{"type":
00000080  22 2c 20 22 6e 61 6d 65  22 3a 20 22 62 22 7d 5d  |", "name
00000090  7d 00 11 3c c2 b5 22 f5  bd f4 bd ab 88 e7 f6 35  |}..<..".
000000a0  46 10 02 0c fe 01 06 66  6f 6f 11 3c c2 b5 22 f5  |F......f
000000b0  bd f4 bd ab 88 e7 f6 35  46 10                    |.......5
```


Serialized data only


```
00000000  fe 01 06 66 6f 6f                                 |...foo|
```

Varints encoding

Value	First byte	Second byte	Third byte	Binary
0	00000000			0000
1	00000001			0001
2	00000010			0010
...				
127	01111111			01111111
128	10000000	00000001		10000000
129	10000001	00000001		10000001
130	10000010	00000001		10000010
...				
16,383	11111111	01111111		00111111 11111111
16,384	10000000	10000000	00000001	01000000 00000000
16,385	10000001	10000000	00000001	01000000 00000001

Idea: small integers should take little space

High-order bit of each byte reserved to indicate if there are more bytes to read

Identify continuation bits:
FE 01 = 11111110 00000001

Least significant group is first, so swap the order:
0000001 1111110 = 254

Bonus: self-delimiting!

Zig zag encoding

Value	**Binary**
0	0000
-1	0001
1	0010
-2	0011
2	0100
...	
-64	0111 1111
64	1000 0000
…	
-127	1111 1101
127	1111 1110

Idea: small **signed** integers should take little space

(i >> bitlength-1) ^ (i << 1)

Example:
127 becomes 254

Protocol Buffers

Uses field numbers and wire types

Code-generated classes using protoc: python_out, cpp_out options

Schema

```
pb_example.proto
syntax = "proto3";
message test {
    int64 a = 1;
    string b = 2;
}
```

Data – using generated class

```
my_test = myschema.test()
my_test.a = 127
my_test.b = "foo"
```

Serialized data

```
00000000  08 7f 12 03 66 6f 6f                              |....foo|
```

Note 08 and 12 (blue dots) is combo of field number and wire types

varint ((field_number << 3) | wire_type)

Also int64 is not zig-zagged as there are separate signed types in proto

Bloomberg

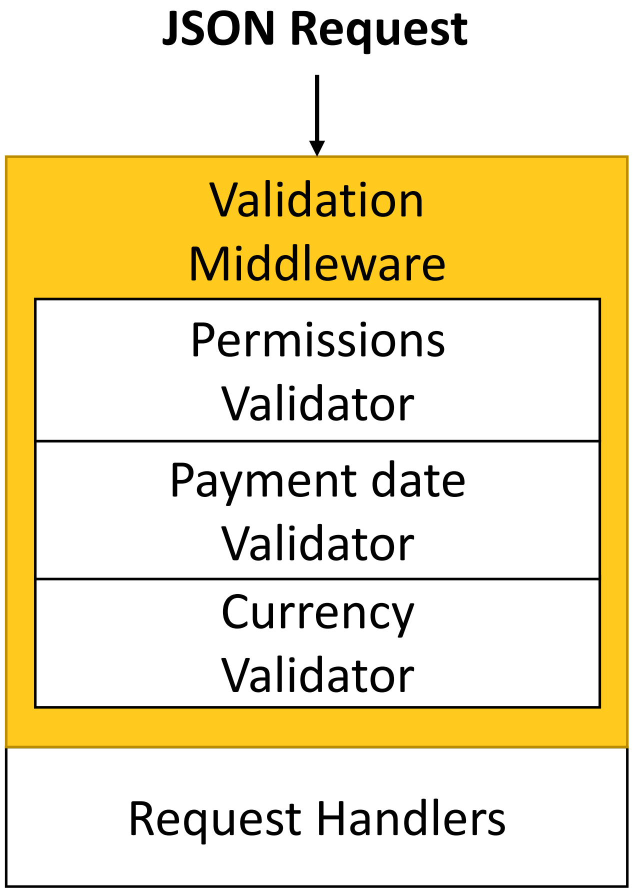
Engineering

Schema-less-ness

- Schema actually still exists, but is **implied through code**
- Need to validate inputs – middleware + collection of validators
- Various JSON parsers deal with numbers differently

```
> (10765432100123456789).toString()
<· "10765432100123458000"
```

```
>>> import json
>>> j = '{"id": 10765432100123456789 }'
>>> parsed_j = json.loads(j)
>>> print(parsed_j)
{'id': 10765432100123456789}
```



Useful patterns

- Ignore unknown fields
  - Clients must ignore any data they do not understand (do not have in its schema)
  - Otherwise, it would not be possible to add even an optional/defaulted field

- Do not discard unknown fields
  - Useful when serialized state is persisted and can be read by new/old code interchangeably

TolerantReader pattern by Martin Fowler: “only take the elements you need, ignore anything you don’t”

Must Ignore – Must Forward

v2

```
pb_example.proto
syntax = "proto3";
message test {
    int64 a = 1;
    string b = 2;
}
```

1) Writes data:
a = 127, b = “foo”

```
08 7f 12 03 66 6f 6f
```

4) Able to read data:
a = 126, b = “foo”

v1

```
pb_example_v.proto
syntax = "proto3";
message test {
    int64 a = 1;
}
```

2) Reads data:
a = 127
Must Ignore b

3) Updates data:
a = 126
Must Forward b

```
08 7e 12 03 66 6f 6f
```

“Required” fields

```
00000000  08 7f 12 03 66 6f 6f                             |....foo|
```

For

- Communicate intention
- Less test cases to cover
  — Validation offloaded to the serialization layer.

Against

- Adding/removal breaks compatibility
- Not restrictive enough
  — Is 0 a valid transaction amount?
  — Is empty string a valid family name?

Is a removal of an optional field always backward compatible?

Summary

- Time for evolution and change comes for all APIs
- Achieving compatibility is still difficult despite a large variety of serialization techniques
- Serializers drive compatibility rules and by extent services evolution
- Consider a “tolerant reader” approach and if “required” fields are actually required

TechAtBloomberg.com

Bloomberg

Engineering

Thank you!

Natalia Pryntsova
npryntsova@bloomberg.net

TechAtBloomberg.com

Bloomberg
Engineering