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 Avro file or a file, can always parse that data because the original scheme must be provided along with the data. However, the reader may be provided with data written using a different schema. For example, if the data was written with a different version of the schema, it is possible that fields may have been added or removed from records. The writer specifies how such scheme differences should be resolved.

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

- **If both are records:**
  - If the ordering of fields may be different: fields are matched by name.
  - If the ordering of fields may not be different: fields are matched 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 writer's record contains a field with a value that the reader does not expect, the value is ignored.
  - If the reader's record schema has a field with no default value, and a writer's schema does not have a field with the same name, then a value is created.

- **If both are arrays:**
  - The ordering of the elements may be different. Elements are matched by name.

- **If both are maps:**
  - The ordering of elements may be different. Elements are matched by key.

- **If both are unions:**
  - The ordering of alternatives may be different. Alternatives are matched by key.

- **If both are sets:**
  - The ordering of the elements may be different. Elements are matched by value.

- **If both are enum:**
  - The ordering of members may be different. Members are matched by name.

- **If both are protocols:**
  - The ordering of members may be different. Members are matched by protocol.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are integers:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are booleans:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are nulls:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are strings:**
  - The ordering of bytes may be different. The fields are matched by value.

- **If both are doubles:**
  - The ordering of bytes may be different. The fields are matched by value.
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
Apache Avro

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

**Schema**

```json
{} avro_example.avsc
```

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

**Data**

```json
{ "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 6f 6d 65 6c 6f 6f 6e 65 6d 65 6d |.null.av
00000020 61 dc 01 7b 22 74 79 70 20 20 22 72 72 3a 20 20 |a..{"typ
00000030 6f 72 22 22 22 22 6e 65 6d 65 6c 6f 6f 6e 65 6d |r"n"
00000040 69 73 74 22 22 22 22 22 22 69 65 6e 65 6d 65 6d |est","f
00000050 5b 7d 22 74 79 70 20 22 65 6e 65 6d 65 6c 6f 6f |["
00000060 20 20 22 22 22 22 22 22 22 22 22 22 22 22 22 |"
00000070 7b 22 22 22 22 22 22 22 22 22 22 22 22 22 22 |"
00000080 22 22 22 22 22 22 22 22 22 22 22 22 22 22 |"
00000090 22 22 22 22 22 22 22 22 22 22 22 22 22 22 |"
000000a0 46 10 02 0c fe 01 06 66 6f 6f 11 3c c2 b5 22 |6
000000b0 2f 2c 22 22 22 22 22 22 22 22 22 22 22 22 |"
```

**Serialized data only**

```
00000000 fe 01 06 66 6f 6f
```

|...foo|
### Varints encoding

<table>
<thead>
<tr>
<th>Value</th>
<th>First byte</th>
<th>Second byte</th>
<th>Third byte</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td></td>
<td>00000000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>00000001</td>
<td></td>
<td>00000001</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>00000010</td>
<td></td>
<td>00000010</td>
<td>0010</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>01111111</td>
<td></td>
<td>01111111</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>10000000</td>
<td>00000001</td>
<td>10000000</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>10000001</td>
<td>00000001</td>
<td>10000001</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>10000010</td>
<td>00000001</td>
<td>10000010</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16,383</td>
<td>11111111</td>
<td>01111111</td>
<td>00111111</td>
<td>11111111 11111111</td>
</tr>
<tr>
<td>16,384</td>
<td>10000000</td>
<td>10000000</td>
<td>00000001</td>
<td>01000000 00000000</td>
</tr>
<tr>
<td>16,385</td>
<td>10000001</td>
<td>10000000</td>
<td>00000001</td>
<td>01000000 00000001</td>
</tr>
</tbody>
</table>

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 111110 = 254

Bonus: self-delimiting!
## Zig zag encoding

<table>
<thead>
<tr>
<th>Value</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>-1</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>0010</td>
</tr>
<tr>
<td>-2</td>
<td>0011</td>
</tr>
<tr>
<td>2</td>
<td>0100</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-64</td>
<td>0111 1111</td>
</tr>
<tr>
<td>64</td>
<td>1000 0000</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>-127</td>
<td>1111 1101</td>
</tr>
<tr>
<td>127</td>
<td>1111 1110</td>
</tr>
</tbody>
</table>

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

\[(i >> \text{bitlength}-1) \^ (i << 1)\]

Example: 127 becomes 254
Protocol Buffers

Uses field numbers and wire types
Code-generated classes using protoc: python_out, cpp_out options

**Schema**

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

**Serialized data**

```
00000000 08 7f 12 03 66 6f 6f
```

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

```python
varint ((field_number << 3) | wire_type)
```

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

**Data – using generated class**

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

```python
>>> (10765432100123456789).toString()
'10765432100123456789'
```

```
>>> 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

```protobuf
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

```protobuf
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

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
Thank you!

Natalia Pryntsova
npryntsova@bloomberg.net