Last Published: 08/02/2017 05:13:20

PDF

Apache Avro™ 1.8.2 Specification

Introduction

This document defines Apache Avro. It is intended to be the authoritative specification. Implementations of Avro must adhere to this document.

Schema Declaration

A Schema is represented in JSON by one of:

- A JSON string, naming a defined type.
- A JSON object, of the form:

  ```
  {"type": "typeName" ...attributes...}
  ```

  where *typeName* is either a primitive or derived type name, as defined below. Attributes not defined in this document are permitted as metadata, but must not affect the format of serialized data.
- A JSON array, representing a union of embedded types.

Primitive Types

The set of primitive type names is:

- `null`: no value
- `boolean`: a binary value
- `int`: 32-bit signed integer
- `long`: 64-bit signed integer
- `float`: single precision (32-bit) IEEE 754 floating-point number
- `double`: double precision (64-bit) IEEE 754 floating-point number
- `bytes`: sequence of 8-bit unsigned bytes
- `string`: unicode character sequence

Primitive types have no specified attributes.

Primitive type names are also defined type names. Thus, for example, the schema "string" is equivalent to:

```
{"type": "string"}
```

Complex Types

Avro supports six kinds of complex types: records, enums, arrays, maps, unions and fixed.

Records

Records use the type name "record" and support three attributes:

- `name`: a JSON string providing the name of the record (required).
- *namespace*, a JSON string that qualifies the name;
- `doc`: a JSON string providing documentation to the user of this schema (optional).
- `aliases`: a JSON array of strings, providing alternate names for this record (optional).
- `fields`: a JSON array, listing fields (required). Each field is a JSON object with the following attributes:
    - `name`: a JSON string providing the name of the field (required), and
    - `doc`: a JSON string describing this field for users (optional).
    - `type`: A JSON object defining a schema, or a JSON string naming a record definition (required).
    - `default`: A default value for this field, used when reading instances that lack this field (optional). Permitted values depend on the field's schema type, according to the table below. Default values for union fields correspond to the first schema in the union. Default values for bytes and fixed fields are JSON strings, where Unicode code points 0-255 are mapped to unsigned 8-bit byte values 0-255.

      field default values

avro type	json type	example
null	null	null
boolean	boolean	true
int,long	integer	1
float,double	number	1.1
bytes	string	"\u00FF"
string	string	"foo"
record	object	{"a": 1}
enum	string	"FOO"
array	array	[1]
map	object	{"a": 1}
fixed	string	"\u00ff"
    - `order`: specifies how this field impacts sort ordering of this record (optional). Valid values are "ascending" (the default), "descending", or "ignore". For more details on how this is used, see the the sort order section below.
    - `aliases`: a JSON array of strings, providing alternate names for this field (optional).

For example, a linked-list of 64-bit values may be defined with:

```
{
  "type": "record",
  "name": "LongList",
  "aliases": ["LinkedLongs"],                      // old name for this
  "fields" : [
    {"name": "value", "type": "long"},             // each element has a long
    {"name": "next", "type": ["null", "LongList"]} // optional next element
  ]
}
```

Enums

Enums use the type name "enum" and support the following attributes:

- `name`: a JSON string providing the name of the enum (required).
- *namespace*, a JSON string that qualifies the name;
- `aliases`: a JSON array of strings, providing alternate names for this enum (optional).
- `doc`: a JSON string providing documentation to the user of this schema (optional).
- `symbols`: a JSON array, listing symbols, as JSON strings (required). All symbols in an enum must be unique; duplicates are prohibited. Every symbol must match the regular expression `[A-Za-z_]`

`[A-Za-z0-9_]*` (the same requirement as for names).

For example, playing card suits might be defined with:

```
{ "type": "enum",
  "name": "Suit",
  "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]
}
```

Arrays

Arrays use the type name `"array"` and support a single attribute:

- `items`: the schema of the array's items.

For example, an array of strings is declared with:

```
{"type": "array", "items": "string"}
```

Maps

Maps use the type name `"map"` and support one attribute:

- `values`: the schema of the map's values.

Map keys are assumed to be strings.

For example, a map from string to long is declared with:

```
{"type": "map", "values": "long"}
```

Unions

Unions, as mentioned above, are represented using JSON arrays. For example, `["null", "string"]` declares a schema which may be either a null or string.

(Note that when a default value is specified for a record field whose type is a union, the type of the default value must match the *first* element of the union. Thus, for unions containing "null", the "null" is usually listed first, since the default value of such unions is typically null.)

Unions may not contain more than one schema with the same type, except for the named types record, fixed and enum. For example, unions containing two array types or two map types are not permitted, but two types with different names are permitted. (Names permit efficient resolution when reading and writing unions.)

Unions may not immediately contain other unions.

Fixed

Fixed uses the type name `"fixed"` and supports two attributes:

- `name`: a string naming this fixed (required).
- *namespace*, a string that qualifies the name;
- `aliases`: a JSON array of strings, providing alternate names for this enum (optional).
- `size`: an integer, specifying the number of bytes per value (required).

For example, 16-byte quantity may be declared with:

```
{"type": "fixed", "size": 16, "name": "md5"}
```

Names

Record, enums and fixed are named types. Each has a *fullname* that is composed of two parts; a *name* and a *namespace*. Equality of names is defined on the fullname.

The name portion of a fullname, record field names, and enum symbols must:

- start with `[A-Za-z_]`
- subsequently contain only `[A-Za-z0-9_]`

A namespace is a dot-separated sequence of such names. The empty string may also be used as a namespace to indicate the null namespace. Equality of names (including field names and enum symbols) as well as fullnames is case-sensitive.

In record, enum and fixed definitions, the fullname is determined in one of the following ways:

- A name and namespace are both specified. For example, one might use `"name": "X", "namespace": "org.foo"` to indicate the fullname `org.foo.X`.
- A fullname is specified. If the name specified contains a dot, then it is assumed to be a fullname, and any namespace also specified is ignored. For example, use `"name": "org.foo.X"` to indicate the fullname `org.foo.X`.
- A name only is specified, i.e., a name that contains no dots. In this case the namespace is taken from the most tightly enclosing schema or protocol. For example, if `"name": "X"` is specified, and this occurs within a field of the record definition of `org.foo.Y`, then the fullname is `org.foo.X`. If there is no enclosing namespace then the null namespace is used.

References to previously defined names are as in the latter two cases above: if they contain a dot they are a fullname, if they do not contain a dot, the namespace is the namespace of the enclosing definition.

Primitive type names have no namespace and their names may not be defined in any namespace.

A schema or protocol may not contain multiple definitions of a fullname. Further, a name must be defined before it is used ("before" in the depth-first, left-to-right traversal of the JSON parse tree, where the `types` attribute of a protocol is always deemed to come "before" the `messages` attribute.)

Aliases

Named types and fields may have aliases. An implementation may optionally use aliases to map a writer's schema to the reader's. This faciliates both schema evolution as well as processing disparate datasets.

Aliases function by re-writing the writer's schema using aliases from the reader's schema. For example, if the writer's schema was named "Foo" and the reader's schema is named "Bar" and has an alias of "Foo", then the implementation would act as though "Foo" were named "Bar" when reading. Similarly, if data was written as a record with a field named "x" and is read as a record with a field named "y" with alias "x", then the implementation would act as though "x" were named "y" when reading.

A type alias may be specified either as a fully namespace-qualified, or relative to the namespace of the name it is an alias for. For example, if a type named "a.b" has aliases of "c" and "x.y", then the fully qualified names of its aliases are "a.c" and "x.y".

Data Serialization

Avro data is always serialized with its schema. Files that store Avro data should always also include the schema for that data in the same file. Avro-based remote procedure call (RPC) systems must also guarantee that remote recipients of data have a copy of the schema used to write that data.

Because the schema used to write data is always available when the data is read, Avro data itself is not tagged with type information. The schema is required to parse data.

In general, both serialization and deserialization proceed as a depth-first, left-to-right traversal of the schema, serializing primitive types as they are encountered.

Encodings

Avro specifies two serialization encodings: binary and JSON. Most applications will use the binary encoding, as it is smaller and faster. But, for debugging and web-based applications, the JSON encoding may sometimes be appropriate.

Binary Encoding

Primitive Types

Primitive types are encoded in binary as follows:

- `null` is written as zero bytes.
- a `boolean` is written as a single byte whose value is either `0` (false) or `1` (true).
- `int` and `long` values are written using variable-length zig-zag coding. Some examples:

value	hex
0	00
-1	01
1	02
-2	03
2	04
...	
-64	7f
64	80 01
...	

- a `float` is written as 4 bytes. The float is converted into a 32-bit integer using a method equivalent to Java's floatToIntBits and then encoded in little-endian format.
- a `double` is written as 8 bytes. The double is converted into a 64-bit integer using a method equivalent to Java's doubleToLongBits and then encoded in little-endian format.
- `bytes` are encoded as a `long` followed by that many bytes of data.
- a `string` is encoded as a `long` followed by that many bytes of UTF-8 encoded character data.

  For example, the three-character string "foo" would be encoded as the long value 3 (encoded as hex `06`) followed by the UTF-8 encoding of 'f', 'o', and 'o' (the hex bytes `66 6f 6f`):

  ```
  06 66 6f 6f
  ```

Complex Types

Complex types are encoded in binary as follows:

Records

A record is encoded by encoding the values of its fields in the order that they are declared. In other words, a record is encoded as just the concatenation of the encodings of its fields. Field values are encoded per their schema.

For example, the record schema

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

An instance of this record whose `a` field has value 27 (encoded as hex `36`) and whose `b` field has value "foo" (encoded as hex bytes `06 66 6f 6f`), would be encoded simply as the concatenation of these, namely the hex byte sequence:

```
36 06 66 6f 6f
```

Enums

An enum is encoded by a `int`, representing the zero-based position of the symbol in the schema.

For example, consider the enum:

```
{"type": "enum", "name": "Foo", "symbols": ["A", "B", "C", "D"] }
```

This would be encoded by an `int` between zero and three, with zero indicating "A", and 3 indicating "D".

Arrays

Arrays are encoded as a series of *blocks*. Each block consists of a `long` *count* value, followed by that many array items. A block with count zero indicates the end of the array. Each item is encoded per the array's item schema.

If a block's count is negative, its absolute value is used, and the count is followed immediately by a `long` block *size* indicating the number of bytes in the block. This block size permits fast skipping through data, e.g., when projecting a record to a subset of its fields.

For example, the array schema

```
{"type": "array", "items": "long"}
```

an array containing the items 3 and 27 could be encoded as the long value 2 (encoded as hex 04) followed by long values 3 and 27 (encoded as hex `06 36`) terminated by zero:

```
04 06 36 00
```

The blocked representation permits one to read and write arrays larger than can be buffered in memory, since one can start writing items without knowing the full length of the array.

Maps

Maps are encoded as a series of *blocks*. Each block consists of a `long` *count* value, followed by that many key/value pairs. A block with count zero indicates the end of the map. Each item is encoded per the map's value schema.

If a block's count is negative, its absolute value is used, and the count is followed immediately by a `long` block *size* indicating the number of bytes in the block. This block size permits fast skipping through data, e.g., when projecting a record to a subset of its fields.

The blocked representation permits one to read and write maps larger than can be buffered in memory, since one can start writing items without knowing the full length of the map.

Unions

A union is encoded by first writing a `long` value indicating the zero-based position within the union of the schema of its value. The value is then encoded per the indicated schema within the union.

For example, the union schema `["null","string"]` would encode:

- `null` as zero (the index of "null" in the union):

  ```
  00
  ```

- the string `"a"` as one (the index of "string" in the union, encoded as hex `02`), followed by the serialized string:

  ```
  02 02 61
  ```

Fixed

Fixed instances are encoded using the number of bytes declared in the schema.

JSON Encoding

Except for unions, the JSON encoding is the same as is used to encode field default values.

The value of a union is encoded in JSON as follows:

- if its type is `null`, then it is encoded as a JSON null;
- otherwise it is encoded as a JSON object with one name/value pair whose name is the type's name and whose value is the recursively encoded value. For Avro's named types (record, fixed or enum) the user-specified name is used, for other types the type name is used.

For example, the union schema `["null","string","Foo"]`, where Foo is a record name, would encode:

- `null` as `null`;
- the string `"a"` as `{"string": "a"}`; and
- a Foo instance as `{"Foo": {...}}`, where `{...}` indicates the JSON encoding of a Foo instance.

Note that a schema is still required to correctly process JSON-encoded data. For example, the JSON encoding does not distinguish between `int` and `long`, `float` and `double`, records and maps, enums and strings, etc.

Single-object encoding

In some situations a single Avro serialized object is to be stored for a longer period of time. One very common example is storing Avro records for several weeks in an Apache Kafka topic.

In the period after a schema change this persistance system will contain records that have been written with different schemas. So the need arises to know which schema was used to write a record to support schema evolution correctly. In most cases the schema itself is too large to include in the message, so this binary wrapper format supports the use case more effectively.

Single object encoding specification

Single Avro objects are encoded as follows:

1. A two-byte marker, `C3 01`, to show that the message is Avro and uses this single-record format (version 1).
2. The 8-byte little-endian CRC-64-AVRO fingerprint of the object's schema
3. The Avro object encoded using Avro's binary encoding

Implementations use the 2-byte marker to determine whether a payload is Avro. This check helps avoid expensive lookups that resolve the schema from a fingerprint, when the message is not an encoded Avro payload.

Sort Order

Avro defines a standard sort order for data. This permits data written by one system to be efficiently sorted by another system. This can be an important optimization, as sort order comparisons are sometimes the most frequent per-object operation. Note also that Avro binary-encoded data can be efficiently ordered without deserializing it to objects.

Data items may only be compared if they have identical schemas. Pairwise comparisons are implemented recursively with a depth-first, left-to-right traversal of the schema. The first mismatch encountered determines the order of the items.

Two items with the same schema are compared according to the following rules.

- `null` data is always equal.
- `boolean` data is ordered with false before true.
- `int`, `long`, `float` and `double` data is ordered by ascending numeric value.
- `bytes` and `fixed` data are compared lexicographically by unsigned 8-bit values.
- `string` data is compared lexicographically by Unicode code point. Note that since UTF-8 is used as the binary encoding for strings, sorting of bytes and string binary data is identical.
- `array` data is compared lexicographically by element.
- `enum` data is ordered by the symbol's position in the enum schema. For example, an enum whose symbols are `["z", "a"]` would sort `"z"` values before `"a"` values.
- `union` data is first ordered by the branch within the union, and, within that, by the type of the branch. For example, an `["int", "string"]` union would order all int values before all string values, with the ints and strings themselves ordered as defined above.
- `record` data is ordered lexicographically by field. If a field specifies that its order is:
    - `"ascending"`, then the order of its values is unaltered.
    - `"descending"`, then the order of its values is reversed.
    - `"ignore"`, then its values are ignored when sorting.
- `map` data may not be compared. It is an error to attempt to compare data containing maps unless those maps are in an `"order":"ignore"` record field.

Object Container Files

Avro includes a simple object container file format. A file has a schema, and all objects stored in the file must be written according to that schema, using binary encoding. Objects are stored in blocks that may be compressed. Syncronization markers are used between blocks to permit efficient splitting of files for MapReduce processing.

Files may include arbitrary user-specified metadata.

A file consists of:

- A *file header*, followed by
- one or more *file data blocks*.

A file header consists of:

- Four bytes, ASCII 'O', 'b', 'j', followed by 1.
- *file metadata*, including the schema.
- The 16-byte, randomly-generated sync marker for this file.

File metadata is written as if defined by the following map schema:

```
{"type": "map", "values": "bytes"}
```

All metadata properties that start with "avro." are reserved. The following file metadata properties are currently used:

- **avro.schema** contains the schema of objects stored in the file, as JSON data (required).

- **avro.codec** the name of the compression codec used to compress blocks, as a string. Implementations are required to support the following codecs: "null" and "deflate". If codec is absent, it is assumed to be "null". The codecs are described with more detail below.

A file header is thus described by the following schema:

```
{"type": "record", "name": "org.apache.avro.file.Header",
 "fields" : [
   {"name": "magic", "type": {"type": "fixed", "name": "Magic", "size": 4}},
   {"name": "meta", "type": {"type": "map", "values": "bytes"}},
   {"name": "sync", "type": {"type": "fixed", "name": "Sync", "size": 16}},
  ]
}
```

A file data block consists of:

- A long indicating the count of objects in this block.
- A long indicating the size in bytes of the serialized objects in the current block, after any codec is applied
- The serialized objects. If a codec is specified, this is compressed by that codec.
- The file's 16-byte sync marker.

Thus, each block's binary data can be efficiently extracted or skipped without deserializing the contents. The combination of block size, object counts, and sync markers enable detection of corrupt blocks and help ensure data integrity.

Required Codecs

null

The "null" codec simply passes through data uncompressed.

deflate

The "deflate" codec writes the data block using the deflate algorithm as specified in RFC 1951, and typically implemented using the zlib library. Note that this format (unlike the "zlib format" in RFC 1950) does not have a checksum.

Optional Codecs

snappy

The "snappy" codec uses Google's Snappy compression library. Each compressed block is followed by the 4-byte, big-endian CRC32 checksum of the uncompressed data in the block.

Protocol Declaration

Avro protocols describe RPC interfaces. Like schemas, they are defined with JSON text.

A protocol is a JSON object with the following attributes:

- *protocol*, a string, the name of the protocol (required);
- *namespace*, an optional string that qualifies the name;
- *doc*, an optional string describing this protocol;
- *types*, an optional list of definitions of named types (records, enums, fixed and errors). An error definition is just like a record definition except it uses "error" instead of "record". Note that forward references to named types are not permitted.
- *messages*, an optional JSON object whose keys are message names and whose values are objects whose attributes are described below. No two messages may have the same name.

The name and namespace qualification rules defined for schema objects apply to protocols as well.

Messages

A message has attributes:

- a *doc*, an optional description of the message,
- a *request*, a list of named, typed *parameter* schemas (this has the same form as the fields of a record declaration);
- a *response* schema;
- an optional union of declared *error* schemas. The *effective* union has `"string"` prepended to the declared union, to permit transmission of undeclared "system" errors. For example, if the declared error union is `["AccessError"]`, then the effective union is `["string", "AccessError"]`. When no errors are declared, the effective error union is `["string"]`. Errors are serialized using the effective union; however, a protocol's JSON declaration contains only the declared union.
- an optional *one-way* boolean parameter.

A request parameter list is processed equivalently to an anonymous record. Since record field lists may vary between reader and writer, request parameters may also differ between the caller and responder, and such differences are resolved in the same manner as record field differences.

The one-way parameter may only be true when the response type is `"null"` and no errors are listed.

Sample Protocol

For example, one may define a simple HelloWorld protocol with:

```
{
  "namespace": "com.acme",
  "protocol": "HelloWorld",
  "doc": "Protocol Greetings",

  "types": [
    {"name": "Greeting", "type": "record", "fields": [
      {"name": "message", "type": "string"}]},
    {"name": "Curse", "type": "error", "fields": [
      {"name": "message", "type": "string"}]}
  ],

  "messages": {
    "hello": {
      "doc": "Say hello.",
      "request": [{"name": "greeting", "type": "Greeting" }],
      "response": "Greeting",
      "errors": ["Curse"]
    }
  }
}
```

Protocol Wire Format

Message Transport

Messages may be transmitted via different *transport* mechanisms.

To the transport, a *message* is an opaque byte sequence.

A transport is a system that supports:

- **transmission of request messages**
- **receipt of corresponding response messages**

Servers may send a response message back to the client corresponding to a request message. The mechanism of correspondance is transport-specific. For example, in HTTP it is implicit, since HTTP directly supports requests and responses. But a transport that multiplexes many client threads over a single socket would need to tag messages with unique identifiers.

Transports may be either *stateless* or *stateful*. In a stateless transport, messaging assumes no established connection state, while stateful transports establish connections that may be used for multiple messages. This distinction is discussed further in the handshake section below.

HTTP as Transport

When HTTP is used as a transport, each Avro message exchange is an HTTP request/response pair. All messages of an Avro protocol should share a single URL at an HTTP server. Other protocols may also use that URL. Both normal and error Avro response messages should use the 200 (OK) response code. The chunked encoding may be used for requests and responses, but, regardless the Avro request and response are the entire content of an HTTP request and response. The HTTP Content-Type of requests and responses should be specified as "avro/binary". Requests should be made using the POST method.

HTTP is used by Avro as a stateless transport.

Message Framing

Avro messages are *framed* as a list of buffers.

Framing is a layer between messages and the transport. It exists to optimize certain operations.

The format of framed message data is:

- a series of *buffers*, where each buffer consists of:
  - a four-byte, big-endian *buffer length*, followed by
  - that many bytes of *buffer data*.
- A message is always terminated by a zero-length buffer.

Framing is transparent to request and response message formats (described below). Any message may be presented as a single or multiple buffers.

Framing can permit readers to more efficiently get different buffers from different sources and for writers to more efficiently store different buffers to different destinations. In particular, it can reduce the number of times large binary objects are copied. For example, if an RPC parameter consists of a megabyte of file data, that data can be copied directly to a socket from a file descriptor, and, on the other end, it could be written directly to a file descriptor, never entering user space.

A simple, recommended, framing policy is for writers to create a new segment whenever a single binary object is written that is larger than a normal output buffer. Small objects are then appended in buffers, while larger objects are written as their own buffers. When a reader then tries to read a large object the runtime can hand it an entire buffer directly, without having to copy it.

Handshake

The purpose of the handshake is to ensure that the client and the server have each other's protocol definition, so that the client can correctly deserialize responses, and the server can correctly deserialize requests. Both clients and servers should maintain a cache of recently seen protocols, so that, in most cases, a handshake will be completed without extra round-trip network exchanges or the transmission of full protocol text.

RPC requests and responses may not be processed until a handshake has been completed. With a stateless transport, all requests and responses are prefixed by handshakes. With a stateful transport, handshakes are only attached to requests and responses until a successful handshake response has been returned over a connection. After this, request and response payloads are sent without handshakes for the lifetime of that connection.

The handshake process uses the following record schemas:

```
{
  "type": "record",
  "name": "HandshakeRequest", "namespace":"org.apache.avro.ipc",
  "fields": [
    {"name": "clientHash",
     "type": {"type": "fixed", "name": "MD5", "size": 16}},
    {"name": "clientProtocol", "type": ["null", "string"]},
    {"name": "serverHash", "type": "MD5"},
    {"name": "meta", "type": ["null", {"type": "map", "values": "bytes"}]}
  ]
}
{
  "type": "record",
  "name": "HandshakeResponse", "namespace": "org.apache.avro.ipc",
  "fields": [
    {"name": "match",
     "type": {"type": "enum", "name": "HandshakeMatch",
              "symbols": ["BOTH", "CLIENT", "NONE"]}},
    {"name": "serverProtocol",
     "type": ["null", "string"]},
    {"name": "serverHash",
     "type": ["null", {"type": "fixed", "name": "MD5", "size": 16}]},
    {"name": "meta",
     "type": ["null", {"type": "map", "values": "bytes"}]}
  ]
}
```

- A client first prefixes each request with a `HandshakeRequest` containing just the hash of its protocol and of the server's protocol (`clientHash!=null, clientProtocol=null, serverHash!=null`), where the hashes are 128-bit MD5 hashes of the JSON protocol text. If a client has never connected to a given server, it sends its hash as a guess of the server's hash, otherwise it sends the hash that it previously obtained from this server.
- The server responds with a `HandshakeResponse` containing one of:
    - `match=BOTH, serverProtocol=null, serverHash=null` if the client sent the valid hash of the server's protocol and the server knows what protocol corresponds to the client's hash. In this case, the request is complete and the response data immediately follows the HandshakeResponse.
    - `match=CLIENT, serverProtocol!=null, serverHash!=null` if the server has previously seen the client's protocol, but the client sent an incorrect hash of the server's protocol. The request is complete and the response data immediately follows the HandshakeResponse. The client must use the returned protocol to process the response and should also cache that protocol and its hash for future interactions with this server.
    - `match=NONE` if the server has not previously seen the client's protocol. The `serverHash` and `serverProtocol` may also be non-null if the server's protocol hash was incorrect.

      In this case the client must then re-submit its request with its protocol text (`clientHash!=null, clientProtocol!=null, serverHash!=null`) and the server should respond with a successful match (`match=BOTH, serverProtocol=null, serverHash=null`) as above.

The `meta` field is reserved for future handshake enhancements.

Call Format

A *call* consists of a request message paired with its resulting response or error message. Requests and responses contain extensible metadata, and both kinds of messages are framed as described above.

The format of a call request is:

- *request metadata*, a map with values of type `bytes`

- the *message name*, an Avro string, followed by
- the message *parameters*. Parameters are serialized according to the message's request declaration.

When the empty string is used as a message name a server should ignore the parameters and return an empty response. A client may use this to ping a server or to perform a handshake without sending a protocol message.

When a message is declared one-way and a stateful connection has been established by a successful handshake response, no response data is sent. Otherwise the format of the call response is:

- *response metadata*, a map with values of type `bytes`
- a one-byte *error flag* boolean, followed by either:
    - if the error flag is false, the message *response*, serialized per the message's response schema.
    - if the error flag is true, the *error*, serialized per the message's effective error union schema.

Schema Resolution

A reader of Avro data, whether from an RPC or a file, can always parse that data because its schema is provided. But that schema may not be exactly the schema that was expected. For example, if the data was written with a different version of the software than it is read, then records may have had fields added or removed. This section specifies how such schema differences should be resolved.

We call 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 names match
    - both schemas are fixed whose sizes and names match
    - both schemas are records with the same 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, then 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.

Parsing Canonical Form for Schemas

One of the defining characteristics of Avro is that a reader is assumed to have the "same" schema used by the writer of the data the reader is reading. This assumption leads to a data format that's compact and also amenable to many forms of schema evolution. However, the specification so far has not defined what it means for the reader to have the "same" schema as the writer. Does the schema need to be textually identical? Well, clearly adding or removing some whitespace to a JSON expression does not change its meaning. At the same time, reordering the fields of records clearly *does* change the meaning. So what does it mean for a reader to have "the same" schema as a writer?

Parsing Canonical Form is a transformation of a writer's schema that let's us define what it means for two schemas to be "the same" for the purpose of reading data written agains the schema. It is called *Parsing* Canonical Form because the transformations strip away parts of the schema, like "doc" attributes, that are irrelevant to readers trying to parse incoming data. It is called *Canonical Form* because the transformations normalize the JSON text (such as the order of attributes) in a way that eliminates unimportant differences between schemas. If the Parsing Canonical Forms of two different schemas are textually equal, then those schemas are "the same" as far as any reader is concerned, i.e., there is no serialized data that would allow a reader to distinguish data generated by a writer using one of the original schemas from data generated by a writing using the other original schema. (We sketch a proof of this property in a companion document.)

The next subsection specifies the transformations that define Parsing Canonical Form. But with a well-defined canonical form, it can be convenient to go one step further, transforming these canonical forms into simple integers ("fingerprints") that can be used to uniquely identify schemas. The subsection after next recommends some standard practices for generating such fingerprints.

Transforming into Parsing Canonical Form

Assuming an input schema (in JSON form) that's already UTF-8 text for a *valid* Avro schema (including all quotes as required by JSON), the following transformations will produce its Parsing Canonical Form:

- [PRIMITIVES] Convert primitive schemas to their simple form (e.g., `int` instead of `{"type":"int"}`).
- [FULLNAMES] Replace short names with fullnames, using applicable namespaces to do so. Then eliminate `namespace` attributes, which are now redundant.
- [STRIP] Keep only attributes that are relevant to parsing data, which are: `type`, `name`, `fields`, `symbols`, `items`, `values`, `size`. Strip all others (e.g., `doc` and `aliases`).
- [ORDER] Order the appearance of fields of JSON objects as follows: `name`, `type`, `fields`, `symbols`, `items`, `values`, `size`. For example, if an object has `type`, `name`, and `size` fields, then the `name` field should appear first, followed by the `type` and then the `size` fields.

- [STRINGS] For all JSON string literals in the schema text, replace any escaped characters (e.g., \uXXXX escapes) with their UTF-8 equivalents.
- [INTEGERS] Eliminate quotes around and any leading zeros in front of JSON integer literals (which appear in the `size` attributes of `fixed` schemas).
- [WHITESPACE] Eliminate all whitespace in JSON outside of string literals.

Schema Fingerprints

"[A] fingerprinting algorithm is a procedure that maps an arbitrarily large data item (such as a computer file) to a much shorter bit string, its *fingerprint,* that uniquely identifies the original data for all practical purposes" (quoted from [Wikipedia]). In the Avro context, fingerprints of Parsing Canonical Form can be useful in a number of applications; for example, to cache encoder and decoder objects, to tag data items with a short substitute for the writer's full schema, and to quickly negotiate common-case schemas between readers and writers.

In designing fingerprinting algorithms, there is a fundamental trade-off between the length of the fingerprint and the probability of collisions. To help application designers find appropriate points within this trade-off space, while encouraging interoperability and ease of implementation, we recommend using one of the following three algorithms when fingerprinting Avro schemas:

- When applications can tolerate longer fingerprints, we recommend using the SHA-256 digest algorithm to generate 256-bit fingerprints of Parsing Canonical Forms. Most languages today have SHA-256 implementations in their libraries.
- At the opposite extreme, the smallest fingerprint we recommend is a 64-bit Rabin fingerprint. Below, we provide pseudo-code for this algorithm that can be easily translated into any programming language. 64-bit fingerprints should guarantee uniqueness for schema caches of up to a million entries (for such a cache, the chance of a collision is 3E-8). We don't recommend shorter fingerprints, as the chances of collisions is too great (for example, with 32-bit fingerprints, a cache with as few as 100,000 schemas has a 50% chance of having a collision).
- Between these two extremes, we recommend using the MD5 message digest to generate 128-bit fingerprints. These make sense only where very large numbers of schemas are being manipulated (tens of millions); otherwise, 64-bit fingerprints should be sufficient. As with SHA-256, MD5 implementations are found in most libraries today.

These fingerprints are *not* meant to provide any security guarantees, even the longer SHA-256-based ones. Most Avro applications should be surrounded by security measures that prevent attackers from writing random data and otherwise interfering with the consumers of schemas. We recommend that these surrounding mechanisms be used to prevent collision and pre-image attacks (i.e., "forgery") on schema fingerprints, rather than relying on the security properties of the fingerprints themselves.

Rabin fingerprints are cyclic redundancy checks computed using irreducible polynomials. In the style of the Appendix of RFC 1952 (pg 10), which defines the CRC-32 algorithm, here's our definition of the 64-bit AVRO fingerprinting algorithm:

```
long fingerprint64(byte[] buf) {
  if (FP_TABLE == null) initFPTable();
  long fp = EMPTY;
  for (int i = 0; i < buf.length; i++)
    fp = (fp >>> 8) ^ FP_TABLE[(int)(fp ^ buf[i]) & 0xff];
  return fp;
}

static long EMPTY = 0xc15d213aa4d7a795L;
static long[] FP_TABLE = null;

void initFPTable() {
  FP_TABLE = new long[256];
  for (int i = 0; i < 256; i++) {
    long fp = i;
    for (int j = 0; j < 8; j++)
```

```
      fp = (fp >>> 1) ^ (EMPTY & -(fp & 1L));
    FP_TABLE[i] = fp;
  }
}
```

Readers interested in the mathematics behind this algorithm may want to read this book chapter. (Unlike RFC-1952 and the book chapter, we prepend a single one bit to messages. We do this because CRCs ignore leading zero bits, which can be problematic. Our code prepends a one-bit by initializing fingerprints using `EMPTY`, rather than initializing using zero as in RFC-1952 and the book chapter.)

Logical Types

A logical type is an Avro primitive or complex type with extra attributes to represent a derived type. The attribute `logicalType` must always be present for a logical type, and is a string with the name of one of the logical types listed later in this section. Other attributes may be defined for particular logical types.

A logical type is always serialized using its underlying Avro type so that values are encoded in exactly the same way as the equivalent Avro type that does not have a `logicalType` attribute. Language implementations may choose to represent logical types with an appropriate native type, although this is not required.

Language implementations must ignore unknown logical types when reading, and should use the underlying Avro type. If a logical type is invalid, for example a decimal with scale greater than its precision, then implementations should ignore the logical type and use the underlying Avro type.

Decimal

The `decimal` logical type represents an arbitrary-precision signed decimal number of the form *unscaled* $\times$ 10^{-scale}.

A `decimal` logical type annotates Avro `bytes` or `fixed` types. The byte array must contain the two's-complement representation of the unscaled integer value in big-endian byte order. The scale is fixed, and is specified using an attribute.

The following attributes are supported:

- `scale`, a JSON integer representing the scale (optional). If not specified the scale is 0.
- `precision`, a JSON integer representing the (maximum) precision of decimals stored in this type (required).

For example, the following schema represents decimal numbers with a maximum precision of 4 and a scale of 2:

```
{
  "type": "bytes",
  "logicalType": "decimal",
  "precision": 4,
  "scale": 2
}
```

Precision must be a positive integer greater than zero. If the underlying type is a `fixed`, then the precision is limited by its size. An array of length `n` can store at most *floor(log_10($2^{8 \times n - 1}$ - 1))* base-10 digits of precision.

Scale must be zero or a positive integer less than or equal to the precision.

For the purposes of schema resolution, two schemas that are `decimal` logical types *match* if their scales and precisions match.

Date

The `date` logical type represents a date within the calendar, with no reference to a particular time zone or time of day.

A `date` logical type annotates an Avro `int`, where the int stores the number of days from the unix epoch, 1 January 1970 (ISO calendar).

Time (millisecond precision)

The `time-millis` logical type represents a time of day, with no reference to a particular calendar, time zone or date, with a precision of one millisecond.

A `time-millis` logical type annotates an Avro `int`, where the int stores the number of milliseconds after midnight, 00:00:00.000.

Time (microsecond precision)

The `time-micros` logical type represents a time of day, with no reference to a particular calendar, time zone or date, with a precision of one microsecond.

A `time-micros` logical type annotates an Avro `long`, where the long stores the number of microseconds after midnight, 00:00:00.000000.

Timestamp (millisecond precision)

The `timestamp-millis` logical type represents an instant on the global timeline, independent of a particular time zone or calendar, with a precision of one millisecond.

A `timestamp-millis` logical type annotates an Avro `long`, where the long stores the number of milliseconds from the unix epoch, 1 January 1970 00:00:00.000 UTC.

Timestamp (microsecond precision)

The `timestamp-micros` logical type represents an instant on the global timeline, independent of a particular time zone or calendar, with a precision of one microsecond.

A `timestamp-micros` logical type annotates an Avro `long`, where the long stores the number of microseconds from the unix epoch, 1 January 1970 00:00:00.000000 UTC.

Duration

The `duration` logical type represents an amount of time defined by a number of months, days and milliseconds. This is not equivalent to a number of milliseconds, because, depending on the moment in time from which the duration is measured, the number of days in the month and number of milliseconds in a day may differ. Other standard periods such as years, quarters, hours and minutes can be expressed through these basic periods.

A `duration` logical type annotates Avro `fixed` type of size 12, which stores three little-endian unsigned integers that represent durations at different granularities of time. The first stores a number in months, the second stores a number in days, and the third stores a number in milliseconds.

Last Published: 08/02/2017 05:13:20