Avro IDL 1.4.1

Table of contents

1. Introduction

This document defines Avro IDL, an experimental higher-level language for authoring Avro schemata. Before reading this document, you should have familiarity with the concepts of schemata and protocols, as well as the various primitive and complex types available in Avro.

N.B. This feature is considered experimental in the current version of Avro and the language has not been finalized. Although major changes are unlikely, some syntax may change in future versions of Avro.

2. Overview

2.1. Purpose

The aim of the Avro IDL language is to enable developers to author schemata in a way that feels more similar to common programming languages like Java, C++, or Python. Additionally, the Avro IDL language may feel more familiar for those users who have previously used the interface description languages (IDLs) in other frameworks like Thrift, Protocol Buffers, or CORBA.

2.2. Usage

Each Avro IDL file defines a single Avro Protocol, and thus generates as its output a JSON-format Avro Protocol file with extension `.avpr`.

To convert a `.avdl` file into a `.avpr` file, it must be processed by the `idl` tool. For example:

```
$ java -jar avroj-tools-1.4.0.jar idl src/test/idl/input/namespaces.avdl
/tmp/namespaces.avpr
$ head /tmp/namespaces.avpr
{
  "protocol" : "TestNamespace",
  "namespace" : "avro.test.protocol",
```

The `idl` tool can also process input to and from *stdin* and *stdout*. See `idl --help` for full usage information.

3. Defining a Protocol in Avro IDL

An Avro IDL file consists of exactly one protocol definition. The minimal protocol is defined

by the following code:

```
protocol MyProtocol {
}
```

This is equivalent to (and generates) the following JSON protocol definition:

```
{
"protocol" : "MyProtocol",
  "types" : [ ],
  "messages" : {
  }
}
```

The namespace of the protocol may be changed using the `@namespace` annotation:

```
@namespace("mynamespace")
protocol MyProtocol {
}
```

This notation is used throughout Avro IDL as a way of specifying properties for the annotated element, as will be described later in this document.

Protocols in Avro IDL can contain the following items:

- Imports of external protocol and schema files.
- Definitions of named schemata, including *record*s, *error*s, *enum*s, and *fixed*s.
- Definitions of RPC messages

4. Imports

Files may be imported in one of three formats:

- An IDL file may be imported with a statement like:
  ```
  import idl "foo.avdl";
  ```
- A JSON protocol file may be imported with a statement like:
  ```
  import protocol "foo.avpr";
  ```
- A JSON schema file may be imported with a statement like:
  ```
  import schema "foo.avsc";
  ```

Messages and types in the imported file are added to this file's protocol.

Imported file names are resolved relative to the current IDL file.

5. Defining an Enumeration

Enums are defined in Avro IDL using a syntax similar to C or Java:

```
enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
}
```

Note that, unlike the JSON format, anonymous enums cannot be defined.

6. Defining a Fixed Length Field

Fixed fields are defined using the following syntax:

```
fixed MD5(16);
```

This example defines a fixed-length type called `MD5` which contains 16 bytes.

7. Defining Records and Errors

Records are defined in Avro IDL using a syntax similar to a `struct` definition in C:

```
record Employee {
  string name;
  boolean active = true;
  long salary;
}
```

The above example defines a record with the name “Employee” with three fields.

To define an error, simply use the keyword `error` instead of `record`. For example:

```
error Kaboom {
  string explanation;
  int result_code = -1;
}
```

Each field in a record or error consists of a type and a name, optional property annotations and an optional default value.

A type reference in Avro IDL must be one of:

- A primitive type
- A named schema defined prior to this usage in the same Protocol
- A complex type (array, map, or union)

7.1. Primitive Types

The primitive types supported by Avro IDL are the same as those supported by Avro's JSON format. This list includes `int`, `long`, `string`, `boolean`, `float`, `double`, `null`, and `bytes`.

7.2. References to Named Schemata

If a named schema has already been defined in the same Avro IDL file, it may be referenced by name as if it were a primitive type:

```
record Card {
  Suit suit; // refers to the enum Card defined above
  int number;
}
```

7.3. Default Values

Default values for fields may be optionally specified by using an equals sign after the field name followed by a JSON expression indicating the default value. This JSON is interpreted as described in the spec.

7.4. Complex Types

7.4.1. Arrays

Array types are written in a manner that will seem familiar to C++ or Java programmers. An array of any type `t` is denoted `array<t>`. For example, an array of strings is denoted `array<string>`, and a multidimensional array of `Foo` records would be `array<array<Foo>>`.

7.4.2. Maps

Map types are written similarly to array types. An array that contains values of type `t` is written `map<t>`. As in the JSON schema format, all maps contain `string`-type keys.

7.4.3. Unions

Union types are denoted as `union { typeA, typeB, typeC, ... }`. For example, this record contains a string field that is optional (unioned with `null`):

```
record RecordWithUnion {
  union { null, string } optionalString;
}
```

Note that the same restrictions apply to Avro IDL unions as apply to unions defined in the JSON format; namely, a record may not contain multiple elements of the same type.

8. Defining RPC Messages

The syntax to define an RPC message within a Avro IDL protocol is similar to the syntax for a method declaration within a C header file or a Java interface. To define an RPC message `add` which takes two arguments named `foo` and `bar`, returning an `int`, simply include the following definition within the protocol:

```
int add(int foo, int bar = 0);
```

Message arguments, like record fields, may specify default values.

To define a message with no response, you may use the alias `void`, equivalent to the Avro `null` type:

```
void logMessage(string message);
```

If you have previously defined an error type within the same protocol, you may declare that a message can throw this error using the syntax:

```
void goKaboom() throws Kaboom;
```

To define a one-way message, use the keyword `oneway` after the parameter list, for example:

```
void fireAndForget(string message) oneway;
```

9. Other Language Features

9.1. Comments

All Java-style comments are supported within a Avro IDL file. Any text following `//` on a line is ignored, as is any text between `/*` and `*/`, possibly spanning multiple lines.

9.2. Escaping Identifiers

Occasionally, one will need to use a reserved language keyword as an identifier. In order to do so, backticks (`) may be used to escape the identifier. For example, to define a message with the literal name *error*, you may write:

```
void `error`();
```

This syntax is allowed anywhere an identifier is expected.

9.3. Annotations for Ordering and Namespaces

Java-style annotations may be used to add additional properties to types and fields throughout Avro IDL.

For example, to specify the sort order of a field within a record, one may use the `@order` annotation before the field name as follows:

```
record MyRecord {
  string @order("ascending") myAscendingSortField;
  string @order("descending")  myDescendingField;
  string @order("ignore") myIgnoredField;
}
```

A field's type may also be preceded by annotations, e.g.:

```
record MyRecord {
  @java-class("java.util.ArrayList") array string myStrings;
}
```

Similarly, a `@namespace` annotation may be used to modify the namespace when defining a named schema. For example:

```
@namespace("org.apache.avro.firstNamespace")
protocol MyProto {
  @namespace("org.apache.avro.someOtherNamespace")
  record Foo {}

  record Bar {}
}
```

will define a protocol in the `firstNamespace` namespace. The record `Foo` will be defined in `someOtherNamespace` and `Bar` will be defined in `firstNamespace` as it inherits its default from its container.

Type and field aliases are specified with the `@aliases` annotation as follows:

```
@aliases(["org.old.OldRecord", "org.ancient.AncientRecord"])
record MyRecord {
  string @aliases(["oldField", "ancientField"]) myNewField;
}
```

10. Complete Example

The following is a complete example of a Avro IDL file that shows most of the above features:

```
/**
 * An example protocol in Avro IDL
 */
@namespace("org.apache.avro.test")
protocol Simple {

  @aliases(["org.foo.KindOf"])
  enum Kind {
    FOO,
    BAR, // the bar enum value
    BAZ
  }

  fixed MD5(16);

  record TestRecord {
    @order("ignore")
    string name;

    @order("descending")
    Kind kind;

    MD5 hash;

    union { MD5, null} @aliases(["hash"]) nullableHash;

    array<long> arrayOfLongs;
  }

  error TestError {
    string message;
  }

  string hello(string greeting);
  TestRecord echo(TestRecord `record`);
  int add(int arg1, int arg2);
  bytes echoBytes(bytes data);
  void `error`() throws TestError;
  void ping() oneway;
}
```

Additional examples may be found in the Avro source tree under the `src/test/idl/input` directory.