Apache Avro# 1.10.0 Getting Started (Java)

Table of contents

This is a short guide for getting started with Apache Avro# using Java. This guide only covers using Avro for data serialization; see Patrick Hunt's Avro RPC Quick Start for a good introduction to using Avro for RPC.

1 Download

Avro implementations for C, C++, C#, Java, PHP, Python, and Ruby can be downloaded from the Apache Avro# Releases page. This guide uses Avro 1.10.0, the latest version at the time of writing. For the examples in this guide, download *avro-1.10.0.jar* and *avro-tools-1.10.0.jar*.

Alternatively, if you are using Maven, add the following dependency to your POM:

```
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro</artifactId>
  <version>1.10.0</version>
</dependency>
```

As well as the Avro Maven plugin (for performing code generation):

```
<plugin>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro-maven-plugin</artifactId>
  <version>1.10.0</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>schema</goal>
      </goals>
      <configuration>
        <sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>
        <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.8</source>
    <target>1.8</target>
  </configuration>
</plugin>
```

You may also build the required Avro jars from source. Building Avro is beyond the scope of this guide; see the Build Documentation page in the wiki for more information.

2 Defining a schema

Avro schemas are defined using JSON. Schemas are composed of primitive types (`null`, `boolean`, `int`, `long`, `float`, `double`, `bytes`, and `string`) and complex types (`record`, `enum`, `array`, `map`, `union`, and `fixed`). You can learn more about Avro schemas and types from the specification, but for now let's start with a simple schema example, *user.avsc*:

```
{"namespace": "example.avro",
 "type": "record",
 "name": "User",
 "fields": [
     {"name": "name", "type": "string"},
     {"name": "favorite_number",  "type": ["int", "null"]},
     {"name": "favorite_color", "type": ["string", "null"]}
 ]
}
```

This schema defines a record representing a hypothetical user. (Note that a schema file can only contain a single schema definition.) At minimum, a record definition must include its type (`"type": "record"`), a name (`"name": "User"`), and fields, in this case `name`, `favorite_number`, and `favorite_color`. We also define a namespace (`"namespace": "example.avro"`), which together with the name attribute defines the "full name" of the schema (`example.avro.User` in this case).

Fields are defined via an array of objects, each of which defines a name and type (other attributes are optional, see the record specification for more details). The type attribute of a field is another schema object, which can be either a primitive or complex type. For example, the `name` field of our User schema is the primitive type `string`, whereas the `favorite_number` and `favorite_color` fields are both `union`s, represented by JSON arrays. `union`s are a complex type that can be any of the types listed in the array; e.g., `favorite_number` can either be an `int` or `null`, essentially making it an optional field.

3 Serializing and deserializing with code generation

3.1 Compiling the schema

Code generation allows us to automatically create classes based on our previously-defined schema. Once we have defined the relevant classes, there is no need to use the schema directly in our programs. We use the avro-tools jar to generate code as follows:

```
java -jar /path/to/avro-tools-1.10.0.jar compile schema <schema file> <destination>
```

This will generate the appropriate source files in a package based on the schema's namespace in the provided destination folder. For instance, to generate a `User` class in package `example.avro` from the schema defined above, run

```
java -jar /path/to/avro-tools-1.10.0.jar compile schema user.avsc .
```

Note that if you using the Avro Maven plugin, there is no need to manually invoke the schema compiler; the plugin automatically performs code generation on any .avsc files present in the configured source directory.

3.2 Creating Users

Now that we've completed the code generation, let's create some `User`s, serialize them to a data file on disk, and then read back the file and deserialize the `User` objects.

First let's create some `User`s and set their fields.

```
User user1 = new User();
user1.setName("Alyssa");
user1.setFavoriteNumber(256);
// Leave favorite color null

// Alternate constructor
User user2 = new User("Ben", 7, "red");

// Construct via builder
User user3 = User.newBuilder()
             .setName("Charlie")
             .setFavoriteColor("blue")
             .setFavoriteNumber(null)
             .build();
```

As shown in this example, Avro objects can be created either by invoking a constructor directly or by using a builder. Unlike constructors, builders will automatically set any default values specified in the schema. Additionally, builders validate the data as it set, whereas objects constructed directly will not cause an error until the object is serialized. However, using constructors directly generally offers better performance, as builders create a copy of the datastructure before it is written.

Note that we do not set `user1`'s favorite color. Since that record is of type `["string", "null"]`, we can either set it to a `string` or leave it `null`; it is essentially optional. Similarly, we set `user3`'s favorite number to null (using a builder requires setting all fields, even if they are null).

3.3 Serializing

Now let's serialize our `User`s to disk.

```
// Serialize user1, user2 and user3 to disk
DatumWriter<User> userDatumWriter = new SpecificDatumWriter<User>(User.class);
DataFileWriter<User> dataFileWriter = new DataFileWriter<User>(userDatumWriter);
dataFileWriter.create(user1.getSchema(), new File("users.avro"));
dataFileWriter.append(user1);
dataFileWriter.append(user2);
dataFileWriter.append(user3);
dataFileWriter.close();
```

We create a `DatumWriter`, which converts Java objects into an in-memory serialized format. The `SpecificDatumWriter` class is used with generated classes and extracts the schema from the specified generated type.

Next we create a `DataFileWriter`, which writes the serialized records, as well as the schema, to the file specified in the `dataFileWriter.create` call. We write our users to the file via calls to the `dataFileWriter.append` method. When we are done writing, we close the data file.

3.4 Deserializing

Finally, let's deserialize the data file we just created.

```
// Deserialize Users from disk
DatumReader<User> userDatumReader = new SpecificDatumReader<User>(User.class);
DataFileReader<User> dataFileReader = new DataFileReader<User>(file, userDatumReader);
User user = null;
while (dataFileReader.hasNext()) {
// Reuse user object by passing it to next(). This saves us from
// allocating and garbage collecting many objects for files with
// many items.
user = dataFileReader.next(user);
System.out.println(user);
}
```

This snippet will output:

```
{"name": "Alyssa", "favorite_number": 256, "favorite_color": null}
{"name": "Ben", "favorite_number": 7, "favorite_color": "red"}
{"name": "Charlie", "favorite_number": null, "favorite_color": "blue"}
```

Deserializing is very similar to serializing. We create a `SpecificDatumReader`, analogous to the `SpecificDatumWriter` we used in serialization, which converts in-memory serialized items into instances of our generated class, in this case `User`. We pass

the `DatumReader` and the previously created `File` to a `DataFileReader`, analogous to the `DataFileWriter`, which reads both the schema used by the writer as well as the data from the file on disk. The data will be read using the writer's schema included in the file and the schema provided by the reader, in this case the `User` class. The writer's schema is needed to know the order in which fields were written, while the reader's schema is needed to know what fields are expected and how to fill in default values for fields added since the file was written. If there are differences between the two schemas, they are resolved according to the Schema Resolution specification.

Next we use the `DataFileReader` to iterate through the serialized `User`s and print the deserialized object to stdout. Note how we perform the iteration: we create a single `User` object which we store the current deserialized user in, and pass this record object to every call of `dataFileReader.next`. This is a performance optimization that allows the `DataFileReader` to reuse the same `User` object rather than allocating a new `User` for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use `for (User user : dataFileReader)` if performance is not a concern.

3.5 Compiling and running the example code

This example code is included as a Maven project in the *examples/java-example* directory in the Avro docs. From this directory, execute the following commands to build and run the example:

```
$ mvn compile # includes code generation via Avro Maven plugin
$ mvn -q exec:java -Dexec.mainClass=example.SpecificMain
```

3.6 Beta feature: Generating faster code

In this release we have introduced a new approach to generating code that speeds up decoding of objects by more than 10% and encoding by more than 30% (future performance enhancements are underway). To ensure a smooth introduction of this change into production systems, this feature is controlled by a feature flag, the system property `org.apache.avro.specific.use_custom_coders`. In this first release, this feature is off by default. To turn it on, set the system flag to `true` at runtime. In the sample above, for example, you could enable the fater coders as follows:

```
$ mvn -q exec:java -Dexec.mainClass=example.SpecificMain \
    -Dorg.apache.avro.specific.use_custom_coders=true
```

Note that you do *not* have to recompile your Avro schema to have access to this feature. The feature is compiled and built into your code, and you turn it on and off at runtime using the feature flag. As a result, you can turn it on during testing, for example, and then off in production. Or you can turn it on in production, and quickly turn it off if something breaks.

We encourage the Avro community to exercise this new feature early to help build confidence. (For those paying one-demand for compute resources in the cloud, it can lead to meaningful cost savings.) As confidence builds, we will turn this feature on by default, and eventually eliminate the feature flag (and the old code).

4 Serializing and deserializing without code generation

Data in Avro is always stored with its corresponding schema, meaning we can always read a serialized item regardless of whether we know the schema ahead of time. This allows us to perform serialization and deserialization without code generation.

Let's go over the same example as in the previous section, but without using code generation: we'll create some users, serialize them to a data file on disk, and then read back the file and deserialize the users objects.

4.1 Creating users

First, we use a `Parser` to read our schema definition and create a `Schema` object.

```
Schema schema = new Schema.Parser().parse(new File("user.avsc"));
```

Using this schema, let's create some users.

```
GenericRecord user1 = new GenericData.Record(schema);
user1.put("name", "Alyssa");
user1.put("favorite_number", 256);
// Leave favorite color null

GenericRecord user2 = new GenericData.Record(schema);
user2.put("name", "Ben");
user2.put("favorite_number", 7);
user2.put("favorite_color", "red");
```

Since we're not using code generation, we use `GenericRecord`s to represent users. `GenericRecord` uses the schema to verify that we only specify valid fields. If we try to set a non-existent field (e.g., `user1.put("favorite_animal", "cat")`), we'll get an `AvroRuntimeException` when we run the program.

Note that we do not set `user1`'s favorite color. Since that record is of type `["string", "null"]`, we can either set it to a `string` or leave it `null`; it is essentially optional.

4.2 Serializing

Now that we've created our user objects, serializing and deserializing them is almost identical to the example above which uses code generation. The main difference is that we use generic instead of specific readers and writers.

First we'll serialize our users to a data file on disk.

```
// Serialize user1 and user2 to disk
File file = new File("users.avro");
DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<GenericRecord>(schema);
DataFileWriter<GenericRecord> dataFileWriter = new
 DataFileWriter<GenericRecord>(datumWriter);
dataFileWriter.create(schema, file);
dataFileWriter.append(user1);
dataFileWriter.append(user2);
dataFileWriter.close();
```

We create a `DatumWriter`, which converts Java objects into an in-memory serialized format. Since we are not using code generation, we create a `GenericDatumWriter`. It requires the schema both to determine how to write the `GenericRecord`s and to verify that all non-nullable fields are present.

As in the code generation example, we also create a `DataFileWriter`, which writes the serialized records, as well as the schema, to the file specified in the `dataFileWriter.create` call. We write our users to the file via calls to the `dataFileWriter.append` method. When we are done writing, we close the data file.

4.3 Deserializing

Finally, we'll deserialize the data file we just created.

```
// Deserialize users from disk
DatumReader<GenericRecord> datumReader = new GenericDatumReader<GenericRecord>(schema);
DataFileReader<GenericRecord> dataFileReader = new DataFileReader<GenericRecord>(file,
 datumReader);
GenericRecord user = null;
while (dataFileReader.hasNext()) {
// Reuse user object by passing it to next(). This saves us from
// allocating and garbage collecting many objects for files with
// many items.
user = dataFileReader.next(user);
System.out.println(user);
```

This outputs:

```
{"name": "Alyssa", "favorite_number": 256, "favorite_color": null}
{"name": "Ben", "favorite_number": 7, "favorite_color": "red"}
```

Deserializing is very similar to serializing. We create a `GenericDatumReader`, analogous to the `GenericDatumWriter` we used in serialization, which converts in-memory serialized items into `GenericRecords`. We pass the `DatumReader` and the previously created `File` to a `DataFileReader`, analogous to the `DataFileWriter`, which reads both the schema used by the writer as well as the data from the file on disk. The data will be read using the writer's schema included in the file, and the reader's schema provided to the `GenericDatumReader`. The writer's schema is needed to know the order in which fields were written, while the reader's schema is needed to know what fields are expected and how to fill in default values for fields added since the file was written. If there are differences between the two schemas, they are resolved according to the Schema Resolution specification.

Next, we use the `DataFileReader` to iterate through the serialized users and print the deserialized object to stdout. Note how we perform the iteration: we create a single `GenericRecord` object which we store the current deserialized user in, and pass this record object to every call of `dataFileReader.next`. This is a performance optimization that allows the `DataFileReader` to reuse the same record object rather than allocating a new `GenericRecord` for every iteration, which can be very expensive in terms of object allocation and garbage collection if we deserialize a large data file. While this technique is the standard way to iterate through a data file, it's also possible to use `for (GenericRecord user : dataFileReader)` if performance is not a concern.

4.4 Compiling and running the example code

This example code is included as a Maven project in the *examples/java-example* directory in the Avro docs. From this directory, execute the following commands to build and run the example:

```
$ mvn compile
$ mvn -q exec:java -Dexec.mainClass=example.GenericMain
```