



Last Published: 09/24/2015 05:05:05Last Published: 07/24/2014 19:22:19



PDF

# Apache Avro™ 1.7.7 Getting Started (Java)

- [Download](#)
- [Defining a schema](#)
- [Serializing and deserializing with code generation](#)
  - [Compiling the schema](#)
  - [Creating Users](#)
  - [Serializing](#)
  - [Deserializing](#)
  - [Compiling and running the example code](#)
- [Serializing and deserializing without code generation](#)
  - [Creating users](#)
  - [Serializing](#)
  - [Deserializing](#)
  - [Compiling and running the example code](#)

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.

## 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 1.7.7, the latest version at the time of writing. For the examples in this guide, download *avro-1.7.7.jar* and *avro-tools-1.7.7.jar*. The Avro Java implementation also depends on the [Jackson](#) JSON library. From the Jackson [download page](#), download the core-asl and mapper-asl jars. Add *avro-1.7.7.jar* and the Jackson jars to your project's classpath (avro-tools will be used for code generation).

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

```
<dependency>
  <groupId>org.apache.avro</groupId>
  <artifactId>avro</artifactId>
  <version>1.7.7</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>1.7.7</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>1.6</source>
  <target>1.6</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.

## 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 unions, represented by JSON arrays. unions 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.

## Serializing and deserializing with code generation

### 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-1.7.7.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-1.7.7.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.

### Creating Users

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

First let's create some `Users` 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`).

## Serializing

Now let's serialize our `Users` 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.

## 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 the data file on disk.

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 `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.

## 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
```

## 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.

### 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 `GenericRecords` 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.

### 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 `GenericRecords` 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.

### 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 the data file on disk.

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.

### 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
```