avro4s

Avro4s is a schema/class generation and serializing/deserializing library for Avro written in Scala. The objective is to allow seamless use with Scala without the need to to write boilerplate conversions yourself, and without the runtime overhead of reflection. Hence, this is a macro based library and generates code for use with Avro at *compile time*.

The features of the library are:

- Schema generation from classes at compile time
- Class generation from schemas at build time
- Boilerplate free serialization of classes to Avro
- Boilerplate free deserialization of Avro to classes

Changelog

- 1.6.1 - Fixed bug when using Option with a default value of None. Updated sbt generator. Added better way to discrimate between tpyes (Ilya Epifanov). Optimized reflection lookup @simonsouter.
- 1.6.0 - Added support for Coproducts (see section on coproducts) @SohumB. Added support for value classes. Bumped 2.12 cross build to M4.
- 1.5.6 - Added support for setting scale and precision on BigDecimal @blbradley
- 1.5.5 - Skipped
- 1.5.4 - Added support for recursive types @SohumB
- 1.5.3 - Added support for Option[X] in default parameters @jecos
- 1.5.2 - Added support for ints, longs, booleans, doubles, arrays, enums and maps in default parameters @whazenberg
- 1.5.1 - Fixed macro bug introduced in 1.5.0 which broke macro generation.
- 1.5.0 - Upgraded to Avro 1.8.1. Deprecated factory methods in io classes in favour of more explicit naming. Added caching for the macro generation (thanks @jtvoorde).
- 1.4.3 - Added support for json serialization
- 1.4.1 - Added support for parameter defaults
- 1.4.0 - Added better support for enums. Added support for UUIDs. Rewrote the macro backend. Added better binary support.
- 1.3.3 - Added missing support for deserializing byte arrays
- 1.3.0 - Added support for Scala 2.12. Removed 2.10 cross build. Fixed issues with private vals. Added binary (no schema) output stream. Exposed RecordFormat[T] typeclass to enable easy conversion of T to/from an Avro Record.
- 1.2.0 - Added support for properties, doc fields, and aliases. These are set via annotations.
- 1.1.0 - Added JSON document to Avro schema converter
- 1.0.0 - Migrated all macros to use Shapeless. Fixed some trickier nested case class issues. Simplified API. Added support for java enums.

- 0.94.0 - Added support for writing/reading Either and Option in serializer/deserializer. Fixed bug with array serialization.
- 0.93.0 - Added support for either and options in schema generator. Added support for aliases via scala annotation.
- 0.92.0 - Added support for unions (and unions of nulls to Options) and enums to class generator.

Schemas

Avro4s allows us to generate schemas directly from classes in a totally straightforward way. Let's define some classes.

```
case class Ingredient(name: String, sugar: Double, fat: Double)
case class Pizza(name: String, ingredients: Seq[Ingredient], vegetarian: Boolean, vegan: Boolean, calories:
```

Next is to invoke the `apply` method of `AvroSchema` passing in the top level type. This will return an `org.apache.avro.Schema` instance, from which you can output, write to a file etc.

```
import com.sksamuel.avro4s.AvroSchema
val schema = AvroSchema[Pizza]
```

Which will output the following schema:

```
{
  "type" : "record",
  "name" : "Pizza",
  "namespace" : "com.sksamuel.avro4s.json",
  "fields" : [ {
    "name" : "name",
    "type" : "string"
  }, {
    "name" : "ingredients",
    "type" : {
      "type" : "array",
      "items" : {
        "type" : "record",
        "name" : "Ingredient",
        "fields" : [ {
          "name" : "name",
          "type" : "string"
        }, {
          "name" : "sugar",
          "type" : "double"
        }, {
          "name" : "fat",
          "type" : "double"
        } ]
      }
    }
  }, {
    "name" : "vegetarian",
    "type" : "boolean"
  }, {
    "name" : "vegan",
    "type" : "boolean"
  }, {
    "name" : "calories",
    "type" : "int"
  } ]
}
```

You can see that the schema generator handles nested case classes, sequences, primitives, etc. For a full list of supported object types, see the table later.

Recursive Schemas

Avro4s supports recursive schemas, but you will have to manually force the `SchemaFor` instance, instead of letting it be generated.

```
case class Recursive(payload: Int, next: Option[Recursive])

implicit val schemaFor = SchemaFor[Recursive]
val schema = AvroSchema[Recursive]
```

Input / Output

Serializing

Avro4s allows us to easily serialize case classes using an instance of `AvroOutputStream` which we write to, and close, just like you would any regular output stream. An `AvroOutputStream` can be created from a `File`, `Path`, or by wrapping another `OutputStream`. When we create one, we specify the type of objects that we will be serializing. Eg, to serialize instances of our Pizza class:

```
import java.io.File
import com.sksamuel.avro4s.AvroOutputStream

val pepperoni = Pizza("pepperoni", Seq(Ingredient("pepperoni", 12, 4.4), Ingredient("onions", 1, 0.4)), fal
val hawaiian = Pizza("hawaiian", Seq(Ingredient("ham", 1.5, 5.6), Ingredient("pineapple", 5.2, 0.2)), false

val os = AvroOutputStream.data[Pizza](new File("pizzas.avro"))
os.write(Seq(pepperoni, hawaiian))
os.flush()
os.close()
```

Deserializing

With avro4s we can easily deserialize a file back into Scala case classes. Given the `pizzas.avro` file we generated in the previous section on serialization, we will read this back in using the `AvroInputStream` class. We first create an instance of the input stream specifying the types we will read back, and the file. Then we call iterator which will return a lazy iterator (reads on demand) of the data in the file. In this example, we'll load all data at once from the iterator via `toSet`.

```
import com.sksamuel.avro4s.AvroInputStream

val is = AvroInputStream.data[Pizza](new File("pizzas.avro"))
val pizzas = is.iterator.toSet
is.close()
println(pizzas.mkString("\n"))
```

Will print out:

```
Pizza(pepperoni,List(Ingredient(pepperoni,12.2,4.4), Ingredient(onions,1.2,0.4)),false,false,500) Pizza(hav
```

Binary Serializing

You can serialize to Binary using the `AvroBinaryOutputStream` which you can create directly or by using `AvroOutputStream.binary`. The difference with binary serialization is that the schema is not included in the output.

Simply

```
case class Composer(name: String, birthplace: String, compositions: Seq[String])
val ennio = Composer("ennio morricone", "rome", Seq("legend of 1900", "ecstasy of gold"))

val baos = new ByteArrayOutputStream()
val output = AvroOutputStream.binary[Composer](baos)
output.write(ennio)
output.close()
val bytes = baos.toByteArray
```

Binary Deserializing

You can deserialize Binary using the `AvroBinaryInputStream` which you can create directly or by using `AvroInputStream.binary` The difference with binary serialization is that the schema is not included in the data.

```
val in = new ByteArrayInputStream(bytes)
val input = AvroInputStream.binary[Composer](in)
val result = input.iterator.toSeq
result shouldBe Vector(ennio)
```

JSON Serializing

You can serialize to JSON using the `AvroJsonOutputStream` which you can create directly or by using `AvroOutputStream.json`

Simply

```
case class Composer(name: String, birthplace: String, compositions: Seq[String])
val ennio = Composer("ennio morricone", "rome", Seq("legend of 1900", "ecstasy of gold"))

val baos = new ByteArrayOutputStream()
val output = AvroOutputStream.json[Composer](baos)
output.write(ennio)
output.close()
println(baos.toString("UTF-8"))
```

JSON Deserializing

You can deserialize JSON using the `AvroJsonInputStream` which you can create directly or by using `AvroInputStream.json`

```
val json = "{\"name\":\"ennio morricone\",\"birthplace\":\"rome\",\"compositions\":[\"legend of 1900\",\"ec
val in = new ByteInputStream(json.getBytes("UTF-8"), json.size)
val input = AvroInputStream.json[Composer](in)
val result = input.singleEntity
result shouldBe Success(ennio)
```

Conversions to/from GenericRecord

To interface with the Java API it is sometimes desirable to convert between your classes and the Avro `GenericRecord` type. You can do this easily in avro4s using the `RecordFormat` typeclass (this is what the input/output streams use behind the scenes). Eg,

To convert from a class into a record:

```
case class Composer(name: String, birthplace: String, compositions: Seq[String])
val ennio = Composer("ennio morricone", "rome", Seq("legend of 1900", "ecstasy of gold"))
val format = RecordFormat[Composer]
// record is of type GenericRecord
val record = format.to(ennio)
```

And to go from a record back into a type:

```
// given some record from earlier
val record = ...
val format = RecordFormat[Composer]
// is an instance of Composer
val ennio = format.from(record)
```

Set a schema's decimal scale and precision

Bring an implicit `ScaleAndPrecision` into scope before using `AvroSchema` .

Scala Type	Avro Type

```
import com.sksamuel.avro4s.ScaleAndPrecision

case class MyDecimal(d: BigDecimal)

implicit val sp = ScaleAndPrecision(8, 20)
val schema = AvroSchema[MyDecimal]
```

```
{
  "type":"record",
  "name":"MyDecimal",
  "namespace":"$iw",
  "fields":[{
    "name":"d",
    "type":{
      "type":"bytes",
      "logicalType":"decimal",
      "scale":"8",
      "precision":"20"
    }
  }]
}
```

Coproducts

Avro supports generalised unions, eithers of more than two values. To represent these in scala, we use `shapeless.:+:`, such that `A :+: B :+: C :+: CNil` represents cases where a type is `A` OR `B` OR `C`. See shapeless' documentation on coproducts for more on how to use coproducts.

Type Mappings

```
import scala.collection.{Array, List, Seq, Iterable, Set, Map, Option, Either}
import shapeless.{:+:, CNil}
```

Scala Type	Avro Type
Boolean	boolean
Array[Byte]	bytes
String	string or fixed
Int	int
Long	long
BigDecimal	decimal with default scale 2 and precision 8
Double	double
Float	float
java.util.UUID	string
Java Enums	enum
scala.Enumeration	enum
sealed trait T	enum
Array[T]	array
List[T]	array
Seq[T]	array
Iterable[T]	array
Set[T]	array
Map[String, T]	map

Scala Type	Avro Type
Option[T]	union:null,T
Either[L, R]	union:L,R
A :+: B :+: C :+: CNil	union:A,B,C
Option[Either[L, R]]	union:null,L,R
Option[A :+: B :+: C :+: CNil]	union:null,A,B,C
T	record

Custom Type Mappings

It is very easy to add custom type mappings. To do this, you need to create instances of `ToSchema`, `ToValue` and `FromValue` typeclasses.

`ToSchema` is used to generate an Avro schema for a given JVM type. `ToValue` is used to convert an instance of a JVM type into an instance of the Avro type. And `FromValue` is used to convert an instance of the Avro type into the JVM type.

For example, to create a mapping for `org.joda.time.DateTime` that we wish to store as an ISO Date string, then we can do the following:

```
implicit object DateTimeToSchema extends ToSchema[DateTime] {
  override val schema: Schema = Schema.create(Schema.Type.STRING)
}

implicit object DateTimeToValue extends ToValue[DateTime] {
  override def apply(value: DateTime): String = ISODateTimeFormat.dateTime().print(value)
}

implicit object DateTimeFromValue extends FromValue[DateTime] {
  override def apply(value: Any, field: Field): DateTime = ISODateTimeFormat.dateTime().parseDateTime(value
}
```

These typeclasses must be implicit and in scope when you invoke `AvroSchema` or create an `AvroInputStream` / `AvroOutputStream`.

Selective Customisation

You can selectively customise the way Avro4s generates certain parts of your hierarchy, thanks to implicit precedence. Suppose you have the following classes:

```
case class Product(name: String, price: Price, litres: BigDecimal)
case class Price(currency: String, amount: BigDecimal)
```

And you want to selectively use different scale/precision for the `price` and `litres` quantities. You can do this by forcing the implicits in the corresponding companion objects.

```
object Price {
  implicit val sp = ScaleAndPrecision(10,2)
  implicit val schema = SchemaFor[Price]
}

object Product {
  implicit val sp = ScaleAndPrecision(8,4)
  implicit val schema = SchemaFor[Product]
}
```

This will result in a schema where both `BigDecimal` quantities have their own separate scale and precision.

Using avro4s in your project

Gradle

```
compile 'com.sksamuel.avro4s:avro4s-core_2.11:xxx'
```

SBT

```
libraryDependencies += "com.sksamuel.avro4s" %% "avro4s-core" % "xxx"
```

Maven

```
<dependency>
    <groupId>com.sksamuel.avro4s</groupId>
    <artifactId>avro4s-core_2.11</artifactId>
    <version>xxx</version>
</dependency>
```

Check the latest released version on Maven Central

Building and Testing

This project is built with SBT. So to build

```
sbt compile
```

And to test

```
sbt test
```

Contributions

Contributions to avro4s are always welcome. Good ways to contribute include:

- Raising bugs and feature requests
- Fixing bugs and enhancing the DSL
- Improving the performance of avro4s
- Adding to the documentation

License