spindle Documentation

Release 2.0.0

Jorge Ortiz, Jason Liszka

June 08, 2016

Spindle is a library and code generation tool developed at Foursquare for representing data stored in MongoDB. It's based heavily on Apache Thrift, so understanding some Thrift fundamentals is important. However, most of Spindle is custom Scala code generation developed to work with Thrift models.

Contents:

CHAPTER 1

Thrift

Thrift is a framework that allows you to define serializable datatypes in a language-independent manner. These datatypes are described in `.thrift` files which conform to the Thrift Interface Description Language (IDL). The standard Thrift compiler will read Thrift IDLs and generate code for many different languages: Java, Python, JavaScript, C++, etc. We've built a custom Thrift compiler (documented here) which generates Scala code.

1.1 Data model

Thrift's data model has the following base types:

- `bool` - true/false boolean value (Java's boolean)
- `byte` - 8-bit signed integer (Java's byte)
- `i16` - 16-bit signed integer (Java's short)
- `i32` - 32-bit signed integer (Java's int)
- `i64` - 64-bit signed integer (Java's long)
- `double` - 64 bit IEEE 754 double precision floating point number (Java's double)
- `string` - UTF-8 encoded text string (Java's string)
- `binary` - sequence of unencoded bytes (Java's Array[Byte] or ByteBuffer)

Thrift's data model includes an `enum` type (equivalent to Java's `enum`). On the wire, enums are represented as 32-bit integers.

Thrift's data model also has the following parametrized container types. The parameters `<a>` and `<b>` can be primitives, `enum`s, or `struct`s.

- `list<a>` - ordered list of elements (Java's `List<A>`)
- `set<a>` - unordered list of unique elements (Java's `Set<A>`)
- `map<a, b>` - a map of unique keys to values (Java's `Map<A, B>`). Warning: non-`string` map keys will fail at runtime if you try to use `TBSONObjectProtocol`.

Thrift includes three "class" types:

- `struct` - numbered fields, with a type and an optional default value
- `union` - tagged union types
- `exception` - like `struct`s, but can be used in the "throws" clause of a service

Finally, Thrift allows you to define constants and typedefs. A constant is a value of any Thrift type. A typedef is an alias of any Thrift type.

1.2 Interface definition language (IDL)

Thrift data structures are described in `.thrift` files which conform to the Thrift IDL grammar. You can see the Thrift Tutorial for an example, or reference the formal grammar.

1.3 Serialization formats

There are four serialization formats we use:

- `TBinaryProtocol` - The original binary encoding included with Thrift. It is not well-documented, but is fairly simple. Field names are ommitted (only integer field identifiers are used), types are encoded as byte identifiers, sizes are prepended as integers.
- `TCompactProtocol` - A more compact binary encoding. Also not well-documented, but somewhat described here
- `TReadableJSONProtocol` - Reads and writes human-readable JSON. Unlike `TJSONProtocol`, ituses actual field names for keys, rather than field identifiers. If the field has a `wire_name` annotation, will use that instead of the name. Includes special handling for `ObjectId`.
- `TBSONObjectProtocol` - Reads and writes BSON `DBObjects`. It uses field names for keys. If the field has a `wire_name` annotation, will use that instead of the name. Includes special handling for `ObjectId` and `DateTime`.

CHAPTER 2

Records

2.1 Creating a record

There are three ways to create a Record.

2.1.1 1. `apply` method

The easiest way to create a Record is to use the companion object's `apply` method. This method only works if you know the value of all of the Record's fields, even optional fields. This is useful for structs that have a small, usually fixed number of fields. (Note that if you add another field to your struct, any existing calls to apply will fail due to insufficient arguments.)

Example (Thrift file):

```
struct LatLong {
  1: optional double lat
  2: optional double long
}
```

Example (Scala file):

```
val ll = LatLong(40.65, -73.78)
```

2.1.2 2. Builder

If you need to omit optional fields, or if you want to pass around a partially-constructed Record before you know all its fields, you will want to use a Builder. Builders are created using the `newBuilder` method on a Record's companion object. Fields on the builder are set using methods with the same name as the field. Field setter methods can be passed either a value or an `Option` value. To finish constructing a Builder, call `result` or `resultMutable` (returns a Record that conforms to the `Mutable` trait). Builders enforce that all required fields are set before result can be called.

Example:

```
val v = (Venue
    .newBuilder
    .id(new ObjectId())
    .venuename(Some("Foursquare HQ"))
    .likeCount(None)
    .result())
```

2.1.3 3. Raw

If for whatever reason a Builder is not flexible enough to do what you want, you can always instantiate a Raw class. You can use the `createRawRecord` method on the Record's companion object or you can call `new` directly on the Record's Raw class. This is unsafe, as it will not verify that required fields are set and can seriously corrupt data. Make sure you know what you're doing before you use Raw constructors.

2.2 Reading/writing records

All records have read and write methods that take one argument: a `TProtocol`. We use a number of protocols, depending on the serialization format you want to read from or write to. See the section on *Serialization formats*. Each protocol has a particular way in which it's constructed. Refer to the documentation for each protocol on how to use them. If you're reading in a record, it's acceptable to use the `createRawRecord` method on the MetaRecord companion object to instantiate a record that you can call read on.

2.3 Record interface methods

Every record has a trait of the same name that defines a mostly immutable interface to that record (the exception being some mutable methods for priming foreign keys). Methods that modify the record are generally only available as part of the Mutable trait.

2.3.1 Field access

The record's trait has a bunch of methods for field access. Not all of them are always available. Given a field named `foo`, the following field access methods may be available:

- `foo` - Returns the value of the field. Only available if a default is available (in which case it aliases `fooOrDefault`), or if the field is required (in which case it aliases `fooOrThrow`).
- `fooOption` - Returns an ``Option` with the value of the field, or `None` if the field is not set. Always available.
- `fooOrDefault` - Returns the value of the field, or the default value if the field is not set. Only available if a default is available (either there's an explicit default in the `.thrift` file, or the field is a primitive or a collection, in which case the default is `0`/`0.0`/`false`/`empty`/etc).
- `fooOrNull` - Returns the value of the field, or `null` if the field is not set. For primitives, typed to return the boxed type (e.g., `java.lang.Integer` instead of scala.Int) so that null is valid. In general, avoid using fooOrNull. Prefer fooOption in general and fooOrThrow if you want to fail fast (such as in tests, etc). Only use `fooOrNull` if you're writing performance-sensitive code that can't afford to allocate an `Option`.
- `fooOrThrow` - Returns the value of the field, or throws an exception if the field is not set.
- `fooIsSet` - Returns `true` if the field is set, `false` otherwise.

2.3.2 Special field access

Some field types have special access methods:

- `fooByteArray` - if the field is of the `binary` thrift type, this method will exist and return an `Array[Byte]` (instead of the more efficient `java.nio.ByteBuffer`)

- `fooStruct` - if the field has a `bitfield_struct` or `bitfield_struct_no_setbits` annotation, this method will exist and return a populated bitfield struct. (See: *Bitfields*)
- `fooFk` - if the field is a `foreign_key` field, this method will exist and return the foreign object. (See: *Priming*)

2.4 Other methods

Other methods on the record trait:

- `toString` - produces a JSON-like string representation of the record. It is not strict JSON because it supports non-string map keys.
- `hashCode` - uses `scala.util.MurmurHash` to produce a hash code from all the already-set fields on the record
- `equals` - compares two records to make sure that all their fields have the same set state, and if they're both set that they have the same value
- `compare` - compares two records by their set state for each field and their value for each field
- `copy` - similar to case class copy, creates a shallow copy of a record; has method arguments with default values so you can override the value of a single field or many fields when copying
- `deepCopy` - creates a deep copy of a record; doesn't take arguments
- `mutableCopy` - creates a shallow copy of a record, with the `Mutable` trait as its return type
- `mutable` - if the underlying implementation is mutable, return this typed as a `Mutable` trait, otherwise make a `mutableCopy`
- `toBuilder` - creates a new builder that has been initialized to have the same state as this record

2.5 Mutable trait

TODO: the Mutable trait interface is likely to change before being finalized

2.6 Raw class

TODO

2.7 Priming

The only way to prime records is through the prime method on DatabaseService, which takes a sequence of records, the field to be primed on those records, and the model to be primed on that field. It optionally takes a sequence of already known foreign records and a Mongo `ReadPreference`. For example:

```
val checkins: List[Checkin] = ...
services.db.prime(checkins, Checkin.venueId, Venue)
```

To access the primed foreign object on field `foo`, use the `fooFk` method on the record, which takes the model of the foreign object as an argument and returns an `Option` of the foreign object:

```
val venues: List[Venue] = checkins.flatMap(_.venueIdFk(Venue))
```

(This is somewhat clunky, mostly because of having to pass around the model of the foreign object (in this case, `Venue`) everywhere. This is necessary in order to decouple foreign key fields from the models they point to and so avoid dependency hairballs.)

2.8 Proxies

Spindle can generate proxy classes that can be used to decorate generated models with additional behavior. For example, suppose you have this thrift definition:

```
struct Rectangle {
  1: double length
  2: double width
}
```

Thrift will generate a class `Rectangle`. But suppose you want to add a convenience method `area` to `Rectangle`. First, instruct Spindle to generate a proxy:

```
struct Rectangle {
  1: double length
  2: double width
} (
  generate_proxy="1"
)
```

Thrift will now generate a class `RectangleProxy` that by forwards all of its methods to an underlying `Rectangle` instance. You can now do:

```
class RichRectangle(override val underlying: Rectangle) extends RectangleProxy(underlying) {
  def area = underlying.length * underlying.width
}

val rect: Rectangle = ... // fetch from database
val myRect = new RichRectangle(rect)
myRect.area
```

2.9 Reflection

TODO

2.10 Field descriptors

TODO

CHAPTER 3

Custom types

In addition to all of the standard Thrift types, Spindle codegen makes available a few extras.

3.1 Enhanced types

Enhanced types are types tagged with an annotation so as to produce either a custom protocol format or a custom Scala type. Enhanced types are defined with the `enhanced_types` annotation on a Thrift typedef, along with custom logic in the code generator to deal with the enhanced type. For example:

```
// A BSON ObjectId, stored as a binary blob.
typedef binary (enhanced_types="bson:ObjectId") ObjectId

// A UTC datetime, stored as millis since the epoch.
typedef i64 (enhanced_types="bson:DateTime") DateTime

// A BSON Object, stored as a binary blob
// This is especially useful if you have serialized data to mongo that cannot be represented in thri
typedef binary (enhanced_types="bson:BSONObject") BSONObject
```

Without the enhanced types mechanism, an `ObjectId` would be serialized as binary over the wire and represented as a `ByteBuffer` or an `Array[Byte]` in Scala code. With the enhanced types mechanism, this will be serialized as binary in the binary protocols (`TBinaryProtocol`, `TCompactProtocol`) but receive special handling in `TBSONObjectProtocol` (using the native `ObjectId` type) and in `TReadableJSONProtocol` (using a custom `ObjectId` encoding). In the Scala code, it will be represented as an instance of `ObjectId`.

3.2 Bitfields

In order to use space more efficiently, sometimes you want to store several boolean values into a single integer (`i32`) or long (`i64`) value. Spindle calls these fields "bitfields". Bitfields come in two variants, with and without "set" bits. (With "set" bits, a single boolean value will take two bits, one to determine whether or not the boolean is set, and another for the boolean value itself.)

Spindle has some features to make working with bitfields more convenient. You can associate a `i32` or `i64` field with a "bitfield struct". A bitfield struct is a Thrift struct with only boolean fields. If a field `foo` is marked with a `bitfield_struct` or `bitfield_struct_no_setbits` annotation, then an additional `fooStruct` method will be generated which returns a populated boolean struct.

Bitfield annotations are applied directly to a class field (not through a `typedef`), as follows:

```
struct Checkin {
  1: optional i32 flags (bitfield_struct="CheckinFlags")
}

struct CheckinFlags {
  1: optional bool sendToTwitter
  2: optional bool sendToFacebook
  3: optional bool geoCheat
}
```

In this example, the `Checkin` struct will have all the normal methods for dealing with `flags` as an `i32`, as well as a `flagsStruct` method that returns an instance of `CheckinFlags`.

3.3 Type-safe IDs

You often use BSON `ObjectIds` as primary keys in Mongo collections. This means common collectons like `Checkin` and `Venue` would use the same type as their primary key. In order to avoid errors (such as passing a `List[ObjectId]` of checkin IDs to a method expecting a `List[ObjectId]` of venue IDs), Spindle includes a mechanism for tagging common types so they have distinct type-safe version.

This behavior is triggered by using the `new_type="true"` annotation on a typedef. In Scala code, this will turn the type alias into a tagged type, which means it's a new subtype of the aliased type. For example, `CheckinId` is a tagged `ObjectId` (unique subtype of `ObjectId`). Because it's a subtype, you can use `CheckinId` anywhere you would expect an `ObjectId`. In order to use an `ObjectId` where a `CheckinId` is required, you will need to cast it. A convenience method (with the same name as the type) will be generated to perform this cast. For example: `CheckinId(new ObjectId())`.

Sample usage, in Thrift (`ids.thrift`):

```
package com.foursquare.types.gen

typedef ObjectId CheckinId (new_type="true")
typedef ObjectId VenueId (new_type="true")

struct Checkin {
  1: optional CheckinId id (wire_name="_id")
}

struct Venue {
  1: optional VenueId id (wire_name="_id")
}
```

And in Scala:

```
import com.foursquare.types.gen.IdsTypedefs.{CheckinId, VenueId}

// cast a regular ObjectId to CheckinId
val checkinId: CheckinId = CheckinId(new ObjectId())

// cast a regular ObjectId to VenueId
val venueId: VenueId = VenueId(new ObjectId())

// compile error, this won't work because VenueId and CheckinId are different subtypes of ObjectId
val checkinId2: CheckinId = venueId

// works, because CheckinId is a subtype of ObjectId and can be automatically upcast
val unsafeCheckinId: ObjectId = checkinId
```

Note that the tagged types live in an object with the same name as the thrift file with `Typedefs` appended to the end.

CHAPTER 4

Enums

Enumerations in Thrift are defined with the `enum` keyword. Enumeration values have names and integer identifiers. When compiled to Scala code, enumerations are represented by a class (of the same name as the enumeration), a companion object, and objects for each of the enumeration values (defined inside the companion object).

4.1 Enum value methods

The following methods are available on each value of an enumeration:

- `id` - integer id of the enum value
- `name` - string name of the enum value, as represented in Thrift source (not intended to be stable)
- `stringValue` - string value of the enum value, as represented by `string_value` annotation in Thrift (intended to be stable); if there is no annotation, the name is used
- `toString` - an alias for `stringValue`, although you should not rely on this being stable
- `compare` - compare two enum values by their ids
- `meta` - access the companion object for the enum

4.2 Companion object methods

The following methods are availabel on an enumeration's companion object:

- `findById` - given an integer id, return an `Option` of the enum value with that id, or `None`
- `findByIdOrNull` - given an integer id, return the enum value with that id, or `null`
- `findByIdOrUnknown` - given an integer id, return the enum value with that id. If not found, returns `UnknownWireValue(id)`
- `findByName` - given a string name, return an Option of the enum value with that name, or `None`
- `findByNameOrNull` - given a string name, return the enum value with that name, or `null`
- `findByStringValue` - given an string value, return an Option of the enum value with that string value, or `None`
- `findByStringValueOrNull` - given an string value, return the enum value with that string value, or `null`
- `findByStringValueOrUnknown` - given an string value `v`, return the enum value with that string value. If not found, returns `UnknownWireValue(v)`

- `unapply` - alias for findByName (TODO: should be findByStringValue)

4.3 Matching and unknown values

Spindle will generate an additional value in every enum called `UnknownWireValue`. This value is meant to handle values read off the wire that do not correspond to any known enum value. This can happen when a value is added to an enum, and old code tries to deserialize a value written by new code. So in order for your enum matches to be exhaustive, you must also match against `UnknownWireValue(id)`.

4.4 Serializing to string

By default, enum fields are serialized as integers. For compatibility or readability reasons, you may want them to be serialized as strings instead. To do this, add the `serialize_as="string"` annotation to the field. It's also recommended that you use the `string_value` annotation on the enum values themselves to determine what string each enum serializes to. If this annotation is not specified, the full name of the enum value will be used.

If an enum field is serialized as a string, and an unknown string value is encountered during deserialization, you will get an instance of `UnknownWireValue` containing the unknown string value.

Example:

```
enum CreditCardType {
  Amex = 1 (string_value="a")
  Visa = 2 (string_value="v")
  MasterCard = 3
}

struct CreditCard {
  1: CreditCardType ccType (serialize_as="string")
  2: string number
}

CreditCard(CreditCardType.Amex, "12345678") // ccType is serialized as "a"
CreditCard(CreditCardType.Visa, "12345678") // ccType is serialized as "v"
CreditCard(CreditCardType.MasterCard, "12345678") // ccType is serialized as "MasterCard"
```

4.5 Examples

Example thrift:

```
enum ClientType {
  Android = 1
  Blackberry = 2
  IPhone = 3
  Web = 4
}
```

Example Scala:

```
val client: ClientType = ClientType.Web

client match {
  case ClientType.Android => ...
  case ClientType.Blackberry => ...
  case ClientType.IPhone => ...
  case ClientType.Web => ...
  case ClientType.UnknownWireValue(id) => ???
}
```

CHAPTER 5

Working with MongoDB

5.1 Annotations

In order to use Spindle structs with Mongo, they must have the required Mongo annotations.

- `mongo_collection` - name of the Mongo collection
- `mongo_identifier` - name of the Mongo instance
- `primary_key` - the name of the primary key field

There are also a number of optional Mongo annotations:

- `foreign_key` - one or more annotations with names of foreign key fields
- `index` - one or more annotations with indexed fields

Example:

```
struct Checkin {
  1: optional ids.CheckinId id (wire_name="_id")
  2: optional ids.UserId userId (wire_name="uid")
  2: optional ids.VenueId venueId (wire_name="venueid")
} (primary_key="id",
   foreign_key="userId",
   foreign_key="venueId",
   index="userId: asc",
   index="venueId: asc",
   mongo_collection="checkins",
   mongo_identifier="foursquare")
```

5.2 Rogue queries

There are two main differences between Lift Record and Spindle when it comes to creating and executing Rogue queries. First, queries are not created by calling Rogue methods on a model. Queries must be created explicitly by wrapping a model in a `SpindleQuery` (which we recommend aliasing to `Q`). Second, queries are not executed by called an execution method on the query, the query must be sent to a `SpindleDatabaseService` object to execute it.

For example:

```
import com.foursquare.rogue.spindle.{SpindleQuery => Q}
import com.foursquare.rogue.spindle.SpindleRogue._

val q = Q(Checkin).where(_.userId eqs 646).and(_.photoCount > 0)
val checkins = db.fetch(q)
```

Here is a basic `SpindleDatabaseService` implementation:

```
import com.foursquare.rogue.spindle.{SpindleDBCollectionFactory, SpindleDatabaseService}
import com.foursquare.spindle.UntypedMetaRecord
import com.mongodb.{DB, Mongo, MongoClient, MongoURI}

object db extends SpindleDatabaseService(ConcreteDBCollectionFactory)

object ConcreteDBCollectionFactory extends SpindleDBCollectionFactory {
  lazy val db: DB = {
    val mongoUrl = System.getenv("MONGOHQ_URL")
    if (mongoUrl == null) {
      // For local testing
      new MongoClient("localhost", 27017).getDB("mydatabase")
    } else {
      // Connect using the MongoHQ connection string
      val mongoURI = new MongoURI(mongoUrl)
      val mongo = mongoURI.connectDB
      if (mongoURI.getUsername != null && mongoURI.getUsername.nonEmpty) {
        mongo.authenticate(mongoURI.getUsername, mongoURI.getPassword)
      }
      mongo
    }
  }
  override def getPrimaryDB(meta: UntypedMetaRecord) = db
  override def indexCache = None
}
```

CHAPTER 6

Indices and tables

- genindex
- modindex
- search