Updated: 2021-03-01

4.4 Serialization

Northeastern University

Why is this important?

- Serialization/deserialization in parallel processing:
  - Marshalling/unmarshalling—required for crossing processor boundaries (thread boundaries too where there is little shared data between threads, as is typical with FP).
    - Examples:
      - server/client (web using REST, SOAP, whatever), messaging subscriptions, etc.
      - inter-processor for map/reduce, actor replication, segmentation of Spark datasets, etc.
    - Technologies:
      - typical server/client solution uses JSON (Javascript Object Notation) but HTML and XML also used (esp. for SOAP/WSDL).
      - JVM solution for inter-processor is to use binary (*Serializable* interface).
- Serialization/deserialization for data acquisition/persistence:
  - Streaming data, and regular data ingest (ETL*):
    - use XML (e.g. RSS), JSON, CSV or plain text.
  - Persistence is oriented towards easy analysis:
    - typically use JSON (NoSQL database) but can also be in (append-only) log files, as well as XML.

*** Extract, transform, load**

Different serialized formats (1)

- Binary
  - The original (but always with problems). Still used for JVM serialization. Advantage: opaque; Disadvantages: opaque; versioning problems.
- XML
  - Successor to “SML”, mainly used for legacy situations. Adv: legible, model-driven, flexible; Disadv: too flexible in some respects, verbose, model-driven.
- HTML
  - Originally extension to “XML” but long-since diverged; Adv: legible, standardized, ubiquitous; Disadv: not standardized.
- JSON
  - Originally developed for Javascript, but now almost *de facto* standard. Adv: legible, very flexible (simple model), objects, ubiquitous; Disadv: slightly verbose.

Different serialized formats (2)

- Avro
  - Serialization file format for Hadoop
- CSV
  - Originally designed for Excel, but now almost *de facto* standard for tabular data. Adv: legible, ubiquitous; Disadv: no formal model (e.g. separators, headers, quotes).
- Text, log files
  - Anything goes. Requires NLP/regex to interpret (but Scala very good at that)
- Database
  - Relational, NoSQL or Search Engine: indexed for fast query response but frequently that's not required.
- Other cross-platform capabilities
  - RPC (remote procedure call), Thrift, etc.

Serialization/deserialization: XML

XML: built-in support

Notice that you don't have to put quotes around the XML string.

```
scala> val x1 = <xml><title>Hello, World!</title></xml>
x1: scala.xml.Elem = <xml><title>Hello, World!</title></xml>
```

The "\" operator gives you a sequence (*NodeSeq*) of matching nodes.

```
scala> x1 \ "xml"
res0: scala.xml.NodeSeq = NodeSeq()
```

The "\\" operator gives you a sequence (*NodeSeq*) of matching nodes including their children.

```
scala> x1 \ "title"
res1: scala.xml.NodeSeq = NodeSeq(<title>Hello, World!</title>)

scala> x1 \\ "xml"
res3: scala.xml.NodeSeq = NodeSeq(<xml><title>Hello, World!</title></xml>)

scala> val x2 = <xml><title language="English">Hello, World!</title></xml>
x2: scala.xml.Elem = <xml><title language="English">Hello, World!</title></xml>
```

The "\@" operator gives you the value of the specified attribute.

```
scala> x2 \ "title" map {_ \@ "language"}
res7: scala.collection.immutable.Seq[String] = List(English)
```

- This literal XML feature is great, but not especially useful for reading xml (which we normally do from a file)

```
scala> val xml = XML.loadFile("poets.xml")
xml: scala.xml.Elem = etc…
```

More XML processing

- Of course, we can still use our for-comprehensions:

```
scala> val poets = for (poets <- XML.loadFile("poets.xml")\\"poets"; poet <- poets\"poet") yield poet
poets: scala.xml.NodeSeq = NodeSeq(<poet>
        <name language="en">Wang Wei</name>
        <name language="zh">王維</name>
     </poet>, <poet>
        <name language="en">Li Bai</name>
        <name language="zh">李白</name>
     </poet>)
```

Notice that we (finally) put in an "if" condition into our for-comprehension

- To get the text of a node(s), we just use *node.text:*

```
    scala> val names = for (poets <- XML.loadFile("poets.xml")\\"poets"; poet <- poets\"poet"; name <-
poet\"name"; lang = name\@"language"; if lang=="en") yield name.text
    names: scala.collection.immutable.Seq[String] = List(Wang Wei, Li Bai)
```

- But suppose we want to write out xml for an object?

```
scala> case class Name(name: String, language: String) {
     |   def toXML = <name language={language}>{name}</name>
     | }
defined class Name

scala> val weiWang = Name("Wei Wang","en")
weiWang: Name = Name(Wei Wang,en)

scala> weiWang.toXML
res3: scala.xml.Elem = <name language="en">Wei Wang</name>
```

We use curly brackets {} to insert expressions.

XML in both directions

- Here's the full poets class:

```scala
package edu.neu.coe.scala.poets
import scala.xml.{XML, Node, NodeSeq}
case class Name(name: String, language: String) {
  def toXML = <name language={language}>{name}</name>
}
case class Poet(names: Seq[Name]) {
  def toXML = <poet>{names map (_.toXML)}</poet>
}
object Poet {
  def fromXML(node: Node) = Poet(Name.fromXML(node \
"name"))
}
object Name {
  def getLanguage(x: Option[Seq[Node]]) = x match {case
Some(Seq(y)) => y.text; case _ => ""}
  def fromXML(nodes: NodeSeq): Seq[Name] = for {
    node <- nodes
  } yield
Name(node.text,getLanguage(node.attribute("language")))
}
object Poets extends App {
  def toXML(poets: Seq[Poet]) = poets map {_ toXML}
  val xml = XML.loadFile("poets.xml")
  val poets = for ( poet <- xml \\ "poet" ) yield
Poet.fromXML(poet)
  println(poets)
  println(toXML(poets))
}
```

- Here is the poets file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<class>
    <name>Tang Dynasty Poetry</name>
    <description language="en">This is just an
example XML file</description>
    <instructor>Robin Hillyard</instructor>
    <poets>
        <poet>
            <name language="en">Wang
Wei</name>
            <name language="zh">王維</name>
        </poet>
        <poet>
            <name language="en">Li Bai</name>
            <name language="zh">李白</name>
        </poet>
    </poets>
</class>
```

Serialization/deserialization:
JSON

JSON libraries

- There is no built-in support for JSON in Scala (never was)— that’s appropriate. There are many libraries but we will concentrate on *spray-json* (https://github.com/spray/spray-json).
  - *build.sbt* file:

```
  val sprayGroup = “io.spray"
  val sprayJsonVersion = “1.3.6"
  libraryDependencies ++= List("spray-json") map {c => sprayGroup %% c %
sprayJsonVersion}
```

JSON processing

- Here is our *Poets* object from last time, but with JSON output enabled

```
object Poets extends App {
  import spray.json._
  import scala.xml.XML
  type PoetSeq = Seq[Poet]
  def toXML(poets: PoetSeq) = poets map {_ toXML}
  val xml = XML.loadFile("poets.xml")
  val poets: PoetSeq = for ( poet <- xml \\ "poet" ) yield Poet.fromXML(poet)
  println(poets)
  println(toXML(poets))

  case class Poets(poets: PoetSeq)

  object MyJsonProtocol extends DefaultJsonProtocol {
      implicit val nameFormat = jsonFormat2(Name.apply)
      implicit val poetFormat = jsonFormat1(Poet.apply)
      implicit val poetsFormat = jsonFormat1(Poets)
  }

  import MyJsonProtocol._

  println(poets.toJson)
}
```

We import the *spray.json* classes and, just for convenience, define this *PoetSeq* type.

New *Poets.Poets* class

Here we must create appropriate implicit values. You need them all, else compile errors. Note that for *Name* and *Poet*, which have explicit companion objects, we must reference the apply method explicitly. Note also that *Name* has two fields so requires *jsonFormat2*.

Very important! These *must* be listed in the correct order least-dependent first, most-dependent last. Otherwise it will not compile. That is one aspect of *implicits* that we have to watch out for!

JSON in REPL

- Run the *main* program:

```
  scala> import edu.neu.coe.scala.poets._
  import edu.neu.coe.scala.poets._

  scala> Poets.main(Array[String]())
  List(Poet(List(Name(Wang Wei,en), Name(王維,zh))), Poet(List(Name(Li Bai,en), Name(李白,zh))))
  List(<poet><name language="en">Wang Wei</name><name language="zh">王維</name></poet>, <poet><name language="en">Li
Bai</name><name language="zh">李白</name></poet>)
  JSON: [{"names":[{"name":"Wang Wei","language":"en"},{"name":"王維","language":"zh"}]},{"names":[{"name":"Li
Bai”,"language":"en"},{"name":"李白","language":"zh"}]}]
```

- Add a *fromJson* method to do the opposite transformation:

```
  def fromJson (string: String) = string.parseJson.convertTo[PoetSeq]
```

- Back in the REPL:

```
  scala> import edu.neu.coe.scala.poets._
  import edu.neu.coe.scala.poets._

  scala> import spray.json._
  import spray.json._

  scala>   val source = """[{"names":[{"name":"Wang Wei","language":"en"},{"name":"王維
","language":"zh"}]},{"names":[{"name":"Li Bai","nguage":"en"},{"name":"李白","language":"zh"}]}]"""
  source: String = [{"names":[{"name":"Wang Wei","language":"en"},{"name":"王維","language":"zh"}]},{"names":[{"name":"Li
Bai","language":"en"},{"name":"李白","language":"zh"}]}]

  scala> Poets.fromJson(source)
  res0: edu.neu.coe.scala.poets.Poets.PoetSeq = List(Poet(List(Name(Wang Wei,en), Name(王維,zh))), Poet(List(Name(Li
Bai,en), Name(李白,zh))))
```

- It’s that easy, particularly if you use case classes!