Records, Tags and Pipelines

Serving XML

Daniel **Parker**

Dec 2, 2005

Abstract

Serving XML is a markup language for expressing XML pipelines, and an extendible Java framework for defining the elements of the language. It provides a markup language for expressing flat-XML, XML-flat, flat-flat, and XML-XML transformations in pipelines. This article provides a brief introduction to the vocabulary of this language, and some examples of its flat-XML capabilities.

Table of Contents

1. Introduction

Serving XML is a markup language for expressing XML pipelines, and an extendible Java framework for defining the elements of the language. The XML pipeline is a sequence of tasks performed on a stream of SAX events. The tasks can include XSLT transforms, Schema validation, custom filtering, and fragment processing. The focus is primarily on content conversion, with special emphasis on adapting legacy data formats to XML, and XML to legacy data formats.

The Serving XML open source project [http://servingxml.sourceforge.net/] provides an implementation of this language. It owes a special debt to Cocoon 1 [http://cocoon.apache.org/], circa 1999, for the idea of an XML pipeline. It is also influenced by the XFlat markup language described in [XFlat], and the work by [Rawlins] on legacy data conversion.

Serving XML supports reading XML content and transforming it with XSLT stylesheets and custom SAX filters. The source XML can come from XML files or dynamic XML readers, but it can also come from adaptors of legacy data formats. It can come from flat files or SQL database records, adapted to XML. The output XML can be serialized to XML, HTML, or PDF, but it can also be flattened and written out in legacy data formats. It can be written to flat files or SQL databases.

Some examples of things that Serving XML can do are as follows:

- Convert flat file records to XML, validating each record with an XML Schema, discarding bad records.
- Convert the results of a SQL query to XML.
- Flatten XML to a record stream, to be written to a flat file or a database.
- Convert flat files from one layout to another.
- Convert database records to flat files and vice versa.
- Transform and validate XML with SAX filters, XSLT stylesheets, and schema validation.
- Process a directory of flat files or XML documents.
- Process the fragments of an XML document.

Serving XML attempts to provide a lightweight solution to these problems that can be expressed declaratively in a script, and run on the command line or embedded in a Java application.

2. XML Pipelines

Michael Kay shows how to implement a SAX pipeline using the Java JAXP API in [XSLT], Appendix F. He gives an example of a three-stage pipeline, where XML is streamed first through a pre-processing Java `XMLFilter`, then an XSLT stylesheet, and finally a post-processing Java `XMLFilter`. In Serving XML, this example can be expressed as

```
<sx:resources xmlns:sx="http://www.servingxml.com/core">
  <sx:service name="myPipeline">
    <sx:serialize>
      <sx:transform>
        <sx:saxFilter class="PreFilter"/>
        <sx:style>
```

```
          <sx:urlSource url="filter.xsl"/>
        </sx:style>
        <sx:saxFilter class="PostFilter"/>
      </sx:transform>
    </sx:serialize>
  </sx:service>
</sx:resources>
```

The pipeline can be executed from the command line by entering

```
java -jar servingxml.jar -r resources.xml myPipeline
    < input.xml > output.xml
```

The Serving XML framework needs to load the Java `PreFilter` and `PostFilter` classes, so it must be able to find them in the classpath. When running Serving XML with the `java -jar` command, it is enough to place these files in a `classes` sub-directory under the directory containing the `servingxml.jar` library.

2.1. The Resources Script

A Serving XML resources script looks like this.

```
<sx:resources xmlns:sx="http://www.servingxml.com/core">

  <sx:include href="other-resources.xml"/>

  <sx:service name="customers">
    <!-- pipeline body -->
  </sx:service>

  <sx:service name="suppliers">
    <!-- pipeline body -->
  </sx:service>

</sx:resources>
```

All Serving XML elements that belong to the core vocabulary are in the namespace identified by the URL `http://www.servingxml.com/core`. Pipeline bodies are exposed through named `service` elements. The `service` elements are assigned QNames, and can be executed by name in a Serving XML application. An `include` element allows resource definitions to be pulled in from other resources scripts.

2.2. Conditional Processing

Serving XML supports conditional processing with an `sx:choose` element, which tests XPath boolean expressions against parameters to determine which of several alternative pipeline bodies to execute. Here's an example.

```
<sx:resources xmlns:sx="http://www.servingxml.com/core">

  <sx:parameter name="validate">
    <sx:defaultValue>no</sx:defaultValue>
  </sx:parameter>

  <sx:service name="customers">
    <sx:choose>
      <sx:when test="$validate='yes'"
        <!-- validating pipeline body -->
      </sx:when>
      <sx:otherwise>
        <!-- non-validating pipeline body -->
      </sx:otherwise>
    </sx:choose>
  </sx:service>

</sx:resources>
```

2.3. Parameters

The `sx:parameter` element is used to define a parameter as a QName-value pair, for instance,

```
  <sx:parameter name="validate">no</sx:parameter>
```

A parameter defined inside an element is visible in all sibling and descendent elements, but not in ancestor elements. If the parameter has the same QName as a parameter in an ancestor, the new parameter value replaces the old one within the scope of siblings and descendents, but not in the scope of ancestors, the old value is still visible in ancestor elements. This is to avoid side effects.

The application processing the resources script can pass parameters to the script. You can pass the parameter `validate` through the console app by entering

```
java -jar servingxml.jar -r resources.xml -i input.xml -o output.xml
    customers validate=yes
```

If you want to define a default value for the parameter, you must do so with an `sx:defaultValue` element, like this

```
  <sx:parameter
```

```
name="validate"><sx:defaultValue>no</sx:defaultValue></sx:parameter>
```

Default values can be overriden in a Serving XML application through passed parameters. More generally, default values in a descendent element can be overriden by values set in an ancestor.

2.4. A More Interesting Example

Serving XML supports filters that extract document fragments and perform serialization or other tasks on those fragments. Consider, for example, a file `invoices.xml` containing multiple invoice elements.

```
<inv:invoices xmlns:inv="http://www.telio.be/ns/2002/invoice">
  <inv:invoice id="200302-01" ...

  <inv:invoice id="200302-02" ...
</inv:invoices>
```

By applying the resources script below, you can produce a separate PDF file for each `inv:invoice` fragment, where the output filename is named for the invoice id.

```
<sx:resources xmlns:sx="http://www.servingxml.com/core"
              xmlns:fop="http://www.servingxml.com/extensions/fop"
              xmlns:inv="http://www.telio.be/ns/2002/invoice">

  <sx:service name="invoices">
    <sx:transform>
      <!-- Here we extract a document fragment from the SAX stream -->
      <sx:processFragmentFilter path="/inv:invoices/inv:invoice">
         <!-- Serialize invoice document fragment to pdf-->
         <sx:serialize>
             <!-- We initialize a parameter with an XPATH expression
                  applied to the document fragment -->
            <sx:parameter name="invoice-name" select="@id"/>
            <fop:foEmitter>
              <sx:fileSink file="output/invoice{$invoice-name}.pdf"/>
            </fop:foEmitter>
            <sx:transform>
              <sx:transform ref="steps1-4"/>
             <sx:style><sx:urlSource url="styles/invoice2fo.xsl"/></sx:style>

            </sx:transform>
         </sx:serialize>
      </sx:processFragmentFilter>
    </sx:transform>
  </sx:service>

  <sx:transform name="steps1-4">
```

```
    <sx:style><sx:urlSource url="styles/step1.xsl"/></sx:style>
    <sx:style><sx:urlSource url="styles/step2.xsl"/></sx:style>
    <sx:style><sx:urlSource url="styles/step3.xsl"/></sx:style>
    <sx:style><sx:urlSource url="styles/step4.xsl"/></sx:style>
  </sx:transform>

</sx:resources>
```

Note that the `sx:processFragmentFilter` instruction extracts each invoice fragment from the document, and begins a new serialization task where the fragment becomes the default content. In the output directory, expect to find the following files:

```
invoice200302-01.pdf
invoice200302-02.pdf
```

2.5. Abstract Elements

Serving XML supports the idea of abstract elements. New elements can be created as specializations of abstract elements and used interchangeably with core Serving XML elements in resources scripts. Want your XML serialized to a file on an FTP server? Use the `ftpSink`:

```
<sx:resources xmlns:sx="http://www.servingxml.com/core"
              xmlns:edt="http://www.servingxml.com/extensions/edtftp">

  <edt:ftpClient name="myFtpClient"
                 host="tor3" user="dap" password="spring"/>

  <sx:service name="myPipeline">

    <sx:serialize>
     <sx:xmlEmitter>
       <edt:ftpSink remoteDir="incoming" remoteFile="output.xml">
         <edt:ftpClient ref="myFtpClient"/>
       </edt:ftpSink>
     </sx:xmlEmitter> ...
```

3. Record Streams

3.1. The Record

Serving XML supports the notion of records that have fields, possibly multi-valued, and nested subrecords, possibly repeating.

A record can be represented in BNF notation as follows:

```
Record ::= name (Field+) (Record*) |
  name (Field*) (Record+)

Field:= name (value*)
```

Here, the `name` of the record represents the type of the record.

A record has a defined Java interface,

```
public interface Record {
  RecordType getRecordType();
  String getValueAsString(Name name);
  String[] getValuesAsStrings(Name name);
  Record[] getSegments();
  Record[] getSegments(Name[] path);
  XMLReader createXmlReader();
}
```

Note the `createXmlReader` method, which provides an XML view of a record.

The example below shows the XML representation of an "Employee" type record. This record has three fields, named `Employee-No`, `Employee-Name` and `Children`.

```
<Employee>
  <Employee-No>0001</Employee-No>
  <Employee-Name>Matthew</Employee-Name>
  <Children>Joe</Children>
  <Children>Julia</Children>
  <Children>Dave</Children>
</Employee>
```

Note that `Children` is a multivalued field.

3.2. Record Content

The `sx:recordContent` element adapts a stream of records to XML. It contains a record reader and (optionally) a record mapping.

```
 <sx:recordContent name="employeeDoc">
   <sx:flatFileReader ...
   <sx:recordMapping ...
```

```
 </sx:recordContent>
```

Record readers read a stream of records from a data source. Examples of data sources include

- A comma separated value (CSV) file
- An EDI file
- The results of a SQL query
- The pathname entries in a directory listing

A record mapping maps records to XML. This section is optional, since there is a default mapping that emits the canonical XML representation of records. But typically the XML you want is different from the canonical representation, you may want a field mapped to an attribute rather than an element, for example, or some field mappings contained in a literal element. Now, in principle, you could do that by adding an XSLT stylesheet to the pipeline, but XSLT transformations require in-memory trees, so you wouldn't normally want to do that for a very large flat file. Also, perhaps eighty percent of field mappings can be expressed with very simple mapping instructions.

3.3. Flat File Reader

The employees file below has three pipe-delimited fields: `Employee-No`, `Employee-Name`, and `Children`. The `Children` field is multi-valued, with subfields delimited by semi-colons. Note that employee Scott has no children.

```
Employee-No|Employee-Name|Children
0001|Matthew|Joe;Julia;Dave
0003|Scott|
```

The file layout is described below.

```
<sx:flatFile name="employeesFile">
  <sx:flatFileHeader lineCount="1"/>
  <sx:flatFileBody>
    <sx:flatRecordType name="employee">
      <sx:fieldDelimiter value="|"/>
      <sx:delimitedField name="Employee-No"/>
      <sx:delimitedField name="Employee-Name"/>
      <sx:delimitedField name="Children">
        <sx:subfieldDelimiter value=";"/>
      </sx:delimitedField>
    </sx:flatRecordType>
  </sx:flatFileBody>
</sx:flatFile>
```

The record reader combines a flat file description with a stream source (url, file, file on an FTP server, etc.) If the stream source is omitted, it defaults to the default stream source, which in the console app is the file passed with the -i option.

```
<sx:flatFileReader>
  <sx:flatFile ref="employeesFile"/>
  <sx:urlSource url="data/employees.txt"/>
</sx:flatFileReader>
```

3.4. Record Mapping

Suppose you want the output XML to look like this.

```
<acme:employees xmlns:acme="http://www.AcmeCorporation.com">
  <acme:employee employee-no="0001">
    <acme:name>Matthew</acme:name>
    <acme:children>
      <acme:child>Joe</acme:child>
      <acme:child>Julia</acme:child>
      <acme:child>Dave</acme:child>
    </acme:children>
    </acme:employee>
    <acme:employee employee-no="0003">
    <acme:name>Scott</acme:name>
  </acme:employee>
</acme:employees>
```

This differs from the canonical XML representation in a number of ways. The `Employee-No` field, for example, is mapped as an attribute of `employee`, and individual `sx:child` elements are nested in an `sx:children` element.

The required record mapping is as follows.

```
<sx:recordMapping name="employeesToXmlMapping"
              xmlns:acme="http://www.AcmeCorporation.com">
  <acme:employees>
  <sx:onRecord>
    <acme:employee>
      <sx:fieldAttributeMap field="Employee-No"
                            attribute="employee-no"/>
      <sx:fieldElementMap field="Employee-Name"
                          element="acme:name"/>
      <acme:children>
        <sx:fieldElementMap field="Children"
```

```
                              element="child"/>
      </acme:children>
    </acme:employee>
  </sx:onRecord>
  </acme:employees>
</sx:recordMapping>
```

This record mapping will largely produce the desired tags, with one exception: it will emit an (unwanted) empty `acme:children` element for Scott. That will be fixed up later.

3.5. Finally - Flat to XML

Here is the last stage in the pipeline - the `sx:service` instruction that transforms and serializes the record content. The `sx:removeEmptyElementFilter` does the job of pruning the empty `acme:children` elements from the output.

```
<sx:resources xmlns:sx="http://www.servingxml.com/core"
      xmlns:acme="http://www.AcmeCorporation.com">

  <sx:service name="employees">
    <sx:serialize>
      <sx:transform>
        <sx:content ref="employeesDoc"/>
        <sx:removeEmptyElementFilter elements="acme:children"/>
      </sx:transform>
    </sx:serialize>
  </sx:service>

  <sx:recordContent name="employeeDoc"> ...

</sx:resources>
```

4. More Complicated Flat Files

4.1. Variant Record Types

Many flat files have multiple record types. The trades flat file below has records of two types, where the type is indicated by a two-character tag field at the front of the record.

```
TR0001This is a trade record
TN0002X1234A child transaction
```

This layout can be described as follows.

```
<sx:flatFileBody>
  <sx:flatRecordTypeChoice>
    <sx:positionalField name="record_type" width="2"/>
    <sx:when test="record_type='TR'">
      <sx:flatRecordType name="trade">
        <sx:positionalField name="record_type" width="2"/>
        <sx:positionalField name="id" width="4"/>
        <sx:positionalField name="description" width="30"/>
      </sx:flatRecordType>
    </sx:when>
    <sx:when test="record_type='TN'">
      <sx:flatRecordType name="transaction">
        <sx:positionalField name="record_type" width="2"/>
        <sx:positionalField name="id" width="4"/>
        <sx:positionalField name="reference" width="5"/>
        <sx:positionalField name="description" width="30"/>
      </sx:flatRecordType>
    </sx:when>
  </sx:flatRecordTypeChoice>
</sx:flatFileBody>
```

Here, the fields at the front of the record that go into the record choice appear immediately below an `sx:flatRecordTypeChoice` element. These are followed by a sequence of `sx:when` elements that have `test` attributes containing XPath boolean expressions, which will be evaluated against the leading fields. An optional `sx:otherwise` element can come at the end, for a default.

The first `sx:when` element whose test expression evaluates as true determines the record type. If none do, and if there is an `sx:otherwise` element, the default record is selected. If there is no `sx:otherwise` element, the record is skipped.

4.2. Repeating Groups

Flat files can have repeating groups of fields within a record. Consider the students flat file below.

```
JANEENGLC-MATHA+1972BLUECHICAGOILATLANTAGA
```

The file has the following layout.

name	4 characters
subject-grade	repeating group, repeats twice
year-born	4 characters
favorite-color	4 characters

address	repeating group, repeats twice

It can be described as follows.

```
<sx:flatFileBody>
  <sx:flatRecordType name="student">
    <sx:positionalField name="name" width="4"/>
    <sx:repeatingGroup count="2">
      <sx:flatRecordType name="subject-grade">
        <sx:positionalField name="subject" width="4"/>
        <sx:positionalField name="grade" width="2"/>
      </sx:flatRecordType>
    </sx:repeatingGroup>
    <sx:positionalField name="year-born" width="4"/>
    <sx:positionalField name="favorite-color" width="4"/>
    <sx:repeatingGroup count="2">
      <sx:flatRecordType name="address">
        <sx:positionalField name="city" width="7"/>
        <sx:positionalField name="state" width="2"/>
      </sx:flatRecordType>
    </sx:repeatingGroup>
  </sx:flatRecordType>
</sx:flatFileBody>
```

The XML representation of the record will be.

```
<student>
  <name>JANE</name>
  <subject-grade>
    <subject>ENGL</subject>
    <grade>C-</grade>
  </subject-grade>
  <subject-grade>
    <subject>MATH</subject>
    <grade>A+</grade>
  </subject-grade>
  <year-born>1972</year-born>
  <favorite-color>BLUE</favorite-color>
  <address>
    <city>CHICAGO</city>
    <state>IL</state>
  </address>
  <address>
    <city>ATLANTA</city>
    <state>GA</state>
  </address>
</student>
```

5. More Complicated Record Mappings

One common request from users is for markup to group records, for emitting tags around groups of records that are related in some way. While inserting an XSLT transformation into the pipeline is one possibility, this could be a problem for very large input files. Also, users seem to want something that is closer to the way that reporting tools work, with break logic kicking in on certain events triggered by changes in field values.

5.1. Grouping Records by Field

The financial plan file below has records for `cost` and `revenue`, by `project`, `detail-name`, and `period`.

```
project,detail-name,period,cost,revenue
1767,AD_Sales_SDT_SVA,2003,150,0
1767,AD_Sales_SDT_SVA,2004,24750,0
1767,OPS_SQA,2004,113,0
1785,AD_Sales_SDT_SVA,2004,7920,0
```

Now, suppose you want to group the financial information by `project` and `detail-name`, like this:

```
<Plans>
  <Plan project="1767">
    <Details>
      <Detail detailName="AD_Sales_SDT_SVA">
        <PlanData period="2003" cost="150" revenue="0"/>
         <PlanData period="2004" cost="24750" revenue="0"/>
      </Detail>
      <Detail detailName="OPS_SQA">
        <PlanData period="2004" cost="113" revenue="0"/>
      </Detail>
    </Details>
  </Plan>
  <Plan project="1785">  ...
```

The `sx:groupBy` element can be used to group multiple adjacent records by one or more fields, emitting summary tags around the grouped records.

```
<sx:recordMapping name="plansMapping">
  <Plans>
    <sx:groupBy fields="project">
      <Plan>
        <sx:fieldAttributeMap field="project" attribute="project"/>
        <Details>
          <sx:groupBy fields="project,detail-name">
```

```
            <Detail>
            <sx:fieldAttributeMap field="detail-name" attribute="detailName"/>

              <sx:onRecord>
                <PlanData>
                  <sx:fieldAttributeMap field="period" attribute="period"/>
...
                  <sx:fieldAttributeMap field="cost" attribute="cost"/> ...
                  <sx:fieldAttributeMap field="revenue" attribute="revenue"/>
 ...
```

The `sx:groupBy` instruction works somewhat analogously to "group by" in SQL, except that it only applies to adjacent records. It can be nested to any depth.

5.2. Generalizing Grouping

Some users have requirements for gouping based on the *specific* value of a field, as opposed to breaks in the value. Consider, for example, the data below.

```
BFH01|value01
BCH02|value02
BOH03|value03
...
BOT94|value07
BOH03|value08
...
BOT94|value15

BCT95|value16
BFT99|value17
```

Here, the `BFH01` record type indicates the beginning of a group of level 1, the `BFT99` indicates the end. The `BCH02` indicates the beginning of a group of level 2, the `BCT95` indicates the end. The `BOH03` indicates the beginning of a group of level 3, the `BOT94` indicates the end.

Two record mapping elements have been introduced to address these cases:

- `sx:innerGroup`
- `sx:outerGroup`

These elements contain an `sx:startGroup`, then (optionally) an `sx:endGroup`, then record mapping elements. The `sx:startGroup` and `sx:endGroup` elements define the beginning and end of a group through XPath boolean expressions applied to adjacent records. The `sx:innerGroup` instruction will always skip a record if the record does not satisfy its grouping criteria. The `sx:outerGroup` instruction, in contrast, will always pass the record down to the next nested grouping instruction (if any), even if the record does not satisy its own grouping criteria, but in that case it will not emit any tags inbetween.

The`sx:innerGroup` and `sx:outerGroup` elements are the most general grouping elements, and `sx:groupBy` becomes a special case. The fragment below will emit the same tags as the corresponding `sx:groupBy` section in the previous example.

```
<sx:recordMapping name="plansMapping">
  <Plans>
    <sx:innerGroup>
      <sx:startGroup test="not(sx:previous//project)
            or sx:current//project != sx:previous//project"/>
      <Plan>
         ...
```

Note the `sx:previous` and `sx:current` elements appearing in the XPath expression; these refer to the current and previous records in the record stream. An `sx:next` element can also be used.

6. Extendability

Serving XML works as an "inversion of control" (IoC) container that supports assembling components from a variety of projects - the Apache FOP project, the Sun MSV project and others - and making them work together to process records and XML. The term "inversion" conveys the idea that the Serving XML container does not instantiate components directly, but rather supports an extendible component assembly framework that allows externally defined components to be injected into the container. See [IoC] for a discussion of inversion of control containers.

New components can be created as extensions and used interchangeably with framework components in resources scripts. The `edtftpj` extension, for example, provides the `edt:ftpSource` and `edt:ftpSink` implementations of the abstract `sx:streamSource` and `sx:streamSink` components.

7. Summary

Serving XML is an open source project that is primarily about content conversion, providing a markup language for expressing flat-XML, XML-flat, flat-flat, and XML-XML transformations in pipelines. This article provides a brief introduction to the vocabulary of this language, and some examples of its flat-XML capabilities.

Acknowledgements

The examples referred to in this article are fragments of complete examples contributed by the following users: Pierre-Alexandre Losson, Shaun Woodrow, Dimitrios Varsos, Pedro Mendoza and Ken Brewer. Many users have contributed use cases, examples and bug fixes, and to them I would like to express my gratitude. A special thanks to Michael Kay for answering all of my questions about XSLT and pipelines on the saxon and xsl mailing lists, in the early days of this project.

Bibliography

[XFlat] Bob Lyons *Converting Flat File Content into XML and Vice Versa* *[http://www.infoloom.com/gcaconfs/WEB/TOC/t0413_.HTM]* XML 99, Philadelphia, December 1999

[XSLT] Michael Kay *XSLT 2nd Edition Programmer's Reference* *[http://www.amazon.com/exec/obidos/ASIN/0764543814/102-5652674-7751330]* Wrox Press, 2001

[Rawlins] Michael C. Rawlins *Using XML with Legacy Business Applications* *[http://www.awprofessional.com/bookstore/product.asp?isbn=0321154940&rl=1]* Addison-Wesley Professional, 2003

[IoC] Mike Spille *Inversion of Control Containers* *[http://www.pyrasun.com/mike/mt/archives/2004/11/06/15.46.14/]* The Spille Blog, 2004

Biography

Daniel **Parker**
Consultant
Economic Technology, Inc.
Toronto
Ontario
Canada

Daniel Parker lives and works as a freelance consultant in Toronto. He has worked for a wide variety of corporate clients, including start-ups and large financial institutions. He has been involved in building financial trading and risk management systems, telco provisioning products, web applications, and enterprise integration tools. He can be reached at <danielaparker@gmail.com>