An Introduction to Designing XML Data Documents

By Frank Font of Room4me.com Software LLC © February 2010

What is an XML Data Document?
As long as systems have communicated with each other, system designers have engineered formatting solutions to communicate data between them. There is a long history of custom data structures that are only readable by the systems for which they were built. Such solutions implement formatting conventions that may only exist in those systems and nowhere else. One early attempt to standardize some of this was the EDI initiative, and it had some success in promoting standards within industry domains through the 1990’s. However, EDI is a binary format without widely available development tools and is not internet protocol friendly. For these reasons and others it did not establish itself everywhere the way XML already has. When a system is engineered to communicate data using an XML format, the data is more accessible to more consumers and becomes a candidate for sharing through various popular protocols.

Introduction to XML
XML (eXtensible Markup Language) is a highly-flexible well-established standard for formatting data in a machine independent manner so that it can be processed and consumed by many different kinds of technologies and processes. In many cases, raw XML feeds can even be read and understood directly by people using simple text editors and web browsers. This readability is a great benefit to system professionals when creating systems and diagnosing issues in existing systems. XML is also a common a key component of Service Oriented Architectures (SOA) because it is so portable and the standards are so well supported across many systems.

- XML is at the core of many modern system messaging designs
- XML is the most common feed format used in Web Services
- Many developer and consumer tools are now XML aware

Much of the power of XML comes from the simple accessibility of its format: an XML feed is really just a text file that has been formatted according to the rules of the XML standard. We call these feeds “XML documents” and if you were to look at an XML document using a simple text editor or popular web browser you would see a collection of words, and sometimes numbers, enclosed in angle brackets; and in some cases quotation marks. If you are familiar with HTML, you’ll notice from the sample well-formed XML fragment of Figure 1 that the structure of XML is reminiscent of HTML in that they share common formatting conventions for declaring tags and attributes and use them in the same way. Angle

---

1 The most basic requirement for an XML document is that it be well-formed. This means that every element in the document is properly terminated. Most parsers will tell you right away if an XML document is not well-formed.
brackets enclose tags (we call them elements in XML) and attributes are to the right of element names with assigned values enclosed in quotation marks.

![Figure 1](image1)

Figure 1 – A fragment from a simple XML instance document displayed in a popular web browser

When XML documents are carefully designed for a domain context, they are self-describing, flexible and to a great extent “future-proofed” such that changes to underlying systems do not require wholesale changes to consumers and producers of the XML documents in that system.

**Fundamental Markup Features**

All XML documents have at least one element. The first element in an XML document is called the root element. For example, the root element of the XML fragment in Figure 1 is `Person`.

An XML document is considered “**well-formed**” when all its XML elements in are properly closed. An element can be closed by having an appropriate matching close-marker (see “`</Person>`” in Figure 2) or by including a forward slash before the ending angle bracket (see “`/`” in Figure 3).

![Figure 2](image2)

Figure 2 - Example of Person element closed with matching tag.

Only one element shown in Figure 2 and it is called “Person”. This element has two attributes; `firstname` and `lastname`. The values associated with the attributes appear to the right of each attribute after an equal sign and are enclosed in quotes. XML allows you to use double quotes (e.g., “Jane”) or single quotes (e.g., ‘Jane’), your choice.

![Figure 3](image3)

Figure 3 - Example of Person element closed with embedded slash.

An element can contain attributes and other elements in addition to data, but an attribute can only contain one item of data. For example, Figure 1 shows the `Person` element containing an `Address` element. Both those elements contain attributes of their own. That same figure shows a `Street` element that has no attributes but it does contain the data “123 Elm Street”.

XML is case-sensitive, so `Person` is not the same as `PERSON` and is not the same as `person`. Likewise, `firstname` is not the same as `FirstName` and is not the same as `FIRSTNAME`. For example, it is possible to
create an element with both the attributes *firstname* and *FirstName* and any other combinations of letter case you can think of; but doing such a thing would be insane because it would serve no useful purpose and would be very confusing!

If element names in an XML document seem to include a colon, then that element is explicitly part of a declared namespace. The text in front of the colon is the namespace prefix and the text after the colon is the actual element name. Using namespaces is not necessary if you are not blending elements from different vocabularies into one feed; but if you are blending, it allows you to share identical element names with the other vocabulary even though they may have different meanings. Not adding a namespace qualifier to your element names means your elements are part of the “default” namespace for that XML feed. A fully qualified element name for *Person* might look like Figure 4 if the prefix for the namespace was *ns*.

```
<ns:Person firstname='Jane' lastname='Smith' />
```

*Figure 4 - A fully qualified Person element showing a namespace prefix of "ns".*

**TIP** Don’t worry about the namespace prefix for your XML data document while you are designing the document. A namespace prefix can be changed at anytime and does not have to affect any of your validations and transformations.

### The Core Family of XML Technologies

XML is about markup. It is certainly used to markup data, but it can and is also used to markup rules that processes follow when working with data. The benefit of using XML for more than just data markup is that the anything packaged as XML can then be manipulated using the same tools and processes already developed to work with XML feeds.

<table>
<thead>
<tr>
<th>Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Schema</td>
<td>Schema files commonly have an XSD file extension. A validating XML document processor can consume XML Schema feeds to verify that XML documents conform to an expected structure. The older way of declaring schemas for XML documents was to use DTD files, but that technology is not as powerful and is itself not in an XML format.</td>
</tr>
<tr>
<td>XSL Transformations</td>
<td>XSL Transformation files commonly have XSL or XSLT file extensions. Some XML document processors can transform XML documents by applying the rules encoded in an XSL Transformation file. This is a popular technique for directly converting XML instance documents into HTML pages and other end user formats. XSL Transformation files can also be used to programmatically supplement XSD validation of XML instance documents when coded to return validation codes.</td>
</tr>
</tbody>
</table>
Both XML Schema and XSL Transformations are declarative representations of rules. An XML parser reads these declarations to do specific things to XML instance documents. One of the neat benefits of XML Schema and XSL Transformations being in an XML format is that these technologies can be applied back on themselves. In other words, you can create XML Schemas that validate XSL Transformations and XSL Transformations that process other XSL Transformations or process XML Schemas. Anything formatted as XML can be processed by XML Schema and XSL Transformation solutions.

These technologies can be used in place of procedural solutions, such as validating/transforming XML instance documents by writing a conventional program (e.g., Java, C++, C#, PHP, etc) or stored procedure (e.g., PL/SQL, T-SQL, etc). When you write a conventional program to do these tasks, you then need someone versed in that particular language to maintain it; and if you choose to move into another machine platform environment you may need to re-write that solution in a new language or use a different API on the new system. However, once you implement an XML Schema validation or an XSL Transformation you can be sure it will run on most systems without needing to re-write it. And since XML Schemas and XSL Transformations are declarative, you can write XSL Transformations to morph their content into alternative presentations for other consumers; something that is not as easy to do with programmatic solutions coded as conventional programs. You could even generate a conventional program from an XSD or XSL file by applying an XSL transformation to it; this is not so easy to do the other way around.

In general, when you encode your data validation and transformation knowledge into an XML format you have more programmatic choices and greater solution flexibility than when you commit this knowledge into a standard program or stored procedure. This can pay dividends in reduced maintenance effort, increased logic visibility, and increased transformational and rule communication opportunities.

**TIP** Consider using XML technologies to capture validation and transformation logic instead of writing custom program code in a procedural language. The XML is machine transformable, easily sharable, and will run on many more systems.

**XML Data Document Types**

When we encode data into an XML format for the purpose of communicating that data, we sometimes refer to that XML feed as an *XML Instance Document*. There are two general categories of XML instance documents; *Document-Centric*, and *Data-Centric*.

**Document-Centric**

A Document-Centric XML instance document can be thought of as a text document that has *some* XML sprinkled into it. The XML in such a document is used to explain some key content areas of the document, but not all of. Some or most of the content in this kind of document is not described by any XML element or attribute. This approach to encoding XML instance documents is less powerful than the Data-Centric approach and is generally only appropriate when converting legacy solutions into XML in partial steps.
An Introduction to Designing XML Data Documents

Data-Centric
A Data-Centric XML instance document has XML everywhere. All the content is described by at least one XML element or attribute, nothing is left out. This is the most powerful kind of XML instance document because the entire content can be machine processed using XML processing conventions rather than custom and ad-hoc parsing programs.

TIP When designing an XML data document for a new system, plan on marking up all the data in that document, not just some of it.

Designing Your XML Data Document
When properly designed, an XML data document can be flexible to future growth and friendly toward validation and transformation efforts by document consumers. As an added benefit, carefully designed XML data documents can also be meaningful to direct inspection by developers and knowledgeable data domain experts thus facilitating testing and verification of system components during development and maintenance of the systems that work with those XML documents. Some key questions to ask yourself when starting the design process include the following:

1. Will this XML data document be part of a solution in a data domain that already has design standards? If so, conform to those standards in your design so that your document is more likely to mesh neatly with other components created to work in that domain. (e.g., XBRL, FIXML, RDF, XSPF, EML, RSS, etc)

2. What parts, if any, of the data do we anticipate might change in the future? There are many ways to grow the data in an XML document design, some are better than others. In general, the best way to grow related data in an XML design is to embed dependent data into parent elements as either new attributes or new elements. Keep that in mind when coming up with names and structures so that your names are generic enough to encompass future embedded content but no so generic that they lose meaning.

3. Will direct-readability of the document by people be significantly important? If your files are small, don’t compromise readability of the files. Construct your document so that the element nesting has clear meaning and all element and attribute names are unambiguous. Avoid abbreviations.

4. Will there be much data and does the file-size matter? XML is not a compact data format, but if you want to minimize the file size, choose shorter element and attribute names. Strike a comfortable compromise between compactness and readability.

The Case for Elements
You cannot have an XML document without at least one XML element. And XML elements are very flexible. An element can contain any number of embedded attributes and any number of embedded elements. You can represent hierarchies by embedding elements within elements; and when done
right, these hierarchies describe your data in a clear and rich way. For example, from Figure 1 you intuitively know that John Doe lives in New York. How? Because the State element is embedded within the Address element which is embedded within the Person element. You can see the relationship. We could alternatively have designed the document fragment to capture the same information using just elements as shown in Figure 5.

Most people have heard the refrain “everything is a nail if all you have is a hammer.” There is a line of thinking in XML document design that since XML elements are more flexible than XML attributes, documents should consist exclusively of elements. We can think of all data being “nails” and our “hammer” is the XML element if we chose; but we give something up when we give up attributes.

The Case for Attributes
Think of attributes as seasoning, not the meal. They make the meal better! For example, look at Figure 6 and compare the two Person elements in that fragment.
When displaying an XML instance document, many tools, including popular web browsers provide the ability to expand and collapse element “nodes”. The usual symbols are “+” to indicate something can be expanded and “-” to indicate the tool will allow you to collapse the detail. Collapsing detail is a common practice when navigating through large XML files. Notice in Figure 7 what happens to our XML document from Figure 6 when we collapse the Person elements; we cannot tell who the person is without showing the nested elements.

```
- <People>
  + <Person firstname="John" lastname="Doe">
  + <Person>
  </People>
```

Figure 7 - Cannot identify second person when node is collapsed

There are also significant XSD and XSL benefits to working with attributes when they are available instead of relying on nested elements. The syntax of the corresponding XML Schema and XSL Transformation files is simpler to work with and is sometimes easier to maintain.

Criteria for Deciding between Element and Attribute for Encoding Data

For many reasons (some similar to normalization considerations in relational database design) you want to make sure your attribute is atomic. For example, an attribute called name containing both a first and a last name is not really atomic and thus is a poor design choice.

When we are confident that an item of data is atomic then that data is an excellent candidate to be an attribute. Classic examples of atomic data include firstname, lastname, and middlename. We don’t realistically expect there to be additional information other than the name itself associated with each of those labels. And associating these attributes directly to the most relevant element keeps a collapsed node display clear and makes node selection in XSL Transformations straightforward. Also, for those cases where file size is a consideration, judicious introduction of attributes into the design can help reduce file size without impairing file readability.

Table 2 - Some common attributes

<table>
<thead>
<tr>
<th>Example Attribute Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>firstname</td>
<td>The first name of a person.</td>
</tr>
<tr>
<td>middlename</td>
<td>The middle name of a person.</td>
</tr>
<tr>
<td>lastname</td>
<td>The last name of a person.</td>
</tr>
<tr>
<td>id</td>
<td>A unique identifier associated with the element.</td>
</tr>
<tr>
<td>suffix</td>
<td>The suffix associated with a person.</td>
</tr>
<tr>
<td>gender</td>
<td>The gender of a person.</td>
</tr>
<tr>
<td>category</td>
<td>Holds a context meaningful keyword.</td>
</tr>
<tr>
<td>reference_id</td>
<td>Holds a context meaningful identifier. Substitute a more relevant word for “reference” when appropriate.</td>
</tr>
</tbody>
</table>
An Introduction to Designing XML Data Documents

TIP
When data is atomic and related to only one element, create it as an attribute on that element.

Naming Considerations
If you are using XML for your data encoding, then at some level readability probably matters. A consistent and understandable naming convention certainly will matter to those individuals working directly with the XML instance documents. If the element and attribute names are clear and understandable, the corresponding code that works with those XML instance documents will be easier to write and maintain.

Characters to Avoid in your Names
Feel free to incorporate the underscore "_" character into your names, but avoid the minus symbol "-". Although the minus symbol is valid in names, it causes real headaches sometimes when creating XSL Transformations because the parser may try to interpret it as subtraction.

Word Choice
A common beginner mistake is to prefix all elements or attribute names with a set prefix that identifies the item as part of your vocabulary. For example, if your XML documents will contain biological data, you might be tempted to prefix all your elements with "bio_". Don’t do that! The namespace support of XML is designed expressly to handle that separation for you, so keep your names relevant within your vocabulary context without concerning yourself that the name may clash with someone else’s vocabulary.

Another naming mistake is to create a name that describes a parent child relationship. For example, in a medical data situation you might be inclined to create elements like HospitalLab and HospitalEmergencyRoom. Don’t do it! Instead, create nested elements where each name describes one entity such as what you see in Figure 8.

![Figure 8 - Example XML fragment showing nesting instead of redundantly compound element names.](image)

Only abbreviate when absolutely necessary; and even then try to stay clear. For example a Person element with an attribute called fname instead of firstname is reasonable clear. The same element with an attribute of fn is less clear, but in context might be bearable. The element name Medicine is much better than the abbreviation Med because it is not ambiguous.

Unless your element is explicitly intended to be a container for repeating instances of another element, try to select singular rather than plural words. For example, an element that describes one Person should be called Person not People. Likewise an element that describes one Car should be called Car not
Cars. If however, you want an element to contain repeating occurrences of another element, that wrapper should probably be a plural word; for example the container of all the Person elements could naturally be called People.

Attribute names should always be singular rather than plural because attribute values should always be atomic.

Remember, if someone needs a dictionary to make sense of the names in your XML document design, then your design may have failed at some level. Shoot for clarity.

Name Formatting
There are many schools of thought on how to format element and attribute names. If you are working in a data domain that has an established formatting convention, try to follow it. If you are breaking new ground, establish a pattern that you like and stick to it. Consider having one standard for element names and a different standard for attribute names so that they stand out more clearly in the instance documents and in the programs/transformations that consume those documents.

TIP Apply one consistent naming convention for element names and a different convention for attribute names.

UCC (Uppercase Camel Case)
Names following this convention start with an uppercase letter and for compound names, the connected words also start with a capital letter. Some examples of this formatting convention would include names such as Person, AccountingSystem, EngineDetail, Form1040, and AssayTray. Starting element names with a capital letter is a popular convention.

LCC (Lowercase Camel Case)
Some examples of this formatting convention would include names such as person, accountingSystem, engineType, deliveryVector, assayTray. Starting attribute names with a lowercase letter is a popular convention.

All Uppercase
Some examples of this formatting convention would include names such as PERSON, ACCOUNTINGSYSTEM, ENGINETYPE, DELIVERYVECTOR, ASSAYTRAY.

All Lowercase
Some examples of this formatting convention would include names such as person, accountingsystem, enginetype, and deliveryvector.

Delimiting with Underscore instead of Case
Some examples of this formatting convention would include names such as accounting_system, engine_type, delivery_vector, and ASSAY_TRAY. This is a popular convention for attribute names.
Structure Considerations
A smartly structured XML document design is able to absorb additional data structures without having to alter existing hierarchies or change existing names. Consider that in the future you may enhance your solution to convey more information, but you want to avoid having to rewrite all consumers of your document. XML can handle this scenario, but only if you design your solution correctly.

Expandable Structure
An expandable structure is an element that is not too constrained. For example, if you are building an XML document that contains information about hospitals, you probably want a Hospital element contained within a Hospitals element. If your document will also contain information about doctors, you probably want a collection of Doctor elements too, but might want to avoid putting those elements with full detail into the Hospital element. Otherwise, it becomes difficult to properly accommodate the situation where one doctor has privileges at more than one listed hospital². In that situation, consider creating detailed Doctor elements outside of Hospital and skeleton Doctor elements inside of Hospital. The skeleton elements would only contain a reference_id attribute that corresponds to a unique id attribute on each detailed Doctor element.

TIP Create nested element hierarchies for parent-child relationships that are clear and not likely to change. Generally, don’t do this if the child element potentially has other direct parents.

In general, adding new elements and attributes to an existing XML data document design will not break existing consumers. It will probably require updating existing XSD files, but properly coded XSL Transformation processes are very unlikely to break.

TIP Think about what additional data might be blended into your XML data design in the future and leave room for them by creating forward thinking parent structures.

Bad Structure Changes
A bad structure change to an XML data design is a structural design revision that likely breaks compatibility with existing consumers of that design. Try to avoid the following changes to an existing XML data document design because it is likely to break existing consumers of those XML instance documents:

1. Changing existing parent-child element nestings
2. Changing existing element or attribute names
3. Removing existing elements or attributes

Remember the situations you want to avoid down the road when creating a design today.

² If you have no reservations against repeating data in multiple parts of your XML instance document, then this is not really a concern. However, such a de-normalized implementation is at risk of silent failures as the system evolves and developers add functionality that might result in two identical elements of a feed not having identical information.
Versioning your XML Data Design
If you want to version your design, you can generally handle it via the namespace declaration of the root element in your XML instance document by creating a new namespace for each version. This is not something you need to worry about up-front.

**TIP**
If you have to make a bad structure change to an existing production XML data design, change the namespace so that existing consumers do not fail silently when they receive an instance document they may not be engineered to process.

Involve Stakeholders in the XML Data Design Process
A standard practice in relational database design is to share Entity Relationship Diagrams (ERD) with key stakeholders, generally subject matter experts, to get sign-off on the database design before proceeding with a project. Sometimes this is a pro-forma process because large relational databases can be very tricky to engineer and even clear ERDs are too much for some stakeholders to digest and meaningfully comment on.

An XML data document design however, when presented properly, can be intuitively understood by non-technical subject matter stakeholders even for very sophisticated structures. The key to a proper presentation is to have logical element and attribute names, have reasonable element nestings where such nestings are appropriate, and to present the design in digestible sample chunks.

1. Don’t constrain yourself to sharing one XML sample containing all elements of the design. Instead, create as many small relevant XML document fragments as you need.
2. Each XML fragment that illustrates a key factor in your design approach is as small and simple as possible.
3. Use realistic data in all your XML fragment examples. Confirm with the stakeholders that the sample data is realistic.
4. Don’t hide weak or confusing areas of your design; instead solicit their advice and insight in those areas.
5. Make sure you have gotten feedback on all elements and attributes of your design. Are the names reasonable? Are the anticipated multiplicities valid? Are the parent-child relationships valid?
6. Confirm with stakeholders that all your attributes do in fact only contain atomic data.
7. Don’t be too shy to remind stakeholders that the XML design is a machine readable system component that happens to be human readable. Remind them that it must remain programmer friendly if their suggestions come at the expense of practical constraints that you have prioritized.
8. Plan to have at least one follow-up presentation with the stakeholders to incorporate their feedback before considering the design finalized. Plan as many sessions as you need.

Once you have finalized the XML data document design, create an XML schema (XSD) to machine validate XML instance documents. Don’t plan on sharing the XSD with non-technical stakeholders unless they insist: it is not intuitive and does require significant syntax training to properly understand. Do
however, share the XSD with technical members of your team and solicit their feedback to ensure your XSD is accurate and sufficiently enforces key aspects of your design.

**TIP** Share XML sample fragments with non-technical stakeholders, not XSD files. Get their feedback!

**Test Your XML Instance Documents**

Validating your actual XML instance documents with XSD files prior to going into production is a great idea. The XSD files are also a good technical specification to share with other consumers of your XML instance documents.

Since XML instance documents are generally at the edge of a system component, they lend themselves to independent testing and review. Consider sharing XSD files with a testing person (if you are lucky enough to have one on your team) and show that team resource how to generate and interrogate intermediate XML data documents produced by the system so they can design and run test cases. Their feedback and insight can go a long way to increasing confidence that you got things right; and more importantly, may catch mistakes if you got something wrong. Using XML makes this testing option practical, so take advantage of it!

**TIP** Make inspecting and checking the XML instance documents produced by your system a formal part of your testing methodology prior to going live with your system.

**Summary**

XML is a terrific technology for engineering data sharing solutions that are extremely portable and readily extensible. When properly designed, an XML data document can be extended without impacting existing consumers and producers of earlier XML data document versions. The intuitive self-describing nature of a well designed XML data document also lends itself to materially engaging stakeholders early in the design process resulting in better buy-in and a more robust solution. Next time you need to communicate data between systems, think XML before considering anything else.
References