Extracted from:

Enterprise Integration with Ruby

A Pragmatic Guide

Chapter 3

Processing XML

Exchanging data between processes, components, and companies has always been a vital part of enterprise software. Many attempts have been made to create a universal format for data exchange, but they all have failed for various technical and political reasons.

It's hard to believe it took several decades before something like a standard for a platform-independent data representation was both created *and* accepted. The *eXtensible Markup Language* (XML) has, over the years, evolved into such an industry standard for portable data. That's because it has some useful characteristics:

eXtensible Markup Language

- It is plain text.
- It has been standardized by the W3C.[1]
- It is machine independent (so low-level details such as byte ordering do not matter).
- It is easy to use.
- It supports international character sets.

XML is supported by all modern programming languages. The current Ruby distribution comes with good XML support, but compared to languages such as Java and C#, there is still a lot to be done. On the one hand, it is easy to create and parse XML documents in Ruby. On the other hand, Ruby lacks support for some important technologies such as Document Type Definitions (DTDs), schema validation, and XSLT.

[1] http://www.w3.org/XML

REXML: What's the Difference?

Although several XML parsers exist for Ruby (for example, NQXML[*] or xmlparser[†]), the most popular is Sean Russel's REXML.

The majority of XML parsers are based on either the SAX2 or DOM APIs. These have been standardized and hence look the same in all programming languages. That's certainly a good thing, because if you're familiar with DOM programming in Java, you do not have to learn a lot to do DOM programming in C++ or Ruby.

The downside is that general approaches such as DOM are a compromise and tend not to be tailored to exploit the strengths of a particular language. Sean Russel felt so too, and while looking for better alternatives he found the Electric XML library for Java (created by a company called The Mind Electric).[‡] REXML is a pure Ruby implementation of the original Electric XML API.

REXML is not a copy of the Java API but a genuine Ruby port. All classes and methods have been renamed to follow Ruby conventions, and special Ruby features (such as iterators) have been used wherever it was possible and advantageous. In addition, REXML comes with a lot of features that were not part of the original Electric XML interface. There is support for SAX2, a proprietary stream parsing API, an experimental pull parser, and an experimental RELAX NG validator.

[*] http://nqxml.sourceforge.net

[†] This is a binding for James Clark's expat XML parser. It can be found under http://www.yoshidam.net/Ruby.html#xmlparser.

[‡] The company is called Webmethods today, and the Electric XML library is now integrated into their products. It's no longer available as a stand-alone product.

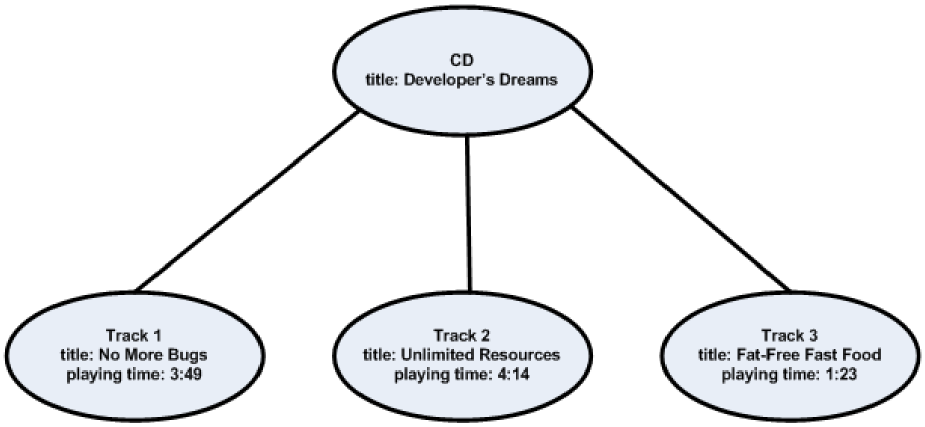

Figure 3.1: Tree Representation of a CD

It's unlikely that you can find a single company in this world that does not use XML in some capacity and a lot of enterprise data is no longer only stored in tables but between angle brackets. Hence, you better know how to extract it and in the following sections we'll cover the most important XML-processing requirements: we'll show you how to create XML documents, how to parse them, and how to validate them.

3.1 A Short XML Reminder

XML is a subset of the more flexible and more liberal *Standard Generalized Markup Language* (SGML). It allows you to define your own markup languages for describing data organized hierarchically in a tree structure. For example, Figure 3.1 shows a possible tree representation of an audio CD. Its XML representation might look like this:

Standard Generalized Markup Language

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!-- Comments look like this! -->
<cd title="Developer's Dreams">
  <track id='1' title='No More Bugs' playing-time='3:49'/>
  <track id='2' title='Unlimited Resources' playing-time='4:14'/>
  <track id='3' title='Fat-Free Fast Food' playing-time='1:23'/>
</cd>
```

All XML documents must be well-formed, which roughly means the following:

- The document must have a single top-level element.
- All elements have to be closed explicitly, and they have to be nested properly; i.e., *<a><b></a></b>* is not allowed, because element *<b>* must be closed before element *<a>*.
- Attributes always have a value, and this value has to be set in single or double quotes. HTML attributes such as NOWRAP or colspan=5 are not allowed in XML documents.

Should I Use Elements or Attributes?

Sometimes it's just a matter of taste, but more often it's a decision that should be made carefully.

The following cases force us to use elements:

- The information you want to describe can potentially occur more than once or can potentially have child elements. It's important to plan for such cases up front—if you are not sure, use an element.
- Whitespace characters are significant.

In other cases, we prefer attributes over elements:

- You do not have to worry about whitespace characters. Using attributes, it's clear that hello differs from ␣hello␣.
- Attributes often produce less noise and are more readable. For example, compare this:

  ```
  <person>
    <name>Homer</name>
    <middle-name>Jay</middle-name>
    <surname>Simpson</surname>
  </person>
  ```

  to this:

  ```
  <person name="Homer" middle-name="Jay" surname="Simpson"/>
  ```

- Attributes are slightly faster, because they usually need less space than elements, and therefore less text has to be processed by the XML parser (this is especially true for documents with long tag names for elements that get opened and closed over and over again). In addition, they increase parsing speed because of the inner structure of most XML parsers. Many XML parsers are event driven and use the SAX2 API. They search for the start tag of elements, and whenever they find one, they call the startElement()

method, transmitting the element name and a list of all attributes belonging to the current element.

If you have a document fragment looking like this:

```
<book>
  <title>Pragmatic Project Automation</title>
  <isbn>0974514039</isbn>
  <publisher>Pragmatic Bookshelf</publisher>
</book>
```

startElement() is called four times (for each of the elements <*book*>, <*title*>, <*isbn*>, and <*publisher*>), and calling methods in programming languages supporting polymorphism is expensive. If we use attributes instead of elements, our document will look like this:

```
<book title='Pragmatic Project Automation' isbn='0974514039'
      publisher='Pragmatic Bookshelf' />
```

Now startElement() is called only once for every <*book*> element. You might not consider this a big performance boost, but if you're Amazon.com and have to parse several hundred thousand books having dozens of elements, it certainly will matter.

3.2 Generating XML Documents

Generating XML documents is often necessary for communicating with other systems. If you are using technologies such as SOAP or XML-RPC, you do not have to worry about the XML generation yourself, because it will be done under the hood by supporting libraries. But there are still many applications today expecting XML documents that you have to create "manually."

In this section we'll show you various techniques for generating XML documents. You'll see how to create documents using rather naive approaches (such as writing raw strings). We'll then look at more sophisticated technologies, such as the REXML API.

To Score Well

From the beginning, customers using PragBouquet's web shop could freely choose from various payment methods: prepaid, invoice, or credit card. But after some months you came to realize that there were actually customers who cheated you. They ordered flowers but never paid for them. Therefore, the company decided to buy a so-called e-score

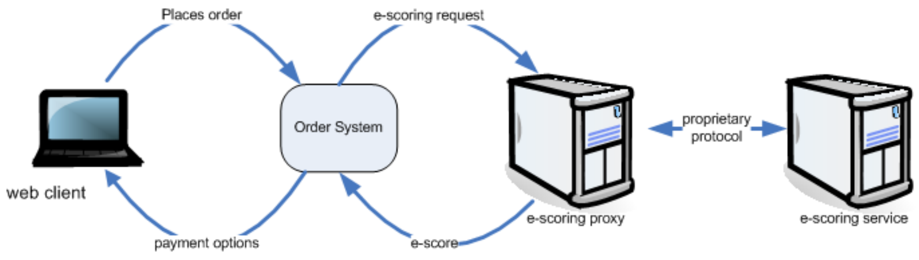

Figure 3.2: e-score architecture

application. This assigns a risk score to each of your new customers. A customer with a low e-score will be allowed to order only if he or she pays up front.

The e-score provider uses a proprietary protocol but gives you a proxy application that hides all that stuff behind an XML/HTTP layer. It expects an XML file containing a list of customers and returns a similar document where every customer is assigned a risk score. You can see the architecture in Figure 3.2 .

It's your task to convert a customer address into an XML document acceptable to the e-scoring application. Based on the response, you'll then decide which payment options will be offered to the customer.

The input documents are simple: they consist of a list of <*person*> elements. Of course, the e-scoring company—like every company employing more than two people—defined its own XML markup, looking like this:

File 169

```
<persons>
  <person name='Max' surname='Mustermann'>
    <address>
      <street>Musterstr. 42</street>
      <city>Berlin</city>
      <postal-code>11011</postal-code>
    </address>
  </person>
</persons>
```

Generating XML Documents Using Raw Strings

Because XML documents are nothing but text, it's tempting to generate them using strings. So, let's start with a simple helper function that tags a certain value:

File 174

```ruby
def tag(tag_name, value, attrs ={})
  tmp = "<#{tag_name}"
  if attrs
    attrs.each { |k,v| tmp += " #{k}='#{v}'" }
  end
  tmp + ">#{value}</#{tag_name}>\n"
end

puts tag('hello', 'world')
puts tag('a', 'b', { 'c' => 'd' })
```

This produces the following:

```
<hello>world</hello>
<a c='d'>b</a>
```

For generating our <*person*> elements, we'll take the object-oriented road—we'll create classes for both addresses and persons. Because they are only storage classes, we use Struct to create them automatically then add to_xml() methods to turn them into XML documents:

File 174

```ruby
Address = Struct.new(:street, :city, :postal_code)

class Address
  def to_xml
    tag('address',
      tag('street', self.street) +
      tag('city', self.city) +
      tag('postal-code', self.postal_code)
    )
  end
end
```

One of the things that makes working with Ruby so much fun is reopening classes. After Struct created an Address class for us, we reopened its definition and added our to_xml() method. It works the same way for the Person class:

File 174

```ruby
Person = Struct.new(:name, :surname, :address)

class Person
  def to_xml
    tag('person',
      self.address.to_xml, {
        'name' => self.name,
        'surname' => self.surname
      }
    )
  end
end
```

Finally, we check whether it all works together:

File 174

```
address = Address.new(
   'Musterstr. 42',
   'Berlin',
   '11011'
)
max_m = Person.new('Max', 'Mustermann', address)
puts max_m.to_xml
```

This produces the following:

```
<person name='Max' surname='Mustermann'><address>
   <street>Musterstr. 42</street>
<city>Berlin</city>
<postal-code>11011</postal-code>
</address>
</person>
```

Although everything looks fine, you should follow this approach only in the simplest cases, because it has some serious disadvantages. For example, you cannot move around and refine document fragments. This is a pity, because XML is such a flexible format and it happens often that new elements or attributes get added to existing document structures. If you've worked exclusively with strings, you have to either parse or manipulate them directly to add the new stuff.

Let's assume we have access to one of those new localization services that determine the geographic coordinates of an address, and you want to add this information to the XML representation of the Address objects without both adding a position attribute and changing to_xml().

Accessing the localization service is easy: you give it the street, the postal code, and the city, and it returns a pair of coordinates:

```
address = Address.new(
  'Musterstr. 42',
  '11011',
  'Berlin'
)
coordinates = LocalizationService.locate(address)
puts coordinates.latitude    # -> 51.5245
puts coordinates.longitude   # -> 6.75
```

Representing the coordinates in XML would probably look like this:

```
<position latitude='51.5245' longitude='6.75'/>
```

How can you add this to an existing XML file containing <*address*> elements? You can try using regular expressions and all the fancy methods of the String class, but think about it for a moment: did you consider all special cases? What about comments or CDATA sections? What

Character	XML Entity	Character	XML Entity
'	&apos;	"	&quot;
<	&lt;	>	&gt;
&	&amp;		

Figure 3.3: XML Standard Entities

about <*address*> elements that don't belong to <*person*> elements? Or <*address*> elements that already have a <*position*> element?

You have to admit that it can get complicated. Sometimes it's nearly impossible to perform this kind of manipulation without parsing the document fragment, adding the new stuff using conventional DOM manipulation methods, and finally creating a new XML string again.

Believe it or not, we still have some disadvantages left. For example, if you work with raw strings, chances are good that you forget to mark up elements correctly as we did in our tag() function previously. What if a person's address is Main Street 7 & 8? The resulting <*street*> element would be as follows:

```
<street>Main Street 7 & 8</street>
```

Every standards-compliant XML parser will reject this, complaining that your document isn't well-formed—blanks are not allowed after an ampersand. Whenever the parser sees an ampersand, it assumes it introduces an entity reference, which has to have an alphanumeric name, ends with a semicolon, and has been defined in a Document Type Definition (DTD). Similar things will happen whenever you use one of XML's special characters. If you really want to use strings for generating your documents, you'll have to replace these characters with their standard entities shown in Figure 3.3 .

Adding such a mechanism to our tag() method is easy:

File 173

```
def encode_markup(text)
  return '' if text.nil? or text == ''
  text.gsub!('&', '&amp;')
  text.gsub!("'", '&apos;')
  text.gsub!('"', '&quot;')
  text.gsub!('<', '&lt;')
  text.gsub!('>', '&gt;')
end
```

```
def tag(tag_name, value, attrs = nil)
  tmp = "<#{tag_name}"
  if attrs
    attrs.each { |k,v| tmp += " #{k}='#{encode_markup(v)}'" }
  end
  tmp + ">#{encode_markup(value)}</#{tag_name}>\n"
end

puts tag('favorite', 'Starsky & Hutch')
```

This produces the following:

```
<favorite>Starsky &amp; Hutch</favorite>
```

You'll face more subtle problems if you ignore character set issues (as we did in the tag() method). An XML document that does not explicitly specify a character set encoding in its header automatically is supposed to contain only UTF-8 characters. For ASCII texts this is perfect, but what if you have a customer from Germany with the popular surname Müller? In UTF-8 the German umlaut, ü, is represented as a two-byte sequence (0xc2, 0x81), but it's a single byte (0xfc) in the character set ISO-8859-1 (see Section 6.1, *Internationalization and Localization*, on page 240, for more details).

Whenever you get text data from an external source, from a database, from a file, or from an HTTP server, for example, you have to determine what character set encoding has been used.

If the specified encoding and the document's content do not match, your XML parser will reject it or—even worse—will misinterpret some characters. Before reading on, you should have a look at Joel Spolsky's awesome essay *The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)*.[2]

For our documents we now have two choices: we can set the encoding attribute in the XML header correctly, or we can convert our final document into UTF-8. To convert texts between different character sets in Ruby, you can use the *Iconv* library. Its interface is simple: a single line of code converts text encoded in the ISO-8859-1 character set into the UTF-8 character set: *Iconv*

```
Iconv.conv('utf-8', 'iso-8859-1', 'Müller')
```

[2]http://www.joelonsoftware.com/articles/Unicode.html

Our final version of method tag() will use this, assuming that all element and attribute values are encoded in ISO-8859-1:

File 172

```
require 'iconv'

def empty?(text) text.nil? or text == ''; end

def encode_markup(text)
  return '' if empty?(text)
  text.gsub!('&', '&amp;')
  text.gsub!("'", '&apos;')
  text.gsub!('"', '&quot;')
  text.gsub!('<', '&lt;')
  text.gsub!('>', '&gt;')
end

def to_utf8(text)
  Iconv.conv('utf-8', 'iso-8859-1', text)
end

def encode(value)
  encode_markup(to_utf8(value))
end

def tag(tag_name, value, attrs ={})
  tmp = "<#{tag_name}"
  if !attrs.nil? and !attrs.empty?
    attrs.each { |k,v| tmp += " #{k}='#{encode(v)}'" }
  end
  if !empty?(value)
    tmp += ">#{encode(value)}</#{tag_name}>\n"
  else
    tmp += "/>\n"
  end
end
```

This version is much better than our first one, but it's also much longer, and it still has some flaws that cannot be solved easily. For example, it does not check whether element and attribute names are valid according to the XML standard (be honest: do you know the rules?). Additionally, you cannot reformat the generated document—you do not have much control over indentation, line breaks, etc. Obviously, generating well-formed XML documents is not as simple as it seems.

Generating XML Documents with REXML

Although REXML does not implement the original DOM interface, it offers an API based on trees. Using this API you can convert an XML document into trees and create trees that represent XML documents.

Everything starts with a document. With REXML you create it like this:

```
require 'rexml/document'
doc = REXML::Document.new
```

An empty document is as useful as an empty bottle of beer. Let's add an element to it:

```
root = REXML::Element.new('my-root')
doc.add_element(root)
puts doc.to_s
```

This produces the following:

```
<my-root/>
```

Creating attributes makes our toolbox complete:

```
root.add_attribute('an-attribute', 'a-value')
puts doc.to_s
```

This results in the following:

```
<my-root an-attribute='a-value'/>
```

Now we can turn our Address object into XML the right way.

File 174

```
class Address
  def to_xml
    adr = REXML::Element.new('address')
    adr.add_element('street').add_text(self.street)
    adr.add_element('city').add_text(self.city)
    adr.add_element('postal-code').add_text(self.postal_code)
    adr
  end
end

address.to_xml.write($stdout, 0)
```

This produces the following:

```
<address>
  <street>Musterstr. 42</street>
  <city>Berlin</city>
  <postal-code>11011</postal-code>
</address>
```

That's how it should look: every element is created explicitly, and the to_xml() method no longer returns a simple string but a document fragment. In addition, we can now use the write() method. This allows us to control the string representation of an XML document. It expects an object derived from IO and the level of indentation to be used.

The result of the to_xml() method can be processed further by other methods now. Adding coordinates to an Address object, for example, can be done like this:

File 174

```
adr = address.to_xml
pos = REXML::Element.new('position')
pos.add_attribute('longitude', '12.345')
pos.add_attribute('latitude', '56.789')
adr.add_element(pos)
adr.write($stdout, 0)
```

This produces the following:

```
<address>
  <street>Musterstr. 42</street>
  <city>Berlin</city>
  <postal-code>11011</postal-code>
  <position latitude='56.789' longitude='12.345'/>
</address>
```

REXML correctly encodes markup characters, but you still can't ignore character set encoding issues. REXML internally uses the UTF-8 character set, so you have to encode all strings before inserting them into a REXML document, and you have to decode them accordingly when reading them back.

We already saw how to achieve this using the Iconv library in Section 3.2, *Generating XML Documents Using Raw Strings*, on page 84. You can also use Ruby's unpack() and pack() methods.

"hello".unpack("C*").pack("U*") turns a string into UTF-8. To do the opposite, call "hello".unpack("U*").pack("C*").

Builder

As we've seen, building well-formed XML documents is not a trivial task. So people constantly try to simplify it. One of those people is Jim Weirich, who produced the Builder library for Ruby.[3] Its core class is Builder::XmlMarkup, which provides everything you need to generate well-formed XML documents. For example:

File 158

```
require 'rubygems'
require 'builder'

doc = Builder::XmlMarkup.new
doc.person(:name => 'Max', :surname => 'Mustermann')
puts doc.target!
```

[3] http://builder.rubyforge.org

This prints the following little document:

```
<person surname="Mustermann" name="Max"/>
```

The technique is probably familiar to Ruby fans: Builder defines a special handler named method_missing() that catches calls to unknown methods and turns them into XML tags with the same name as the method that was called originally. In addition, a hash of parameters is turned into attributes of the newly created element.

The resulting document can be obtained by calling target!() as we did in line 6.

To build hierarchical documents, XmlMarkup has a nice syntax: if you pass a code block to one of those “unknown” methods, it gets passed the current element automatically:

File 158

```
Line 1   xml = ''
     -   doc = Builder::XmlMarkup.new(:target => xml)
     -   doc.person(:name => 'Max', :surname => 'Mustermann') { |person|
     -     person.address { |address|
     5       address.street('Hauptstr. 42')
     -     }
     -   }
     -   puts xml
```

This produces the following:

```
<person surname="Mustermann" name="Max"><address>
   <street>Hauptstr. 42</street></address></person>
```

Intuitive, isn’t it? Note that in line 2 we have specified the target option and set it explicitly to a String object. Hence, Builder fills up the xml variable with our document. The target option accepts any object that responds to the <<(text) operator.

That’s all very nice, but the formatting of the result document is, let’s say, suboptimal. Fortunately, there is the indent option:

File 158

```
Line 1   xml = ''
     -   doc = Builder::XmlMarkup.new(:target => xml, :indent => 2)
     -   doc.person(:name => 'Max', :surname => 'Mustermann') { |person|
     -     person.address { |address|
     5       address.street('Hauptstr. 42')
     -       address.tag!('postal-code', '12345')
     -       address.city('Musterstadt')
     -     }
     -   }
    10   puts xml
```

This prints this beautiful XML document:

```
<person surname="Mustermann" name="Max">
  <address>
    <street>Hauptstr. 42</street>
    <postal-code>12345</postal-code>
    <city>Musterstadt</city>
  </address>
</person>
```

There's also a margin option, which specifies the indentation offset, so you can format your XML documents in any way you like.

Did you notice the little trick with the postal code in line 6? postal-code() is not a valid method name in Ruby, but *<postal-code>* is a perfectly valid XML tag. To get around this, Builder lets you explicitly insert tags using the tag!(sym,*args,&block) method.

To make sure that we do not get into trouble when XML documents without an explicit encoding get prohibited by federal law, we better add another line of code:

File 158

```
xml = ''
doc = Builder::XmlMarkup.new(:target => xml, :indent => 2)
doc.instruct!

doc.person(:name => 'Max', :surname => 'Mustermann') { |person|
  person.address { |address|
    address.street('Hauptstr. 42')
    address.tag!('postal-code', '12345')
    address.city('Musterstadt')
  }

  person.position(:longitude => '12.345', :latitude => '56.789')
}

puts xml
```

This produces this perfect XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<person surname="Mustermann" name="Max">
  <address>
    <street>Hauptstr. 42</street>
    <postal-code>12345</postal-code>
    <city>Musterstadt</city>
  </address>
  <position longitude="12.345" latitude="56.789"/>
</person>
```

That's nearly all you have to know to create XML documents with Builder, but for some special cases you'd might need to use some of the following methods as well:

text!(text)

Allows you to create elements with mixed content:

File 158

```
doc = Builder::XmlMarkup.new(:indent => 2)
doc.foo { |f|
  f.bar
  f.text! "I live outside the bar!\n"
}
puts doc.target!
```

This prints the following:

```
<foo>
  <bar/>
I live outside the bar!
</foo>
```

cdata!(data)

Inserts a CDATA section into an XML document:

File 158

```
doc = Builder::XmlMarkup.new
doc.cdata!('Do not run with scissors!')
puts doc.target!
```

This prints the following:

```
<![CDATA[Do not run with scissors!]]>
```

comment!(text)

Inserts a comment into an XML document:

File 158

```
doc = Builder::XmlMarkup.new
doc.comment!('Some comments are totally useless!')
puts doc.target!
```

This prints the following:

```
<!-- Some comments are totally useless! -->
```

declare!(instruction,*args,&block)

Allows you to insert DTD declarations into your document:

File 158

```
doc = Builder::XmlMarkup.new
doc.declare!(:ENTITY, :pp, 'Pragmatic Programmers')
puts doc.target!
```

This prints the following:

```
<!ENTITY pp "Pragmatic Programmers">
```

Conclusion

It should be clear by now that creating XML documents is by no means as simple as it seems. Because of this, you've probably already received