

# Rails y XML como herramienta de Integración

[jramirez@aspgems.com](mailto:jramirez@aspgems.com)

» ¿por qué xml?

» ¿por qué rails?

» ¿cómo?

» integración vía REST

# ¿por qué xml?

---

- » estándar de facto, universal
- » legible y flexible
- » posibilidad de representar jerarquías

# ¿por qué rails?

---

- » iteradores cómodos
- » consola para pruebas
- » métodos y librerías de xml en la distribución
- » conversión automática

# ¿cómo?

---

» método to\_xml

» builder

generación  
de xml

» XMLSimple

interpretación  
de xml

» REXML

instruct

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<topic>
```

root

```
<parent-id></parent-id>
```

```
<title>The First Topic</title>
```

```
<author-name>David</author-name>
```

```
<id type="integer">1</id>
```

```
<approved type="boolean">>false</approved>
```

```
<replies-count type="integer">0</replies-count>
```

```
<bonus-time type="datetime">2000-01-01T08:28:00+12:00</bonus-time>
```

```
<written-on type="datetime">2003-07-16T09:28:00+1200</written-on>
```

```
<content>Have a nice day</content>
```

```
<author-email-address>david@loudthinking.com</author-email-address>
```

```
<last-read type="date">2004-04-15</last-read>
```

```
</topic>
```

attribute

text

element

# método to\_xml..

---

- » xml desde cualquier modelo, con includes de primer nivel
- » xml desde cualquier hash, incluso anidadas
- » xml desde arrays si todos sus elementos responden a to\_xml

..método to\_xml.

---

- » no soporta atributos
- » sustitución de \_ por -
- » :root, :except, :only,  
:skip\_instruct, :include,  
:indent

- » permite generar cualquier xml de forma sencilla, con elementos, atributos, namespaces, contenido mixto y cualquier anidamiento
- » permite lanzar eventos al estilo SAX



» `Builder::XmlMarkup.new`

» `:target`

» `:indent`

» `:margin`

» `instruct! (tag=:xml,attrs{})`

» `<?xml version="1.0" encoding="UTF-8"?>`

## » comment!

- » xm.comment! 'test'
  - » <!-- test -->

## » declare!

- » xm.declare! :DOCTYPE :article :PUBLIC "//OASIS/DTD"
  - » <!DOCTYPE article PUBLIC "//OASIS/DTD">

## » cdata!

- » xm.cdata! 'test'
  - » <![CDATA[test]]>

## » text!

- » para texto no escapado puede usarse xm << 'test & >'

## » method\_missing

- » `xm.evento 'conferencia rails'`
  - » `<evento>conferencia rails</evento>`
- » `xm.evento 'conferencia', :id=>1, :year=>'2006'`
  - » `<evento year="2006" id="1">conferencia</evento>`
- » `xm.RAILS :evento, 'conferencia', :year=>'2006'`
  - » `<RAILS:evento year="2006">conferencia</RAILS:evento>`

» tag!

## » contenido mixto

```
<p class="weblink">visita la web<a href="http://www.conferenciarails.org"> conferencia rails</a></p>
```

```
xm.p('visita la web ', :class=>'weblink') do
  xm.a 'conferenciarails',
    :href=>'http://www.conferenciarails.org'
end
```



```
xm.p(:class=>'weblink') do
  xm.text! 'visita la web'
  xm.a 'conferencia rails',
    :href=>'http://www.conferenciarails.org'
end
```

» en vistas rxml, accesible mediante el objeto xml

» para usar partials

» en el caller

» `render partial=>'partial_name',  
:locals=>{parent_xml=>xml}`

» en el partial

» `xml = parent_xml unless !parent_xml`

# xmlsimple..

---

- » mapeo de xml a arrays/hashes  
ruby de forma sencilla
- » depende de rexml. rexml viene  
con ruby y xmlsimple con rails
- » es una adaptación de la librería  
perl XML::Simple

- » XmlSimple.xml\_in nil|xml\_string|filename|IO, args
- » XmlSimple.xml\_out (hash)

# ..xmlsimple..

---

```
<opt>
  <person firstname="Joe" lastname="Smith">
    <email>joe@smith.com</email>
    <email>jsmith@yahoo.com</email>
  </person>
  <person firstname="Bob" lastname="Smith">
    <email>bob@smith.com</email>
  </person>
</opt>
```

```
{ 'person' => [
  { 'email' => [ 'joe@smith.com', 'jsmith@yahoo.com' ],
    'firstname' => 'Joe', 'lastname' => 'Smith' },
  { 'email' => [ 'bob@smith.com' ], 'firstname' => 'Bob',
    'lastname' => 'Smith' }
] }
```

# ..xmlsimple..

---

## **ForceArray (true|false) | ([lista])**

```
<opt>
```

```
  <person><email>joe@smith.com</email></person>
```

```
</opt>
```

```
XmlSimple.xml_in (xml,'ForceArray'=>true)
```

```
{'opt'=>[{'person' =>[{ 'email' => [ 'joe@smith.com' ]}]}]}
```

```
XmlSimple.xml_in (xml,'ForceArray'=>false)
```

```
{'opt'=>{'person' =>{ 'email' =>'joe@smith.com'}}}
```

# ..xmlsimple..

---

**KeyAttr [lista] | {elemento=>lista}**

```
<opt>
  <user login="grep" fullname="Gary R Epstein" />
  <user login="stty" fullname="Simon T Tyson" />
</opt>
```

```
xml_in (xml, {'KeyAttr' => 'login' })
```

```
{ 'user' => { 'stty' => { 'fullname' => 'Simon T Tyson' },
  'grep' => { 'fullname' => 'Gary R Epstein' } } }
```

```
xml_in (xml, {'KeyAttr' => { 'user' => "+login" } })
```

```
{ 'user' => {
  'stty' => { 'fullname' => 'Simon T Tyson', 'login' => 'stty' },
  'grep' => { 'fullname' => 'Gary R Epstein', 'login' => 'grep' } }
}
```

# ..xmlsimple..

---

```
xml =%q( <opt> <x>text1</x> <y a="2">text2</y>
</opt>)
```

```
{ 'x' => 'text1',
  'y' => { 'a' => '2', 'content' => 'text2' } }
```

## ForceContent

```
XmlSimple.xml_in(xml, { 'ForceContent' => true })
  { 'x' => { 'content' => 'text1' },
    'y' => { 'a' => '2', 'content' => 'text2' } }
```

## ContentKey

```
XmlSimple.xml_in(xml, { 'ContentKey' => 'text' })
  { 'x' => 'text1',
    'y' => { 'a' => '2', 'text' => 'text2' } }
```

## ..xmlsimple..

---

- » KeepRoot (true|false)
- » RootName (xml\_out) def:opt
- » OutputFile (xml\_out)
- » SupressEmpty (true|nil|'') def:[]
- » Variables (name=>value) \${}

» XmlDeclaration (true|string)

» KeyToSymbol (true|false)

» NoEscape (true|false)

» NormaliseSpace (0|1|2)

» 0: sin normalizar

» 1: normaliza sólo hash keys

» 2: normaliza todo

## ..xmlsimple..

---

» Si queremos usar varias veces los mismos parámetros de inicialización, podemos crear una instancia con las opciones deseadas

```
my_xml = XmlSimple.new (ForceArray=>true,  
  KeepRoot=>true)
```

```
my_xml.xml_in %q(<root><H1>test</H1></root>)
```

..xmlsimple.

---

» Si recibimos un request con  
content-type = "application/xml"  
se usa xmlsimple dejando una  
hash en params[:root\_name]

» Para otros content-types

```
ActionController::Base.param_parsers  
[Mime::Type.lookup(  
'application/xml+soap'  
)] = :xml_simple
```

- » parser xml completo.incluído en ruby
- » soporte de xpath
- » modelos soportados
  - » tree / dom
  - » stream / sax (push)
  - » stream / sax2 (push)
  - » stream / stax (pull)

» modelo DOM, con estilo ruby

java: (hasta jsdk1.5)

```
for (Enumeration e=parent.getChildren();  
     e.hasMoreElements(); )  
  { Element child = (Element)e.nextElement();  
    // Do something with child }
```

rexml:

```
parent.each_child  
  { |child| # Do something with child }
```

» document.root

» element.each

» element.elements

» element.attributes

## » element

- » `element.name` devuelve el tag
- » `element.next_element`
- » `element.previous_element`
- » `element.next_sibling`
- » `element.previous_sibling`
- » `element.root_node`

» elements

- » accesible por index, desde 1
- » opcionalmente por index,nombre
- » accesible por nombre. devuelve el primero con ese tag
- » each\_element\_with\_attribute
- » each\_element\_with\_text
- » elements.to\_a

» attributes

» accessible por nombre

» each { |key,value| }

» to\_a ['key=value',...]

## » xpath

- » `REXML::XPath.first (element,expr)`
- » `REXML::XPath.each (element,expr)`
- » `REXML::XPath.match (element,expr)`

- » xpath desde elements
  - » elements.each (xpath\_expr)
  - » elements[xpath\_expr]
  - » elements.to\_a (xpath\_expr)

## » ejemplos con xpath

```
doc.elements.each('/inventory/section/item')
```

```
doc.elements.each('//item[@stock='18'])
```

```
doc.elements["//price[text()='14.50']/.."]
```

» textos

» accesible mediante `element.text`

» siempre en utf-8

» se sustituyen las entidades tanto XML como definidas en el DTD

## » creación / modificación de nodos

» `el.add_element 'item'`

» `el.add_element 'item', {'stock'=>'18'}`

» `el.text = '4.95'`

» `el.add_text '4,95'`

» `el.delete_element`

» `el.delete_attribute`

» `el['stock'] = '18'`

## » streaming parsers

» lanzan eventos mientras se lee el xml

» más rápidos que DOM y con menos consumo de memoria.  
útiles en documentos grandes

» requieren 'máquina de estados'

» rexml stream parser ~ sax

Document.parse\_stream(source, listener)

» métodos REXML::StreamListener

» tag\_start / tag\_end

» text / cdata

» comment

» entity / entitydecl

» doctype / doctype\_end

» attlistdecl / elementdecl

» instruction / notationdecl / xmldecl

» sax2 parser

» es como el streaming parser,  
pero con la posibilidad de filtrar  
los eventos que se lanzan

» permite escuchar eventos vía  
listeners o mediante procs

---

## proceso de eventos via procs

```
require 'rexml/sax2parser'
```

```
parser = REXML::SAX2Parser.new( File.new(  
  'documentation.xml' ) )
```

```
parser.listen( :characters ) { |text| puts text }
```

```
parser.parse
```

```
parser.listen( :characters, %w{ changelog todo } ) { |  
  text| puts text }
```

```
parser.listen(%w{ item } ) { |  
  uri,localname,qname,attributes| puts qname}
```

---

## proceso de eventos via listeners

```
listener = MySAX2Listener.new
parser.listen( listener )
item_listener = MySAX2Listener.new
parser.listen (%w{item}, item_listener)
parser.parse
```

podemos procesar eventos mediante procs y listeners de forma combinada. Los listeners se procesan más rápidamente, pero necesitamos crear un objeto con los métodos apropiados

» pull parser ~ stax

» en lugar de escuchar eventos, se piden items y luego se comprueba su tipo

» requiere mantener máquina de estados, pero el código es más legible

» el pull parser en rexml está todavía en estado experimental

```
parser = PullParser.new( "<a>text<b att='val'/>txet</a>" )
while parser.has_next?
  res = parser.next
  puts res[1]['att'] if res.start_tag? and res[0] == 'b'
  raise res[1] if res.error?
end
```

# REST: representational state transfer..

---

- » los servicios web son demasiado complejos para muchos casos
- » cuando se realizan consultas, se envía menos información que la que se recibe. podemos exponer la misma api al usuario final y a la máquina

- » modelo rails = rest resource
- » un recurso se identifica con una uri
  - » [www.example.com/item/200776](http://www.example.com/item/200776)
- » permite cachear las peticiones ya servidas

- » rest propone separar las operaciones con resources en tres partes
  - » verbo
  - » nombre
  - » content-type
  
- » a diferencia de XML-RPC, el conjunto de operaciones (verbos) es limitado y hay en su lugar muchos nombres (recursos) diferentes

- » rest efectúa las operaciones CRUD mediante los verbos HTTP PUT/ GET/ POST/ DELETE
- » el content-type puede ser cualquiera, incluso diferente para la entrada y la salida
  - »(ej: url=>xml)

para distinguir la operación a realizar

```
if request.post?  
  if params[:id]  
    # update  
  else # no ID  
    # create  
  elsif request.get?  
    # mostrar el modelo en xml  
  elsif request.delete?  
    # delete  
end
```

para poder servir al cliente diferentes content-types según su cabecera Accept:

```
def show_item
  @item = Item.find(params[:id])
  respond_to do |accepts|
    accepts.html
    accepts.xml {render :xml=>@item.to_xml}
  end
end
```

» delegando en rails para la recepción de parámetros via querystring o xml de forma transparente (usando xmlesimple por debajo) y utilizando el mecanismo respond\_to, podemos servir aplicaciones rest en diferentes formatos con el mínimo esfuerzo

- » las librerías proporcionadas en la distribución estándar de ruby on rails permiten la interpretación y la generación de xml de una forma rápida, sencilla y limpia
- » las facilidades de prueba directa en la consola, agilizan el desarrollo
- » como resultado, usar rails mejora la productividad cuando tratamos xml

# Rails y XML como herramienta de Integración

[jramirez@aspgems.com](mailto:jramirez@aspgems.com)