

# REXML: Parser des documents XML en Ruby

par [Koen Vervloesem](#)

Date de publication : 12 Janvier 2007

Dernière mise à jour : 12 Janvier 2007

REXML (Ruby Electric XML) est l'outil XML de référence pour les développeurs Ruby, livré en standard avec Ruby. Il est rapide, écrit en Ruby, et peut être utilisé de deux façons : parcours d'arbre ou parcours événementiel. Dans cet article, nous allons voir quelques méthodes montrant comment utiliser REXML pour parcourir un XML. Nous allons également voir comment utiliser le debugger interactif de Ruby (irb) pour explorer les documents XML avec REXML. Nous allons utiliser une bibliographie comme exemple de XML. Vous allez apprendre à parcourir le document via l'API de parcours d'arbre, à accéder aux différents éléments et à leurs attributs, ainsi qu'à créer et insérer des éléments. Nous allons également voir les particularités des noeuds texte et du traitement des entités. Enfin, nous verrons un exemple d'utilisation de l'API de parcours événementiel.

- I - Débuter avec le parcours d'arbre
- II - Accéder aux éléments et aux attributs
- III - Création et insertion d'éléments et d'attributs
- IV - Suppression d'éléments et d'attributs
- V - Noeud texte et traitement des entités
- VI - Parcours événementiel
- VII - Conclusion

## I - Débuter avec le parcours d'arbre

### [bibliography.xml](#)

Nous allons commencer avec l'API de parcours d'arbre, qui ressemble beaucoup au DOM mais en plus intuitif. Voici un premier exemple de code :

#### code1.rb - Afficher un fichier XML

```
require 'rexml/document'
include REXML
file = File.new("bibliography.xml")
doc = Document.new(file)
puts doc
```

Le **require** charge la librairie REXML. Nous incluons ensuite l'environnement REXML ; nous n'avons plus ainsi à utiliser de noms comme "REXML::Document" tout le temps. Puis nous ouvrons le fichier existant "bibliography.xml" et nous le parcourons en le stockant dans un objet Document. Enfin, nous affichons le document à l'écran.

Quand vous exécutez la commande "ruby code1.rb", le contenu de notre document XML est affiché.

Il est possible que vous obteniez ce message d'erreur :

```
example1.rb:1:in `require': No such file to load
-- rexml/document (LoadError)
   from example1.rb:1
```

Dans ce cas, c'est dû au fait que REXML n'a pas été installé avec Ruby, ce qui arrive avec certains gestionnaires de packages comme Debian APT qui installent séparément les packages. Installez le package manquant, puis réessayez.

La méthode **Document.new** prend en paramètre des objets de type IO, Document ou String. L'argument spécifie la source à partir de laquelle nous voulons lire le document XML. Dans le premier exemple, nous avons utilisé un objet IO, précisément un objet File qui hérite de la classe IO.

Un autre descendant de la classe IO est la classe Socket, qui peut être utilisée avec **Document.new** pour obtenir un fichier XML via une connexion réseau.

Si le constructeur Document prend en paramètre un objet Document, il sera intégralement cloné dans le nouvel objet Document. Si le constructeur prend en paramètre un objet String, la chaîne attendue devra contenir un flux XML. Petit exemple :

#### code2.rb - Affichage d'un XML contenu dans une chaîne

```
require 'rexml/document'
include REXML
string = <<EOF
<?xml version="1.0" encoding="ISO-8859-15"?>
<!DOCTYPE bibliography PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
"http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd">
<bibliography>
  <biblioentry id="FHIW13C-1234">
    <author>
      <firstname>Godfrey</firstname>
      <surname>Vesey</surname>
    </author>
    <title>Personal Identity: A Philosophical Analysis</title>
    <publisher>
      <publishername>Cornell University Press</publishername>
    </publisher>
  </biblioentry>
</bibliography>
</EOF>
```

#### code2.rb - Affichage d'un XML contenu dans une chaîne

```
<pubdate>1977</pubdate>
</biblioentry>
</bibliography>
EOF
doc = Document.new(string)
puts doc
```

Nous avons utilisé un document de type String : tout les caractères compris entre <<EOF et EOF, nouvelles lignes incluses, font partis de la chaîne.

## II - Accéder aux éléments et aux attributs

A partir de maintenant, nous allons utiliser irb, le débogueur interactif de Ruby, pour les exemples d'utilisation de la librairie REXML.

Au prompt d'irb, nous allons charger le fichier bibliography.xml dans un document. Après ça, nous pourrions exécuter les commandes pour accéder aux éléments et aux attributs de notre document de manière interactive.

```
koan$ irb
irb(main):001:0> require 'rexml/document'
=> true
irb(main):002:0> include REXML
=> Object
irb(main):003:0> doc = Document.new(File.new("bibliography.xml"))
=> <UNDEFINED> ... </>
```

Maintenant, nous pouvons explorer notre document très facilement. Jetons un oeil à une session irb typique avec notre fichier XML :

```
irb(main):004:0> root = doc.root
=> <bibliography id='personal_identity'> ... </>
irb(main):005:0> root.attributes['id']
=> "personal_identity"
irb(main):006:0> puts root.elements[1].elements["author"]
<author>
  <firstname>Godfrey</firstname>
  <surname>Vesey</surname>
</author>
irb(main):007:0> puts root.elements["biblioentry[1]/author"]
<author>
  <firstname>Godfrey</firstname>
  <surname>Vesey</surname>
</author>
irb(main):008:0> puts root.elements["biblioentry[@id='FHIW13C-1260']"]
<biblioentry id='FHIW13C-1260'>
  <author>
    <firstname>Sydney</firstname>
    <surname>Shoemaker</surname>
  </author>
  <author>
    <firstname>Richard</firstname>
    <surname>Swinburne</surname>
  </author>
  <title>Personal Identity</title>
  <publisher>
    <publishersname>Basil Blackwell</publishersname>
  </publisher>
  <pubdate>1984</pubdate>
</biblioentry>
=> nil
irb(main):009:0> root.each_element('//author') {|author| puts author}
<author>
  <firstname>Godfrey</firstname>
  <surname>Vesey</surname>
</author>
<author>
  <firstname>René</firstname>
  <surname>Marres</surname>
</author>
<author>
  <firstname>James</firstname>
  <surname>Baillie</surname>
</author>
<author>
  <firstname>Brian</firstname>
  <surname>Garrett</surname>
</author>
<author>
  <firstname>John</firstname>
  <surname>Perry</surname>
</author>
```

```
<author>
  <firstname>Geoffrey</firstname>
  <surname>Madell</surname>
</author>
<author>
  <firstname>Sydney</firstname>
  <surname>Shoemaker</surname>
</author>
<author>
  <firstname>Richard</firstname>
  <surname>Swinburne</surname>
</author>
<author>
  <firstname>Jonathan</firstname>
  <surname>Glover</surname>
</author>
<author>
  <firstname>Harold</firstname>
  <othername>W.</othername>
  <surname>Noonan</surname>
</author>
=> [<author> ... </>, <author> ...
</>, <author> ... </>, <author> ...
</>, <author> ... </>, <author> ...
</>, <author> ... </>, <author> ...
</>, <author> ... </>, <author> ... </>]
```

Premièrement, nous utilisons le nom "root" pour accéder à la racine de notre document. Ici, la racine du document est l'élément bibliography.

Chaque objet Element a un objet Attributs nommé "attributes" qui agit comme un tableau associatif avec le nom des attributs en guise de clé, et la valeur des attributs en guise de valeur.

Avec **root.attributes['id']** nous avons donc la valeur de l'attribut id de l'élément racine.

De la même façon, chaque objet Element contient un objet Element nommé "elements", et nous pouvons accéder aux sous-éléments en utilisant les méthodes *each* et *[]*.

La méthode *[]* prend comme argument un index ou un Xpath, et retourne l'élément enfant qui correspond à l'expression.

Le Xpath fonctionne comme un filtre, qui va décider quels éléments doivent être retournés.

Notez que **root.elements[1]** est le premier élément enfant, car les index de Xpath commencent à 1, pas à 0. En fait, **root.elements[1]** est équivalent à **root.elements[\*[1]]**, où **\*[1]** est le Xpath du premier enfant.

La méthode *each* de la classe Element parcourt tous les éléments enfants, éventuellement en les filtrant suivant un Xpath donné. Le bloc de code sera alors exécuté à chaque itération. De plus, la méthode **Element.each\_element** est un raccourci pour **Element.elements.each**.

### III - Création et insertion d'éléments et d'attributs

Nous allons maintenant créer une petite bibliographie, consistant en une entrée unique. Voici comment elle se présente :

```

irb(main):010:0> doc2 = Document.new
=> <UNDEFINED/>
irb(main):011:0> doc2.add_element("bibliography",
    {"id" => "philosophy"})
=> <bibliography id='philosophy' />
irb(main):012:0> doc2.root.add_element("biblioentry")
=> <biblioentry />
irb(main):013:0> biblioentry = doc2.root.elements[1]
=> <biblioentry />
irb(main):014:0> author = Element.new("author")
=> <author />
irb(main):015:0> author.add_element("firstname")
=> <firstname />
irb(main):016:0> author.elements["firstname"].text = "Bertrand"
=> "Bertrand"
irb(main):017:0> author.add_element("surname")
=> <surname />
irb(main):018:0> author.elements["surname"].text = "Russell"
=> "Russell"
irb(main):019:0> biblioentry.elements << author
=> <author> ... </>
irb(main):020:0> title = Element.new("title")
=> <title />
irb(main):021:0> title.text = "The Problems of Philosophy"
=> "The Problems of Philosophy"
irb(main):022:0> biblioentry.elements << title
=> <title> ... </>
irb(main):023:0> biblioentry.elements << Element.new("pubdate")
=> <pubdate />
irb(main):024:0> biblioentry.elements["pubdate"].text = "1912"
=> "1912"
irb(main):025:0> biblioentry.add_attribute("id", "ISBN0-19-285423-2")
=> "ISBN0-19-285423-2"
irb(main):026:0> puts doc2
<bibliography id='philosophy'>
  <biblioentry id='ISBN0-19-285423-2'>
    <author>
      <firstname>Bertrand</firstname>
      <surname>Russell</surname>
    </author>
    <title>The Problems of Philosophy</title>
    <pubdate>1912</pubdate>
  </biblioentry>
</bibliography>
=> nil

```

Comme vous le voyez, nous créons un nouveau document vide dans lequel nous ajoutons un élément.

Cet élément devient l'élément racine (root). La méthode *add\_element* prend le nom de l'élément en argument et un argument facultatif qui est la paire nom/valeur du tableau associatif de l'attribut.

Cette méthode ajoute donc un nouveau fils au document ou à l'élément, optionnellement elle peut aussi définir les attributs d'un élément.

Vous pouvez aussi créer un nouvel élément, comme nous l'avons fait avec l'élément "author", et l'ajouter après n'importe quel élément : si la méthode *add\_element* prend un objet *Element*, celui-ci sera ajouté à l'élément parent.

A la place de la méthode *add\_element*, vous pouvez aussi utiliser la méthode *<<* sur *Element.elements*.

Ces deux méthodes retournent l'élément ajouté.

En complément, avec la méthode `add_attribute`, vous pouvez ajouter un attribut à un élément existant. Le premier paramètre est le nom de l'attribut, le second est sa valeur. La méthode retourne l'attribut qui a été ajouté.

La valeur du texte d'un élément peut être facilement changée avec `Element.text` ou bien avec la méthode `add_text`.

Si vous voulez insérer un élément à une position spécifique, vous pouvez utiliser les méthodes `insert_before` et `insert_after`:

```
irb(main):027:0> publisher = Element.new("publisher")
=> <publisher/>
irb(main):028:0> publishername = Element.new("publishername")
=> <publishername/>
irb(main):029:0> publishername.add_text("Oxford University Press")
=> <publishername> ... </>
irb(main):030:0> publisher << publishername
=> <publishername> ... </>
irb(main):031:0> doc2.root.insert_before("//pubdate", publisher)
=> <bibliography id='philosophy'> ... </>
irb(main):032:0> puts doc2
<bibliography id='philosophy'>
  <biblioentry id='ISBN0-19-285423-2'>
    <author>
      <firstname>Bertrand</firstname>
      <surname>Russell</surname>
    </author>
    <title>The Problems of Philosophy</title>
    <publisher>
      <publishername>Oxford University Press</publishername>
    </publisher>
    <pubdate>1912</pubdate>
  </biblioentry>
</bibliography>
=> nil
```

## IV - Suppression d'éléments et d'attributs

Les méthodes `add_element` et `add_attribute` ont leur équivalents respectifs pour détruire les éléments et les attributs. Voici comment cela fonctionne avec les attributs :

```
irb(main):033:0> doc2.root.delete_attribute('id')
=> <bibliography> ... </>
irb(main):034:0> puts doc2
<bibliography>
  <biblioentry id='ISBN0-19-285423-2'>
    <author>
      <firstname>Bertrand</firstname>
      <surname>Russell</surname>
    </author>
    <title>The Problems of Philosophy</title>
    <publisher>
      <publishername>Oxford University Press</publishername>
    </publisher>
    <pubdate>1912</pubdate>
  </biblioentry>
</bibliography>
=> nil
```

La méthode `delete_attribute` retourne l'attribut détruit.

La méthode `delete_element` peut prendre un objet Element, une chaîne de caractères ou un index comme argument :

```
irb(main):034:0> doc2.delete_element("//publisher")
=> <publisher> ... </>
irb(main):035:0> puts doc2
<bibliography>
  <biblioentry id='ISBN0-19-285423-2'>
    <author>
      <firstname>Bertrand</firstname>
      <surname>Russell</surname>
    </author>
    <title>The Problems of Philosophy</title>
    <pubdate>1912</pubdate>
  </biblioentry>
</bibliography>
=> nil
irb(main):036:0> doc2.root.delete_element(1)
=> <biblioentry id='ISBN0-19-285423-2'> ... </>
irb(main):037:0> puts doc2
<bibliography/>
=> nil
```

Le premier appel à `delete_element` dans notre exemple utilise une expression XPath afin de localiser l'élément à détruire.

La seconde fois, nous utilisons l'index 1, ce qui signifie que le premier élément dans le document racine (root) sera détruit.

La méthode `delete_element` retourne l'élément détruit.

## V - Noeud texte et traitement des entités

Nous avons déjà utilisé les noeuds texte dans les exemples précédents.

Dans cette section nous allons voir des fonctions avancées avec ces noeuds texte. Spécialement, Comment REXML prend en compte les entités ?

REXML n'est pas un parser validateur, et donc il n'est pas nécessaire d'assigner les entités externes. Les entités externes ne sont donc pas remplacées par leur valeur, mais les entités internes le sont: Quand REXML parcourt un document XML, il traite la DTD et crée une table avec les entités internes et leur valeur.

Lorsque l'une de ces entités est rencontrée dans le document, REXML la remplace par sa valeur.

Un exemple :

```
irb(main):038:0> doc3 = Document.new('<!DOCTYPE testentity [
irb(main):039:1' <!ENTITY entity "test">]>
irb(main):040:1' <testentity>&entity; the entity</testentity>')
=> <UNDEFINED> ... </>
irb(main):041:0> puts doc3
<!DOCTYPE testentity [
<!ENTITY entity "test">]>
<testentity>&entity; the entity</testentity>
=> nil
irb(main):042:0> doc3.root.text
=> "test the entity"
```

Vous pouvez voir que le document XML, lors de son impression, contient l'entité correcte. Lorsque vous accédez au texte, l'entité "&entity;" est correctement transformée en "test".

Cependant, REXML n'utilise pas une évaluation très poussée des entités. Comme résultat, nous voyons ce problème survenir :

```
irb(main):043:0> doc3.root.text = "test the &entity;"
=> "test the &entity;"
irb(main):044:0> puts doc3
<!DOCTYPE testentity [
<!ENTITY entity "test">
]>
<testentity>&entity; the &entity;</testentity>
=> nil
irb(main):045:0> doc3.root.text
=> "test the test"
```

Comme vous le voyez, le texte "test the &entity;" a été modifié en "&entity; the &entity;".

Si vous changez la valeur de l'entité, cela vous retournera un résultat différent de votre attente : plus de chose seront modifiées dans votre document que vous ne le vouliez.

Si cela est problématique pour votre application, vous pouvez appliquer le flag `:raw` sur n'importe quel noeud texte ou Elements, et même sur le noeud Document. Les entités dans ce noeud ne seront pas traitées et dans ce cas la, vous aurez à les traiter par vous même

Un exemple :

```
irb(main):046:0> doc3 = Document.new('<!DOCTYPE testentity [
irb(main):047:1' <!ENTITY entity "test">]>')
```

```
irb(main):048:1' <testentity>test the &entity;</testentity>',
      {:raw => :all})
=> <UNDEFINED> ... </>
irb(main):049:0> puts doc3
<!DOCTYPE testentity [
<!ENTITY entity "test">
]>
<testentity>test the &entity;</testentity>
=> nil
irb(main):050:0> doc3.root.text
=> "test the test"
```

Les caractères spéciaux comme "&", "<", ">", "\"" (guillemets), et ' sont automatiquement convertis.

D'ailleurs, si vous écrivez un de ces caractères dans un noeud texte ou dans un attribut, REXML les convertira dans son entité équivalente. Ex: "&amp;" pour "&".

## VI - Parcours événementiel

Le parcours événementiel est plus rapide que le parcours d'arbre. Si la vitesse est un critère, le parcours événementiel peut être utile.

Cependant, les options comme XPath ne sont pas valides. Vous devez avoir une class d'audit ("listener") et chaque fois que REXML rencontrera un évènement (balise de début, balise de fin, texte, etc.), le "listener" recevra une notification de l'évènement.

Un programme d'exemple :

### code3.rb - Parcours événementiel en action

```
require 'rexml/document'
require 'rexml/streamlistener'
include REXML

class Listener
  include StreamListener
  def tag_start(name, attributes)
    puts "Start #{name}"
  end
  def tag_end(name)
    puts "End #{name}"
  end
end

listener = Listener.new
parser = Parsers::StreamParser.new(File.new("bibliography2.xml"), listener)
parser.parse
```

### [bibliography2.xml](#)

Exécuter code3.rb donne cette sortie :

```
koan$ ruby code3.rb
Start bibliography
Start biblioentry
Start author
Start firstname
End firstname
Start surname
End surname
End author
Start title
End title
Start publisher
Start publishername
End publishername
End publisher
Start pubdate
End pubdate
End biblioentry
End bibliography
```

## VII - Conclusion

Ruby et XML font une bonne équipe.

Le processeur XML REXML vous permet de créer, accéder et modifier vos documents XML en une seule fois et ce de façon très intuitive. Avec l'aide du debugger interactif irb de Ruby, vous pouvez aussi lire vos documents XML très aisément.

Quelques liens supplémentaires :

- [L'article original](#)
- [Site officiel de REXML](#)
- [Documentation de l'API REXML](#)

*[Télécharger les codes-source de l'article](#)*