

Generación Automática de Código

Trabajo, Práctica #1

Generador de JSON a XML

UNED – Máster Universitario En Investigación En Ingeniería De Software Y Sistemas

H. Paúl Pasquel G.

Noviembre 2017

Tabla de Contenido

| | | |
|-------|---|----|
| 1 | Introducción..... | 3 |
| 2 | Desarrollo de la Práctica | 3 |
| 2.1 | Recursos de Programación..... | 3 |
| 2.2 | Descripción de la Solución | 3 |
| 2.2.1 | Esquema de la solución | 4 |
| 2.2.2 | Elementos de la solución..... | 5 |
| 2.3 | Alcances y limitaciones de la solución..... | 9 |
| 2.4 | Descripción de los casos de Prueba | 9 |
| 2.4.1 | Primer Ejemplo..... | 9 |
| 2.4.2 | Segundo Ejemplo..... | 10 |
| 2.4.3 | Tercer Ejemplo | 11 |
| 2.4.4 | Cuarto Ejemplo..... | 12 |
| 2.4.5 | Quinto Ejemplo | 12 |
| 2.4.6 | Sexto Ejemplo..... | 13 |
| 2.4.7 | Séptimo Ejemplo | 13 |
| 3 | Programas Fuentes | 13 |
| 4 | Utilización | 14 |
| 4.1 | Requisitos..... | 14 |
| 4.2 | Instalación..... | 14 |
| 4.3 | Ejecución | 14 |
| 5 | Conclusiones y Reflexiones | 16 |
| 6 | Bibliografía y Referencias..... | 17 |
| 7 | Índice de Figuras | 17 |

1 Introducción

El objeto del presente documento es detallar cada uno de los elementos utilizados para llevar a cabo la primera práctica.

El objetivo de esta práctica se crear un generador encargado de transformar una cadena JSON en un XML. El segundo objetivo es obtener el conocimiento necesario del lenguaje de programación Ruby.

He considerado que los apartados indicados como referencia en el enunciado de la práctica son suficiente para describir de forma detallada la solución propuesta.

Desarrollo de la Práctica, mostrará el planteamiento del problema, los mecanismos lógicos que se usarán, y finalmente el detalle del código fuente (principal) que se codifico para cumplir con el requerimiento. También describe el alcance y limitaciones del generador. Además, muestra, a través de ejemplos, las funcionalidades implementadas, y las restricciones del mismo.

En la siguiente sección, Programas Fuentes, se describen los archivos físicos que se entregan con la práctica.

La sección Utilización, describe la instalación, y muestra con un ejemplo los pasos requeridos para ejecutar el generador.

La sección Conclusiones, detalle los comentarios, sugerencias y reflexiones que se obtuvieron a partir de la elaboración del generador y su documentación.

Finalmente se coloca una sección de Referencias, con todos los recursos, texto utilizados en el desarrollo de la práctica y de este documento.

2 Desarrollo de la Práctica

2.1 Recursos de Programación

Los elementos utilizados en la práctica fueron:

- Ruby como lenguaje de programación.
- Interprete de Ruby 2.4.1 como entorno de ejecución
- Eclipse como IDE para realizar la codificación
- Windows como sistema operativo base
- Clase StringScanner de Ruby como componente para “barrarse” la cadena JSON

2.2 Descripción de la Solución

De acuerdo a la descripción obtenido de los tipos de generadores de códigos, la presenta práctica será resuelta utilizando el tipo Munger [1] (toma un archivo de entrada, los procesa, y genera un archivo de salida con el resultado)

2.2.1 Esquema de la solución

Para obtener una solución a este tipo de problema, los pasos más idóneos a seguir deberían ser:

- Definir un Modelo. El modelo representa, en el lenguaje de programación, a la estructura JSON.
- Crear un componente “Parser” (Generador). Se encarga de generar, a partir de una cadena de caracteres, una representación del Modelo.
- Generar un mecanismo de conversión. Se encarga de transformar la representación del Modelo, en el esquema final deseado. En el caso de esta práctica XML
- Finalmente se requiere dos procesos destinados a manejar los correspondientes archivos, el primero lee y carga el archivo original a memoria, el segundo envía la cadena resultado a un correspondiente archivo de respuesta.

Para iniciar la elaboración del generador se proponen los siguientes ejemplos de las entradas y salidas esperadas:

Ejemplo 1:

```
{
  "Person" : {
    "Pedro" : {
      "name" : "Pedro Rodriguez",
      "age" : 32,
      "phone" : "593 223 4445",
      "married" : true,
      "sons" : ["Peter", "Luis"]
    }
  }
}
```

Resultado esperado

```
<Person>
  <Pedro>
    <name>"Pedro Rodriguez"</name>
    <age>32</age>
    <phone>"593 223 4445"</phone>
    <married>true</married>
    <sons id="1">"Peter"</sons>
    <sons id="2">"Luis"</sons>
  </Pedro>
</Person>
```

Ejemplo 2:

```
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}
```

```

    ]
  }
}}

```

Resultado esperado

```

<menu>
  <id>"file"</id>
  <value>"File"</value>
  <popup>
    <menuitem id="1">
      <value>"New"</value>
      <onclick>"CreateNewDoc () "</onclick>
    </menuitem>
    <menuitem id="2">
      <value>"Open"</value>
      <onclick>"OpenDoc () "</onclick>
    </menuitem>
    <menuitem id="3">
      <value>"Close"</value>
      <onclick>"CloseDoc () "</onclick>
    </menuitem>
  </popup>
</menu>

```

2.2.2 Elementos de la solución

2.2.2.1 Modelo

El modelo está representado por las clases Pair (par) y JObject (objeto), y se basa en las siguientes reglas de la estructura de un elemento JSON [2]

- Objeto: Un objeto JSON siempre está encerrado en llaves ({}). Contiene ninguno, uno o varios pares (Pair).
- Pair: El elemento mínimo es un par (Pair), el cual se conforma por una clave (key) y un valor (value). Separados por dos puntos (:).
- Key: La clave (key) siempre será una cadena encerrada en comillas dobles.
- Value: Un valor puede corresponder a un Objeto, un Arreglo o un Valor Simple (Simple Value).
- Simple Value: Un valor simple corresponde a:
 - Una cadena de caracteres encerradas en comillas dobles, incluye el vacío ("")
 - Un número, que opcionalmente puede tener una parte decimal.
 - Un valor booleano true o false
- Array: Un arreglo JSON siempre se encuentra encerrado entre corchetes ([]). Puede contener ninguno, uno o más valores (Values); estos separados por una coma (,)

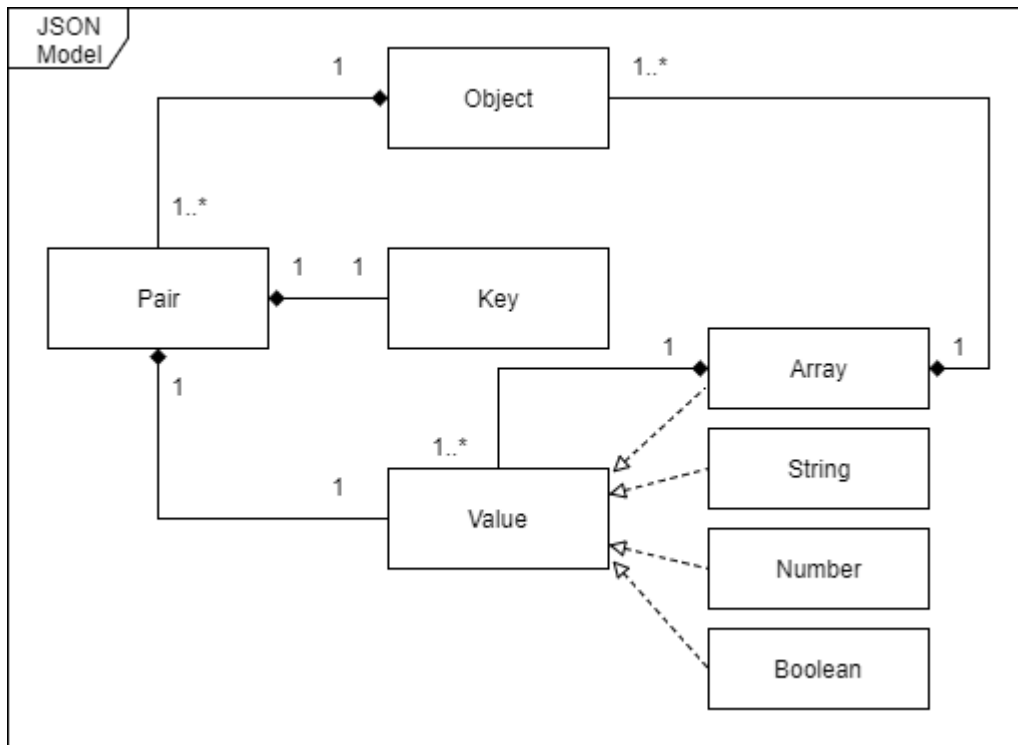


Figura 1 Representación UML Modelo JSON utilizado

En el código Ruby se tendrán las clases (ver archivo JSONModel.rb)

1. JSONModel::JObject .- Representa un Objeto JSON que contiene uno o más pares.
2. JSONModel::Pair .- Representa el Pair JSON que está compuesta por un Key y un Value. Dada la facilidad Ruby el atributo Pair.value puede usarse como arreglo, string, number o boolean, de acuerdo al caso requerido.

2.2.2.2 Parser

Para construir el Parser se parte de premisas indicadas o extraídas del formato JSON, estas son:

1. En JSON se tiene los siguientes caracteres reservados

| | |
|---|--|
| { | Abre un Objeto |
| } | Cierra un Objeto |
| : | Separa un Key de un Value |
| [| Abre un Arreglo |
|] | Cierra un Arreglo |
| , | Separa Pares u Objetos o Valores de un Arreglo |

Tabla 1 Caracteres especiales en JSON [2]

2. Las cadenas JSON válidas para este generador JSON deben siempre estar encerradas en un Objeto. Es decir, siempre debe iniciar con {.
3. Además de todas las premisas indicadas en el paso anterior con respecto al Modelo

Como se ha mencionado anteriormente, el Parser es el encargado de extraer y convertir la cadena JSON en una instancia del Modelo indicado en el párrafo anterior. El código para realizar esa generación se encuentra en el archivo JSONParser.rb

El proceso de Parsing inicia invocando a la **función getObject()**, el cual se encarga de instanciar y devolver un objeto JSONModel::JObject a partir de la cadena JSON, a continuación el detalle de dicha función:

| | |
|--|---|
| <pre>def getObject() object = JObject.new() checkChar("{") skip_spaces if current() == "}" return object end</pre> | <p>Crea un nuevo objeto, y verifica que la cadena JSON inicie con {</p> <p>Eliminar todos lo considerado como espacio en blanco</p> <p>Finalmente verifica si el siguiente caracter disponible es }, lo que significa que se transforma en un objeto vacío.</p> |
| <pre>while(1==1) key = getKey() pair = Pair.new(key) checkChar(":") @buffer.getch pair.value = getValue() object.pairList<<pair skip_spaces break if current() == "}" if current() != "," raise_error(",") end end @buffer.getch return object end</pre> | <p>Mientras no se llegue al final del Objeto, es decir, no se encuentre el caracter } haga lo siguiente:</p> <ul style="list-style-type: none"> - Obtener el correspondiente Key - Verificar que el siguiente caracter sea : - Obtener el correspondiente Value - Si el siguiente caracter es una , continuar para leer el siguiente Pair |

El siguiente paso en el proceso de Parsing es la **función getValue()**, que se encara de obtener un Pair.value de acuerdo al análisis de la cadena JSON, a continuación se describe su implementación:

| | |
|--|---|
| <pre>def getValue() skip_spaces if current() == "{" return getObject() elsif current() == "[" return getArray() else return getSingleValue() end end</pre> | <p>Eliminar los espacios en blanco. Si el siguiente caracter es un { entonces invocar a getObject(). Si el siguiente caracter es [invocar a getArray(), finalmente si no es ninguno de los casos anterior invocar a getSingleValue()</p> |
|--|---|

El siguiente paso del proceso de Parsing es la **función getArray()**, el cual verifica e instancia un arreglo de acuerdo a los datos en la cadena JSON, a continuación se describe su implementación

| | |
|--|--|
| <pre>def getArray() value = [] checkChar("[") while(1==1) @buffer.getch skip_spaces if current() == "{" value << getObject() elsif current() == "[" value << getArray() else value << getSingleValue() end skip_spaces break if current() == "]" end end</pre> | <p>Verificar que el siguiente caracter sea un [. Luego realizar un lazo, hasta encontrar el fin del arreglo, esto es el caracter]</p> <p>En el arreglo, eliminar los espacios en blanco, y si el siguiente caracter es un { invocar a getObject(), si el caracter es un [invocar a getArray(), caso contrario invocar a getSingleValue.</p> <p>Se debe anotar que el operador << permite agregar un nuevo elemento a un arreglo en Ruby.</p> |
|--|--|

Como se puede distinguir, la solución planteada se basa en el uso de recursividad, primordial para solucionar este tipo de problemas.

Un elemento final que se debe explicar, es el uso de expresiones regulares para obtener los elementos Key y Value:

- a. Para obtener un Valor Simple se usó la expresión regular

```
/((("[^"]*" )|((-)?([0-9]+.{1}\d+)|([0-9]+))|b(true|false){1}\b)/
```

Esta expresión busca

- Cualquier cadena de caracteres encerrada en comillas dobles, o
 - Un número decimal opcionalmente con una parte decimal (punto como separador), o
 - Un valor booleano (true o false)
- b. Para un obtener un Key, se utilizó la expresión regular

```
/("[^"]*" )/
```

Esta expresión busca cualquier cadena de caracteres encerrada en comillas dobles

2.2.2.3 Mecanismo de Conversión

La última parte de la solución tiene que ver con la lógica utilizada para generar la representación XML requerida.

De acuerdo a los ejemplos de generación planteada, el mecanismo de conversión funcionará de la siguiente manera:

- Los Key de cada Pair se transforman en tag XML.
- Los Value de cada Pair se transforman en el contenido del tag XML.
- En el caso de Arreglos, se generan tantos Key como valores haya en el arreglo, añadiendo un atributo id en la salida XML para indicar la secuencia o posición del elemento tag en el arreglo.

La función que realiza esta transformación es convertToXML (ver archivo JSON2XML.rb).

En la función se podrá observar que se agrega al inicio y al final de la cadena XML, el tag XML esto es, todo el resultado se encierra en `<xml>` y `</xml>`.

2.3 Alcances y limitaciones de la solución

El generador soporta los elementos Objeto, Pares, Arreglos. También considera que los KEY de un par corresponde a una cadena de caracteres encerrada en comillas dobles. El generador acepta valores simples que pueden ser números decimales (opcionalmente pueden tener una parte decimal), cadena de caracteres encerradas en comillas dobles, o valores booleanos. El generador soporta anidación (uno dentro de otro) de objetos y arreglos. El generador considera que siempre la cadena JSON (contenido del archivo) inicia con el caracter `{`.

Una limitación muy evidente es el uso de la memoria, la aplicación carga todo el archivo a memoria, y luego realiza la transformación utilizando en todo momento variables de memoria, hasta que finalmente se vuelca el resultado en un archivo de salida. Es evidente que en casos particulares de archivos JSON gigantes el proceso sería bastante ineficiente.

Una limitación evidente es el soporte a los números reales €, dentro del formato JSON [2], así como también el soporte para el valor null.

Otra limitación tiene que ver con caracteres especiales dentro del JSON al transformar a XML, como por ejemplo el uso de los caracteres `<`, `>`, `&` o el uso de `:` dentro de los Key.

Otra limitación del generador está relacionada con los prefijos de atributos XML (namespaces). Si un KEY en la cadena JSON contiene el caracter dos puntos (`:`), el XML generado no es válido ya que en XML la cadena a la izquierda de los 2 puntos del tag se considera un prefijo.

La última limitación tiene que ver con la definición string descrita en la especificación JSON [2]. Las expresiones regulares utilizadas en el generador no soportan todos los elementos que NO se consideraría un string válido en JSON.

2.4 Descripción de los casos de Prueba

A continuación, se proporciona en detalle, ejemplos de las cadenas JSON soportadas por el generador, así como de cadenas no válidas y rechazadas por el generador. Los 2 primero ejemplo expuestos corresponden a los ejemplos planteados anteriormente en la sección 2.2.1.

2.4.1 Primer Ejemplo

El primer ejemplo, es sencillo, muestra las características básicas del generador. Anidación de objetos. Arreglo simple, valores permitidos.

```

C:\Java\ws\jsonParser>more c:\temp\example01.txt
{
  "Person" : {
    "Pedro" : {
      "name" : "Pedro Rodriguez",
      "age" : 32,
      "phone" : "593 223 4445",
      "married" : true,
      "sons" : ["Peter","Luis"]
    }
  }
}

C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example01.txt c:\temp\result01

```

Figura 2 Archivo del primer ejemplo

```

C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example01.txt c:\temp\result01
<xml>
  <Person>
    <Pedro>
      <name>
        "Pedro Rodriguez"
      </name>
      <age>
        32
      </age>
      <phone>
        "593 223 4445"
      </phone>
      <married>
        true
      </married>
      <sons id="1">
        "Peter"
      </sons>
      <sons id="2">
        "Luis"
      </sons>
    </Pedro>
  </Person>
</xml>

```

Figura 3 Resultado XML para el primer ejemplo

2.4.2 Segundo Ejemplo

El segundo ejemplo es igualmente un ejemplo sencillo, pero muestra la funcionalidad asociada con arreglos. El elemento menuitem es un arreglo de objetos

```

C:\Java\ws\jsonParser>more c:\temp\example02.txt
{"menu": {
  "id": "file",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}

C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example02.txt c:\temp\result02

```

Figura 4 Archivo del segundo ejemplo

```

C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example02.txt c:\temp\result02
<xml>
  <menu>
    <id>
      "file"
    </id>
    <value>
      "File"
    </value>
    <popup>
      <menuitem id="1">
        <value>
          "New"
        </value>
        <onclick>
          "CreateNewDoc()"
        </onclick>
      </menuitem>
      <menuitem id="2">
        <value>
          "Open"
        </value>
        <onclick>
          "OpenDoc()"
        </onclick>
      </menuitem>
      <menuitem id="3">
        <value>
          "Close"
        </value>
        <onclick>
          "CloseDoc()"
        </onclick>
      </menuitem>
    </popup>
  </menu>
</xml>

```

Figura 5 Resultado XML para el segundo ejemplo

2.4.3 Tercer Ejemplo

El tercer ejemplo es mucho más complejo y su objetivo es mostrar varios niveles de anidación de objetos y arreglos. Además, muestra el soporte a números decimales, y valores booleanos. Se puede también observar la disposición de los caracteres JSON, llaves, corchetes, unos separados por espacios en blanco, y otro juntos. De la misma manera se puede observar el uso del objeto vacío {}

```

C:\Java\ws\jsonParser>more c:\temp\example03.txt
{
  "Clients": [
    {
      "Id" : "0X1666ADFE11123",
      "Name" : "John Miller",
      "Balance" : -2345.11,
      "Address" : [
        {
          "Street" : "Main Avenue", "Phone" : ["55544422", "66622211"]
        },
        {
          "Street" : "Mapple Street", "Phone" : ["77744412"]
        }
      ]
    },
    {
      "Reference" : {
        "Personal" : [
          {
            "Name": "Alice Miller", "Relation": { "isParent": true, "Relation" : "Wife" }
          },
          {
            "Name": "Robert Miller", "Relation": { "isParent": false, "Relation" : "Friend" }
          }
        ]
      }
    }
  ]
}

```

Figura 6 Archivo Tercer Ejemplo

```

C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example03.txt c:\temp\result03.txt
<xml>
  <Clients id="1">
    <Id>
      "0X1666ADFE11123"
    </Id>
    <Name>
      "John Miller"
    </Name>
    <Balance>
      -2345.11
    </Balance>
    <Address id="1">
      <Street>
        "Main Avenue"
      </Street>
    </Address>
  </Clients>
  <Address id="3">
  </Address>
  <Reference>
    <Personal id="1">
      <Name>
        "Alice Miller"
      </Name>
      <Relation>
        <isParent>
          true
        </isParent>
      </Relation>
      "Wife"
    </Personal>
  </Reference>
</xml>

```

Figura 7 Parte del resultado XML para el tercer ejemplo

Notas. - Se puede hacer uso de herramientas como [1] para validar que el XML generado sea un XML válido. En el archivo comprimido adjunto se puede encontrar los archivos de ejemplo usados y generados.

2.4.4 Cuarto Ejemplo

Se envía a generar un XML a partir de un archivo vacío. Esto resulta en un error, ya que el programa siempre espera un { como inicio.

```

C:\Java\ws\jsonParser>more c:\temp\example04.txt
C:\Java\ws\jsonParser>_

```

Figura 8 Archivo del cuarto ejemplo

```

C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example04.txt c:\temp\result04.txt
C:\Java\ws\jsonParser\JSONParser.rb:149:in 'raise_error': Formato erroneo requerido=> < encontro=> posicion=> 0
from C:\Java\ws\jsonParser\JSONParser.rb:29:in 'checkChar'
from C:\Java\ws\jsonParser\JSONParser.rb:118:in 'getObject'
from C:\Java\ws\jsonParser\JSONParser.rb:18:in 'initialize'
from JSON2XML.rb:33:in 'new'
from JSON2XML.rb:33:in 'parse'
from JSON2XML.rb:128:in '<main>'

```

Figura 9 Resultado de ejecutar el cuarto ejemplo

2.4.5 Quinto Ejemplo

Uso de caracteres no soportados, como comillas simples.

```

C:\Java\ws\jsonParser>more c:\temp\example05.txt
{
  'name' : 'prueba'
}

```

Figura 10 Archivo del quinto ejemplo

```
C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example05.txt c:\temp\result05.txt
C:/Java/ws/jsonParser/JSONParser.rb:149:in 'raise_error': Formato erroneo requerido=> " encontro=> ' posicion=> 4
from C:/Java/ws/jsonParser/JSONParser.rb:41:in 'getKey'
from C:/Java/ws/jsonParser/JSONParser.rb:127:in 'getObject'
from C:/Java/ws/jsonParser/JSONParser.rb:18:in 'initialize'
from JSON2XML.rb:33:in 'new'
from JSON2XML.rb:33:in 'parse'
from JSON2XML.rb:128:in '<main>'
```

Figura 11 Resultado de la ejecución del quinto ejemplo

2.4.6 Sexto Ejemplo

Cadena JSON mal conformada, en el siguiente ejemplo se omite el] que cierra un arreglo de valores.

```
C:\Java\ws\jsonParser>more c:\temp\example06.txt
{
  "objeto" : { "valor" : ["null"]
}
```

Figura 12 Archivo del sexto ejemplo

```
C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example06.txt c:\temp\result06.txt
C:/Java/ws/jsonParser/JSONParser.rb:149:in 'raise_error': Formato erroneo requerido=> , o l encontro=> > posicion=> 34
from C:/Java/ws/jsonParser/JSONParser.rb:70:in 'getArray'
from C:/Java/ws/jsonParser/JSONParser.rb:85:in 'getValue'
from C:/Java/ws/jsonParser/JSONParser.rb:131:in 'getObject'
from C:/Java/ws/jsonParser/JSONParser.rb:83:in 'getValue'
from C:/Java/ws/jsonParser/JSONParser.rb:131:in 'getObject'
from C:/Java/ws/jsonParser/JSONParser.rb:18:in 'initialize'
from JSON2XML.rb:33:in 'new'
from JSON2XML.rb:33:in 'parse'
from JSON2XML.rb:128:in '<main>'
C:\Java\ws\jsonParser>
```

Figura 13 Resultado de la ejecución del sexto ejemplo

2.4.7 Séptimo Ejemplo

Cadena JSON válida que únicamente contiene un conjunto de Pares.

```
C:\Java\ws\jsonParser>more c:\temp\example07.txt
{
  "id" : "199199199",
  "name" : "Federico"
}
```

Figura 14 Archivo séptimo ejemplo

```
C:\Java\ws\jsonParser>ruby -I. JSON2XML.rb c:\temp\example07.txt c:\temp\result07.txt
<xml>
<id>
  "199199199"
</id>
<name>
  "Federico"
</name>
</xml>
```

Figura 15 Resultado de la ejecución del séptimo ejemplo

3 Programas Fuentes

El archivo comprimido adjunto jsonParser.zip contiene 2 carpetas:

1. JSONParser, que corresponde al proyecto eclipse con el código fuente del generador

2. ejemplos, que contiene la lista de archivos ejemplos descritos en este documento, y como también los archivos resultado

Dentro del proyecto eclipse se encuentra 3 archivos Ruby:

1. JSONModel.rb contiene las clases que representa el Modelo JSON del generador.
2. JSONParser.rb contiene la clase Parser la cual implementa la lógica de transformación de una cadena a un objeto JSONModel::JObject
3. JSON2XML.rb contiene el programa principal, y una clase que implementa utilidades (lee y escribir archivos) y la lógica de transformación o conversión de un objeto JSONModel::JObject a una representación XML

4 Utilización

4.1 Requisitos

El único requisito para utilizar el generador es tener instalado Ruby versión 2.4.1 o superior.

4.2 Instalación

Copiar los siguientes archivos en un mismo directorio local

- JSON2XML.rb
- JSONModel.rb
- JSONParser.rb

4.3 Ejecución

Para ejecutar el generador es necesario colocar la cadena JSON que se desea transformar en un archivo texto. La extensión .txt del archivo es opcional. Se puede utilizar cualquier nombre de archivo y extensión que sea soportada por el sistema operativo.

A continuación ejemplificamos la ejecución del script Ruby, considerando que los archivo .rb se copiaron en el directorio c:\JSONParser de un máquina con sistema operativo Windows.

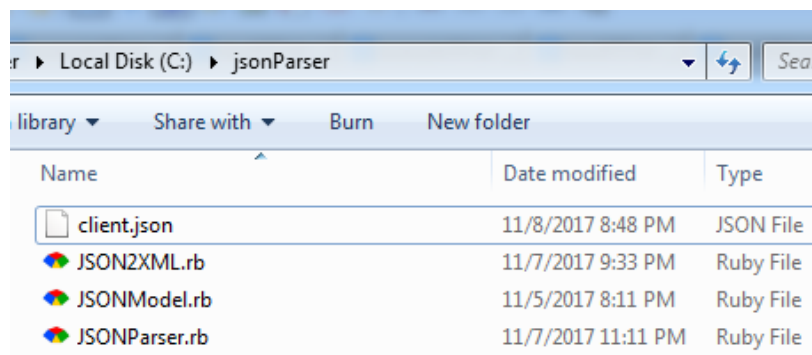
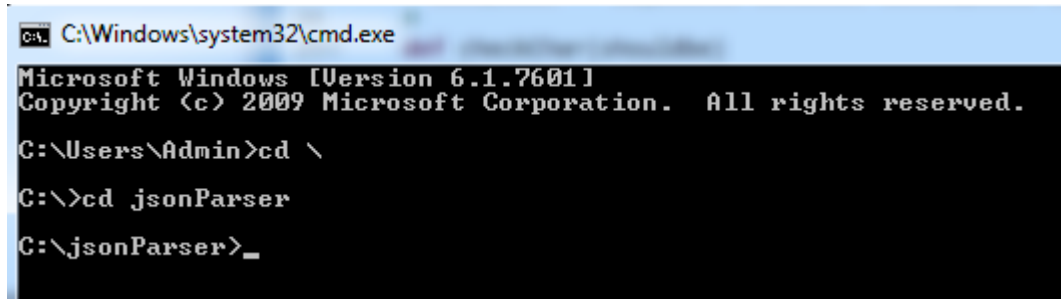


Figura 16 Archivos generador instalados en una máquina Windows

En la imagen anterior, se puede observar que además de los archivo Ruby, se encuentra un archivo client.JSON el cual quiere usarse como entrada para el generador.

A continuación, se inicia una ventana para línea de comandos (Shell) y usando comandos del sistema operativo se posiciona en el directorio de instalación, en este caso c:\JSONParser



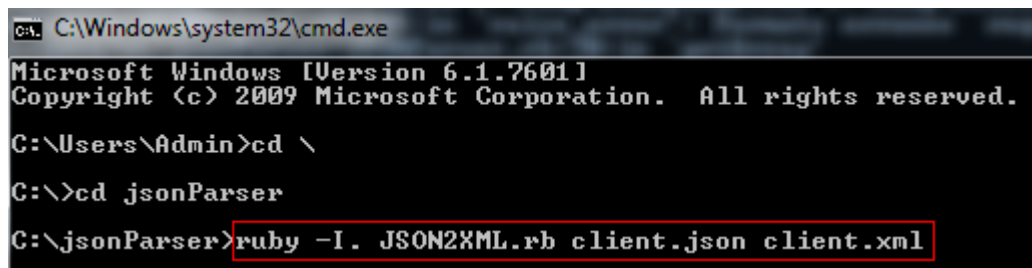
```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd \
C:\>cd jsonParser
C:\jsonParser>_
```

Figura 17 Ejecución de generador desde la línea de comandos

En la línea de comandos se debe escribir el siguiente comando:

```
ruby -I. JSON2XML.rb [inputFile] [outputFile]
```

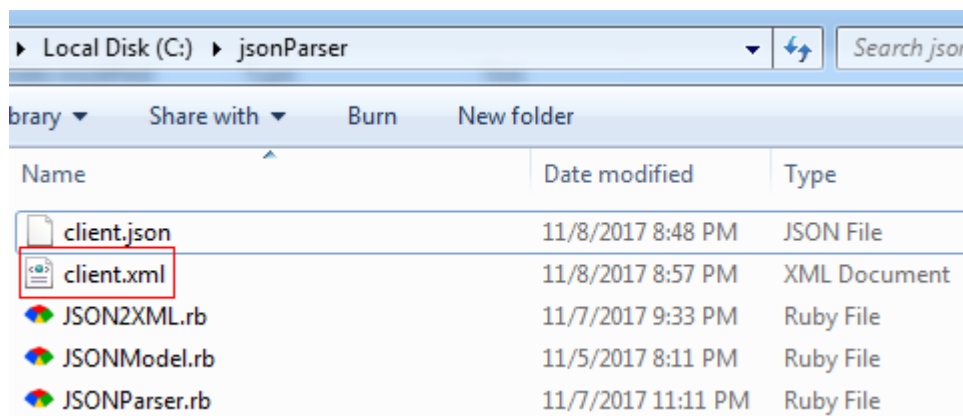


```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Admin>cd \
C:\>cd jsonParser
C:\jsonParser>ruby -I. JSON2XML.rb client.json client.xml
```

Figura 18 Ejemplo de ejecución

En este ejemplo, el archivo de entrada es client.JSON, y el archivo de salida será client.XML. Se procede a ejecutar el script, y el generador (de no existir ningún problema), mostrará en la consola el XML resultado y lo grabará el archivo de salida, en este ejemplo client.XML



| Name | Date modified | Type |
|---------------|--------------------|--------------|
| client.json | 11/8/2017 8:48 PM | JSON File |
| client.xml | 11/8/2017 8:57 PM | XML Document |
| JSON2XML.rb | 11/7/2017 9:33 PM | Ruby File |
| JSONModel.rb | 11/5/2017 8:11 PM | Ruby File |
| JSONParser.rb | 11/7/2017 11:11 PM | Ruby File |

Figura 19 Ejemplo de la creación del archivo de salida

5 Conclusiones y Reflexiones

Desde un punto de vista comercial, parecería que la opción más adecuada para resolver el problema planteado, sería utilizar JavaScript, dado que JSON es un elemento intrínseco de este lenguaje de programación. Aunque parecería obvia, el objeto de la práctica era usar los conocimientos que se obtienen de la lectura del texto base del curso [2] por lo que la opción mencionada fue descartada de inmediato.

El primer problema a resolver en la transformación era decidir la correspondencia que existirá los elementos JSON versus los elementos XML. La solución fue tomar los elementos calificados como KEY (claves) en el JSON para transformarlos en elementos TAG de XML. Pero esta premisa de transformación no soluciona casos, como por ejemplo cuando la cadena JSON no posee un KEY inicial (ver ejemplo 2.47), ya que el XML así obtenido no tendría un elemento raíz. El segundo problema a resolver tuvo que ver con los Arreglos. La decisión fue tomar el KEY correspondiente al objeto que lo contiene, y agregar un atributo XML nombrado como id, al cual se le asigna la posición que el elemento ocupaba en el arreglo.

Como menciona el libro base [2], armar un generador de este tipo en un lenguaje diferente a los indicados o sugeridos en dicho libro; logrando el mismo resultado; tomaría muchísimo más tiempo (al menos el doble). Por ende, la decisión de usar Ruby es la más adecuada para implementar soluciones a este tipo de problemas. En este punto con respecto a Ruby se puede recalcar el no uso de Tipos de Datos, o, dicho de otra manera, que una misma variable o atributo de una clase, pueda tener distintos tipos de datos (incluido arreglo) es una ventaja que facilite en mucho la solución del requerimiento planteado.

Quedan algunos puntos por mejorar en el generador, los más evidentes son:

- Permitir el uso de números reales (e), el uso del valor null.
- Mejorar el formato de salida del XML generado.
- Mejorar el mensaje que acompaña el texto de error, para dar más información sobre el mismo, como por ejemplo el número de línea.
- Leer y parsear el archivo de entrada línea a línea, para evitar un uso innecesario de la memoria RAM.
- Crear un archivo ejecutable a partir del script Ruby.

Una mejora sustancial, sería crear un generador a partir de plantillas. Estas plantillas, similar al uso de esquemas y XSLT en el mundo XML, podrían permitir transformaciones más robustas y versátiles, posiblemente se pueden implementar a través del uso de mecanismos como el propuesto por JSON Schemas [4]

6 Bibliografía y Referencias

- [1] J. Herrington, de *Code Generation In Action*, Greenwich, Manning Publications Co., 2003.
- [2] ECMA International, «Standards@Internet Speed,» 10 2013. [En línea]. Available: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>. [Último acceso: 06 11 2017].
- [3] FreeFormatter.com, «Free Online Tools For Developer,» [En línea]. Available: <https://www.freeformatter.com/xml-formatter.html#ad-output>. [Último acceso: 2017 11 07].
- [4] json-schema-org, «JSON Schema,» [En línea]. Available: <http://json-schema.org>. [Último acceso: 08 11 2017].
- [5] W3Schools, «JSON Syntax,» 21 10 2017. [En línea]. Available: https://www.w3schools.com/js/js_json_syntax.asp.

7 Índice de Figuras

| | |
|---|----|
| Figura 1 Representación UML Modelo JSON utilizado | 6 |
| Figura 2 Archivo del primer ejemplo | 10 |
| Figura 3 Resultado XML para el primer ejemplo | 10 |
| Figura 4 Archivo del segundo ejemplo | 10 |
| Figura 5 Resultado XML para el segundo ejemplo | 11 |
| Figura 6 Archivo Tercer Ejemplo | 11 |
| Figura 7 Parte del resultado XML para el tercer ejemplo | 12 |
| Figura 8 Archivo del cuarto ejemplo | 12 |
| Figura 9 Resultado de ejecutar el cuarto ejemplo | 12 |
| Figura 10 Archivo del quinto ejemplo | 12 |
| Figura 11 Resultado de la ejecución del quinto ejemplo | 13 |
| Figura 12 Archivo del sexto ejemplo | 13 |
| Figura 13 Resultado de la ejecución del sexto ejemplo | 13 |
| Figura 14 Archivo séptimo ejemplo | 13 |
| Figura 15 Resultado de la ejecución del séptimo ejemplo | 13 |
| Figura 16 Archivos generador instalados en una máquina Windows..... | 15 |
| Figura 17 Ejecución de generador desde la línea de comandos..... | 15 |

| | |
|--|----|
| Figura 18 Ejemplo de ejecución | 15 |
| Figura 19 Ejemplo de la creación del archivo de salida | 15 |