Zend_xml2json - Senthil Nathan

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_xml2json Component Proposal

Proposed Component Name	Zend_xml2json
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_xml2json
Proposers	Senthil Nathan
Revision	0.1 - 23 April 2007: Initial proposal. 0.2 - 30 May 2007: Revised proposal after incorporating the review comments. 0.3 - 31 May 2007: Another revision done to do the logic in two functions. 0.4 - 14 June 2007: Minor changes made before the first commit into incubator. (wiki revision: 37)

Table of Contents

1. Overview

This is a proposal to add a new feature to the Zend_Json module. This new feature will provide a conversion capability. Specifically, it will allow any arbitrary XML formatted document to be converted into JSON formatted string. The following paragraph forms the motivation for this proposal.

Since the late 1990s, many enterprise data architectures have been using XML. That choice made it a common practice for mid-tier (PHP and J2EE) servers to exchange enterprise data with the browser applications in XML format. However, current popularity of JSON makes it a credible data interchange format between browser applications and mid-tier servers. Combined with the rising adoption of Ajax in enterprise-grade applications, JSON is emerging as a natural choice for data encoding at both the client and middle tiers. The key reason being that there is no extra parsing required in the browser applications to interpret JSON formatted data. Because, JSON is readily consumable in browser applications as it follows the data syntax rules of JavaScript. In addition, JSON makes the job of the browser application developers a lot simpler by eliminating the need to deal with all the intricacies of XML.

As Web 2.0 technologies proliferate in large enterprises, it will become necessary for PHP-based mid-tier server applications to exchange enterprise application data with the browser applications in JSON format rather than in XML format. In order to achieve this, server-side PHP application developers have to convert XML-formatted data into JSON-formatted data before sending it to browser. In reality, it would be extremely beneficial to make this conversion as part of the Zend Framework. Hence, this proposal is made to add this simple, but powerful feature in the Zend_Json module.

2. References

- Zend_Json

- xml2json - IBM developerWorks article

3. Component Requirements, Constraints, and Acceptance Criteria

- This component **will** require a new static function to be added to Json.php
- This component **will not** require use of configuration files.
- This component **will** add a new class to Zend_Json module.
- Given a XML formatted string as input, this component **will** convert the entire XML content into JSON formatted string.
- This component **will** keep the nested structure of the XML content intact in the resulting JSON data as well.
- This component **will** also optionally convert the XML attributes present in the XML content into a corresponding representation in the resulting JSON data.

4. Dependencies on Other Framework Components

- Zend_Json
- Zend_Exception

5. Theory of Operation

In order to bring the xml2json conversion feature to the Zend Framework, we will add two new static functions to the Zend_Json class in Json.php file. Those functions are described below.

- fromXml
- _processXml

A function named "fromXml" is the one that users will call to convert any arbitrary XML content to JSON data. This function will take as input, a string containing XML formatted content and an optional boolean value of true to indicate the need to ignore or false for NOT to ignore the XML attributes in the conversion process. This function will then convert this input XML string value into SimpleXMLElement. Then, it will call another worker function (_processXml) in the same class to convert the XML elements into a PHP associative array. Once the PHP array is created, it simply uses the Zend_Json "encode" function to convert that PHP array into a JSON formatted string. Result from this conversion will be returned to the caller. In the case of any input error, this function will throw a Zend_Json_Exception.

A worker function named "_processXml" will provide a recursive logic to do the conversion of an XML tree into a PHP associative array. This function takes three parameters. First parameter is the SimpleXMLElement object that contains an XML fragment to be converted into a PHP array. The second parameter is a boolean value that determines if XML attributes need to be preserved in the conversion process. By default, this function will ignore the XML attributes i.e. it will not do anything with the XML attributes during the conversion process. The third parameter specifies the current recursion depth which is used by the internal logic in this function. It recursively traverses the nested XML content structure and stores the visited XML contents into a PHP associative array structure. At the end of the XML tree traversal, it returns the PHP array with all the element/value pairs that were present in the XML input content. If needed, this function will also optionally store the XML attributes in the PHP array so that it can be preserved in the final JSON data. In case of a recursion runtime error, this function will throw a Zend_Json_Exception.

In addition to the above, this conversion logic will also require the following in the Zend_Json class.

- require_once 'Zend/Json/Exception.php'; // It is needed for throwing exceptions whenever there is an error.
- public static $maxRecursionDepthAllowed=25; // It is needed to check the allowed nesting depth of the XML tree.

6. Milestones / Tasks

- Milestone 1: [DONE] Publish this proposal.
- Milestone 2: [DONE] Revise proposal, approve for Incubator development.
- Milestone 3: Commit working prototype to Incubator.
- Milestone 4: Commit unit tests and documentation.
- Milestone 5: Promote code from incubator to framework release.

7. Class Index

- Zend_Json

8. Use Cases

The use cases below show a few scenarios that explain the input and output characteristics of Zend_xml2json feature.

UC-01

```
<?xml version="1.0" encoding="UTF-8"?>
<contacts>
    <contact id="1">
        <name>John Doe</name>
        <phone>123-456-7890</phone>
        <address>
            <street>123 JFKStreet</street>
            <city>Any Town</city>
            <state>Any State</state>
            <zipCode>12345</zipCode>
        </address>
    </contact>
</contacts>
```

The XML content shown above defines a parent element <contacts> that can include one or more <contact> child element. Every <contact> child element will have a nested structure of its own children. When this XML content is passed as an input to the xml2json conversion function, the expected output in JSON format is as shown below.

```
{
   "contacts" : {
      "contact" : {
         "@attributes" : {
            "id" : "1"
         },
         "name" : "John Doe",
         "phone" : "123-456-7890",
         "address" : {
            "street" : "123 JFK Street",
            "city" : "Any Town",
            "state" : "Any State",
            "zipCode" : "12345"
         }
      }
   }
}
```

As you can see from the JSON formatted output shown above, each piece of element information found in the XML input content is preserved in the JSON formatted output content resulting from xml2json conversion. Optionally, xml2json function can be made to carry over the XML attributes into the JSON output.

UC-02

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book id="1">
        <title>Code Generation in Action</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>Manning</publisher>
    </book>

    <book id="2">
        <title>PHP Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>

    <book id="3">
        <title>Podcasting Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>
</books>
```

XML input content shown above contains a parent element <books> which in turn has repeating <book> child elements. The <book> element contains an XML attribute to define an id for that book. Each <book> element also contains other child elements. This XML is moderately complex as compared to the one we saw in Use Case 1. The JSON formatted text shown below is the result/output generated by the xml2json conversion function.

```
{
   "books" : {
      "book" : [ {
         "@attributes" : {
            "id" : "1"
         },
         "title" : "Code Generation in Action",
         "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "Manning"
      }, {
         "@attributes" : {
            "id" : "2"
         },
         "title" : "PHP Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }, {
         "@attributes" : {
            "id" : "3"
         },
         "title" : "Podcasting Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }
   ]}
}
```

The JSON representation shown above preserves all the content structure and details as in the XML input. The repeating <book> elements in XML are converted into a JSON array of book objects. If the conversion function is called with an option to retain the XML attributes, every JSON book object will carry the attributes in a @attributes property.

UC-03

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseRequisition>
 <Submittor>
  <SubmittorName>John Doe</SubmittorName>
  <SubmittorEmail>john@nodomain.net</SubmittorEmail>
  <SubmittorTelephone>1-123-456-7890</SubmittorTelephone>
 </Submittor>
 <Billing/>
 <Approval/>
 <Item number="1">
  <ItemType>Electronic Component</ItemType>
  <ItemDescription>25 microfarad 16 volt surface-mount tantalum
capacitor</ItemDescription>
  <ItemQuantity>42</ItemQuantity>
  <Specification>
   <Category type="UNSPSC" value="32121501" name="Fixed capacitors"/>
   <RosettaNetSpecification>
    <query max.records="1">
```

```
     <element dicRef="XJA039">
      <name>CAPACITOR - FIXED - TANTAL - SOLID</name>
     </element>
     <element>
      <name>Specific Features</name>
      <value>R</value>
     </element>
     <element>
      <name>Body Material</name>
      <value>C</value>
     </element>
     <element>
      <name>Terminal Position</name>
      <value>A</value>
     </element>
     <element>
      <name>Package: Outline Style</name>
      <value>CP</value>
     </element>
     <element>
      <name>Lead Form</name>
      <value>D</value>
     </element>
     <element>
      <name>Rated Capacitance</name>
      <value>0.000025</value>
     </element>
     <element>
      <name>Tolerance On Rated Capacitance (%)</name>
      <value>10</value>
     </element>
     <element>
      <name>Leakage Current (Short Term)</name>
      <value>0.0000001</value>
     </element>
     <element>
      <name>Rated Voltage</name>
      <value>16</value>
     </element>
     <element>
      <name>Operating Temperature</name>
      <value type="max">140</value>
      <value type="min">-10</value>
     </element>
     <element>
      <name>Mounting</name>
      <value>Surface</value>
     </element>
    </query>
   </RosettaNetSpecification>
  </Specification>
  <Vendor number="1">
           <VendorName>Capacitors 'R' Us, Inc.</VendorName>
           <VendorIdentifier>98-765-4321</VendorIdentifier>

<VendorImplementation>http://acme.com/capaciorsRus/wsdl/buyerseller-implementation.wsdl<
</Vendor>
```

```
 </Item>
</PurchaseRequisition>
```

XML input content shown above represents an example of a purchase order that follows the RosettaNet industry standard. It is sufficiently complex in terms of structure, nesting, empty elements, multiple XML attributes etc. This XML input content, when used as input to the xml2json function will produce the resulting JSON formatted text as shown below.


```
{"PurchaseRequisition":{
    "Submittor":{
        "SubmittorName":"John Doe",
        "SubmittorEmail":"john@nodomain.net",
        "SubmittorTelephone":"1-123-456-7890"},
    "Billing":"",
    "Approval":"",
    "Item":{"@attributes":{"number":"1"},
        "ItemType":"Electronic Component",
        "ItemDescription":"25 microfarad 16 volt surface-mount tantalum capacitor",
        "ItemQuantity":"42",
        "Specification":{
            "Category":{
                "@attributes":{"type":"UNSPSC","value":"32121501","name":"Fixed
capacitors"}
            },
            "RosettaNetSpecification":{
                "query":{
                    "@attributes":{"max.records":"1"},
                    "element":[
                        {
                            "@attributes":{"dicRef":"XJA039"},
                            "name":"CAPACITOR - FIXED - TANTAL - SOLID"},
                        {
                            "name":"Specific Features",
                            "value":"R"},
                        {
                            "name":"Body Material",
                            "value":"C"},
                        {
                            "name":"Terminal Position",
                            "value":"A"},
                        {
                            "name":"Package: Outline Style",
                            "value":"CP"},
                        {
                            "name":"Lead Form",
                            "value":"D"},
                        {
                            "name":"Rated Capacitance",
                            "value":"0.000025"},
                        {
                            "name":"Tolerance On Rated Capacitance (%)",
                            "value":"10"},
                        {
                            "name":"Leakage Current (Short Term)",
                            "value":"0.0000001"},
                        {
                            "name":"Rated Voltage",
```

```
                            "value":"16"},
                        {
                            "name":"Operating Temperature",
                            "value":["140","-10"]},
                        {
                            "name":"Mounting",
                            "value":"Surface"}
                    ]}
                }
            },
        "Vendor":{
            "@attributes":{"number":"1"},
            "VendorName":"Capacitors 'R' Us, Inc.",
            "VendorIdentifier":"98-765-4321",

"VendorImplementation":"http:\/\/acme.com\/capaciorsRus\/wsdl\/buyerseller-implementatio
```

```
}
}
```

As we observed in use cases 1 and 2, the above-mentioned JSON formatted result preserves the content structure as laid out in the XML input. Note that the empty element <Billing /> has been retained as a JSON property with a value of an empty string. Multiple XML attributes present in the <Category> element have been carried over in the corresponding JSON data. All the repeating child elements of <query> also resulted in a JSON array of element objects.

UC-04

```
<?xml version="1.0"?>
<tvshows>
    <show>
        <name>The Simpsons</name>
    </show>

    <show>
        <name><![CDATA[Lois & Clark]]]><![CDATA[></name>
    </show>
</tvshows>
```

The XML content shown above defines a simple tree of different TV shows with repeating elements. One thing that stands out in this XML content is the CDATA section, which can contain any arbitrary general character data. CDATA is useful to express XML fragment as text data within an XML document. In the XML content above, CDATA section contains only a string with an ampersand character which is a special character in XML. When this XML content is passed as an input to the xml2json conversion function, the expected output in JSON format is as shown below.

```
{
   "tvshows" : {
      "show":
      [
         {"name":"The Simpsons"},
         {"name":"Lois & Clark"}
      ]
   }
}
```

As you can see from the JSON formatted output shown above, the text data inside of the CDATA section is preserved in the conversion. It is taken care of by the conversion logic.

UC-05

```
<?xml version="1.0"?>
<demo>
    <application>
        <name>Killer Demo</name>
    </application>

    <author>
        <name>John Doe</name>
    </author>

    <platform>
        <name>LAMP</name>
    </platform>

    <framework>
        <name>Zend</name>
    </framework>

    <language>
        <name>PHP</name>
    </language>

    <listing>
        <code>
            <![CDATA[
<?php
include 'example.php';
$xml = new SimpleXMLElement($xmlstr);

$character = $xml->movie[0]->characters->addChild('character');
$character->addChild('name', "Mr. Parser");
$character->addChild('actor', "John Doe");
// Add it as a child element.
$rating = $xml->movie[0]->addChild('rating', 'PG');
$rating->addAttribute("type", 'mpaa');

echo $xml->asXML();
?>
            ]]]><![CDATA[>
        </code>
    </listing>
</demo>
```

The XML content shown above defines a simple hierarchy of elements that define a structure for a demo application. The interesting part comes at the end of the document where the CDATA section is specified. The CDATA section includes a complete listing of a PHP script. It is important to note that the contents inside of this CDATA section include several reserved characters such as double quote, greater than character, forward slashes etc. This is a good test case for the xml2json conversion function. When this XML content is passed as an input to the xml2json conversion function, the expected output in JSON format is as shown below.

```
{
   "demo":{
      "application":
         {"name":"Killer Demo"},
      "author":
         {"name":"John Doe"},
      "platform":
         {"name":"LAMP"},
      "framework":
         {"name":"Zend"},
      "language":
         {"name":"PHP"},
      "listing":
         {"code":
"<?php\n
include 'example.php';\n
$xml = new SimpleXMLElement($xmlstr);\n\n
$character = $xml->movie[0]->characters->addChild('character');\n
$character->addChild('name', \"Mr. Parser\");\n
$character->addChild('actor', \"John Doe\");\n\n
\/\/ Add it as a child element.\n
$rating = $xml->movie[0]->addChild('rating', 'PG');\n
$rating->addAttribute(\"type\", 'mpaa');\n\n
echo $xml->asXML();\n
?>"
         }
   }
}
```

As you can see from the JSON formatted output shown above, the conversion logic preserved the entire text (PHP script in this case) that was part of the CDATA section in the input XML. It is also interesting to note that the entire CDATA section content is converted into a single string. Some of the special characters are automatically escaped by preceding such characters with a backslash character. This powerful feature to handle CDATA sections is built into the xml2json conversion logic.

All five use cases combined together demonstrate the potential use of the xml2json conversion function named fromXml. Even though this function looks deceptively simple, the benefit it will deliver to the PHP user community could be very valuable.

9. Class Skeletons

As explained earlier, two new static functions will be added to the Zend_Json class in Json.php file. The first function is a static function that provides public access to the framework users. This function front-ends the conversion logic by providing a simple call interface. This static function with added comments is shown below.

```
   /**
     * fromXml - Converts XML to JSON
     *
     * Converts a XML formatted string into a JSON formatted string.
     * The value returned will be a string in JSON format.
     *
     * The caller of this function needs to provide only the first parameter,
     * which is an XML formatted String. The second parameter is optional, which
     * lets the user to select if the XML attributes in the input XML string
     * should be included or ignored in xml2json conversion.
     *
     * This function converts the XML formatted string into a PHP array by
     * calling a recursive (protected static) function in this class. Then, it
     * converts that PHP array into JSON by calling the "encode" static funcion.
     *
     * Throws a Zend_Json_Exception if the input not a XML formatted string.
     *
     * @static
     * @access public
     * @param string $xmlStringContents XML String to be converted
     * @param boolean $ignoreXmlAttributes Include or exclude XML attributes in
     * the xml2json conversion process.
     * @return mixed - JSON formatted string on success
     * @throws Zend_Json_Exception
     */
    public static function fromXml ($xmlStringContents, $ignoreXmlAttributes=true) {

        // Load the XML formatted string into a Simple XML Element object.
        $simpleXmlElementObject = simplexml_load_string($xmlStringContents);

        // If it is not a valid XML content, throw an exception.
        if ($simpleXmlElementObject == null) {
            throw new Zend_Json_Exception('Function fromXml was called with an invalid
XML formatted string.');
        } // End of if ($simpleXmlElementObject == null)

        $resultArray = null;

        // Call the recursive function to convert the XML into a PHP array.
        $resultArray = self::_processXml($simpleXmlElementObject,
$ignoreXmlAttributes);

        // Convert the PHP array to JSON using Zend_Json encode method.
        // It is just that simple.
        $jsonStringOutput = self::encode($resultArray);
        return($jsonStringOutput);
    } // End of function fromXml.
```

The second function is a static function that provides protected access to the callers. Only the front-end function "fromXml" discussed above needs access to the logic inside this protected function. The xml2json conversion logic is a recursive one. This static function with added comments is shown below.

```
    /**
     * _processXml - Contains the logic for xml2json
     *
```

```
     * The logic in this function is a recursive one.
     *
     * The main caller of this function (i.e. fromXml) needs to provide
     * only the first two parameters i.e. the SimpleXMLElement object and
     * the flag for ignoring or not ignoring XML attributes. The third parameter
     * will be used internally within this function during the recursive calls.
     *
     * This function converts the SimpleXMLElement object into a PHP array by
     * calling a recursive (protected static) function in this class. Once all
     * the XML elements are stored in the PHP array, it is returned to the caller.
     *
     * Throws a Zend_Json_Exception if the XML tree is deeper than the allowed limit.

     *
     * @static
     * @access protected
     * @param SimpleXMLElement $simpleXmlElementObject XML element to be converted
     * @param boolean $ignoreXmlAttributes Include or exclude XML attributes in
     * the xml2json conversion process.
     * @param int $recursionDepth Current recursion depth of this function
     * @return mixed - On success, a PHP associative array of traversed XML elements
     * @throws Zend_Json_Exception
     */
    protected static function _processXml ($simpleXmlElementObject,
$ignoreXmlAttributes, $recursionDepth=0) {
        // Keep an eye on how deeply we are involved in recursion.
        if ($recursionDepth > self::$maxRecursionDepthAllowed) {
            // XML tree is too deep. Exit now by throwing an exception.
            throw new Zend_Json_Exception(
                "Function _processXml exceeded the allowed recursion depth of " .
                self::$maxRecursionDepthAllowed);
        } // End of if ($recursionDepth > self::$maxRecursionDepthAllowed)

        if ($recursionDepth == 0) {
            // Store the original SimpleXmlElementObject sent by the caller.
            // We will need it at the very end when we return from here for good.

            $callerProvidedSimpleXmlElementObject = $simpleXmlElementObject;
        } // End of if ($recursionDepth == 0)

        if ($simpleXmlElementObject instanceof SimpleXMLElement) {
            // Get a copy of the simpleXmlElementObject
            $copyOfSimpleXmlElementObject = $simpleXmlElementObject;
            // Get the object variables in the SimpleXmlElement object for us to
iterate.
            $simpleXmlElementObject = get_object_vars($simpleXmlElementObject);
        } // End of if (get_class($simpleXmlElementObject) == "SimpleXMLElement")

        // It needs to be an array of object variables.
        if (is_array($simpleXmlElementObject)) {
            // Initialize a result array.
            $resultArray = array();
            // Is the input array size 0? Then, we reached the rare CDATA text if any.

            if (count($simpleXmlElementObject) <= 0) {
                // Let us return the lonely CDATA. It could even be
                // an empty element or just filled with whitespaces.
                return (trim(strval($copyOfSimpleXmlElementObject)));
            } // End of if (count($simpleXmlElementObject) <= 0)
```

```
            // Let us walk through the child elements now.
            foreach($simpleXmlElementObject as $key=>$value) {
                // Check if we need to ignore the XML attributes.
                // If yes, you can skip processing the XML attributes.
                // Otherwise, add the XML attributes to the result array.
                if(($ignoreXmlAttributes == true) && (is_string($key)) && ($key ==
"@attributes")) {
                    continue;
                } // End of if(($ignoreXmlAttributes == true) && ($key ==
"@attributes"))

                // Let us recursively process the current XML element we just visited.

                // Increase the recursion depth by one.
                $recursionDepth++;
                $resultArray[$key] = self::_processXml ($value, $ignoreXmlAttributes,
$recursionDepth);

                // Decrease the recursion depth by one.
                $recursionDepth--;
            } // End of foreach($simpleXmlElementObject as $key=>$value) {

            if ($recursionDepth == 0) {
                // That is it. We are heading to the exit now.
                // Set the XML root element name as the root [top-level] key of
                // the associative array that we are going to return to the original

                // caller of this recursive function.
                $tempArray = $resultArray;
                $resultArray = array();
                $resultArray[$callerProvidedSimpleXmlElementObject->getName()] =
$tempArray;
            } // End of if ($recursionDepth == 0)

            return($resultArray);
        } else {
            // We are now looking at either the XML attribute text or
            // the text between the XML tags.
            return (trim(strval($simpleXmlElementObject)));
```

```
        } // End of if (is_array($simpleXmlElementObject))
    } // End of function _processXml.
```

The following is a test driver that can be used to exercise the xml2json feature in the Zend framework. This PHP file (Zend_xml2json_test.php) can be run with a command-line argument of any valid filename that contains XML contents.

```
/**
 * Zend Framework
 *
 * This source file provides a test driver for testing the xml2json feature of
Zend_Json
 * This PHP script will take an XML filename as the command line argument. It can also
 * take an optional second argument to indicate whether the XML attributes in the XML
file
 * should be ignored or not ignored during xml2json conversion.
 * A value of 1 means ignore XML attributes and a 0 means not to ignore XML attribtes.
 * If the optional second argument is omitted, then the XML attributes will be
ignored.
 * It uses the fromXml function in the Zend_Json class to convert the XML string into
a JSON string.
 *
 * @category   Zend
 * @package    Zend_Json
 * @copyright  Copyright (c) 2005-2007 Zend Technologies USA Inc.
(http://www.zend.com)
 * @license    http://framework.zend.com/license/new-bsd     New BSD License
 */
<?php
    // Change this to match your Zend Framework directories.
    set_include_path( 'c:\\temp\\ZendFramework-trunk\\incubator\\library' .
PATH_SEPARATOR
              . 'c:\\temp\\ZendFramework-trunk\\library' . PATH_SEPARATOR
              . get_include_path());

    require_once("Zend/Json.php");

    // Filename from where XML contents are to be read.
    $testXmlFile = "";
    $ignoreXmlAttributes = true;

    // Read the filename from the command line.
    if ($argc <= 1) {
        print("Please provide the XML filename as a command-line argument:\n");
        print("\tphp -f Zend_xml2json_test.php test1.xml\n");
        print("You can also provide an optional second argument as 1 or 0. Default is
1.\n");
        print("1 means ignore XML atrributes in xml2json conversion. 0 means don't
ignore XML attributes.\n");
        print("\tphp -f Zend_xml2json_test.php test1.xml 1\n");
        return;
    } else {
        // Get the XML filename from the command line argument.
        $testXmlFile = $argv[1];

        // Get the optional argument to ingore or not ignore the XML attributes.
        if (($argc == 3) && ($argv[2] == "0")) {
```

```
        $ignoreXmlAttributes = false;
    } // End of if (($argc == 3) && ($argv[2] == "0"))
} // End of if ($argc <= 1)

//Read the XML contents from the input file.
file_exists($testXmlFile) or die('Could not find file ' . $testXmlFile);
$xmlStringContents = file_get_contents($testXmlFile);

$jsonContents = "";
// Convert it to JSON now.
// fromXml function simply takes a String containing XML contents as input.
$jsonContents = Zend_Json::fromXml($xmlStringContents, $ignoreXmlAttributes);

echo("JSON formatted output generated by Zend_Json::fromXml\n\n");
```

```
    echo($jsonContents);
?>
```

]]></ac:plain-text-body></ac:macro>
]]></ac:plain-text-body></ac:macro>