IBM

developerWorks.

Practically Groovy: Building, parsing, and slurping XML

Effortless XML manipulation

Skill Level: Introductory

Scott Davis (scott@thirstyhead.com)
Founder
ThirstyHead.com

19 May 2009

Learn how easy it is to slice and dice XML using Groovy. In this installment of *Practically Groovy*, author Scott Davis shows that whether you're creating XML with `MarkupBuilder` and `StreamingMarkupBuilder`, or parsing XML with `XmlParser` and `XmlSlurper`, Groovy offers a set of compelling tools for dealing with this ubiquitous data format.

It feels like XML has been around forever. In fact, XML celebrated its tenth anniversary in 2008 (see Resources). With the Java™ language predating XML by only a couple of years, one could argue that for Java developers, XML *has* been around forever.

Java language inventor Sun Microsystems has been a big supporter of XML since the very beginning. After all, XML's promise of platform independence fits nicely with the Java language's "write once, run anywhere" message. Given the shared sensibilities of the two technologies, you'd think that the Java language and XML would get along better than they do. In fact, parsing and generating XML in the Java language feels oddly foreign and convoluted.

About this series

Groovy is a modern programming language that runs on the Java platform. It offers seamless integration with existing Java code while introducing dramatic new features like closures and metaprogramming. Put simply, Groovy is what the Java language would look like had it been written in the 21st century.

© Copyright IBM Corporation 2009. All rights reserved.

The key to incorporating any new tool into your development toolkit is knowing when to use it and when to leave it in the box. Groovy can be extremely powerful, but only when applied properly to appropriate scenarios. To that end, the *Practically Groovy* series explores the practical uses of Groovy, helping you learn when and how to apply them successfully.

Happily, Groovy introduces new and better ways to create and process XML. With the help of some examples (all available for download), this article shows you how Groovy makes building and parsing XML refreshingly simple.

Comparing Java and Groovy XML parsing

At the end of "Reaching for Each," I introduced the simple XML document shown in Listing 1. (I've added the `type` attribute this time to make things slightly more interesting.)

Listing 1. An XML document listing languages I know

```
<langs type="current">
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
```

Parsing this trivial XML document is decidedly nontrivial in the Java language, as you can see in Listing 2. It takes 30 lines of code to parse a five-line XML file.

Listing 2. Parsing an XML file in Java

```
import org.xml.sax.SAXException;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.IOException;

public class ParseXml {
  public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse("src/languages.xml");

      //print the "type" attribute
      Element langs = doc.getDocumentElement();
      System.out.println("type = " + langs.getAttribute("type"));

      //print the "language" elements
      NodeList list = langs.getElementsByTagName("language");
      for(int i = 0 ; i < list.getLength();i++) {
        Element language = (Element) list.item(i);
        System.out.println(language.getTextContent());
      }
    }catch(ParserConfigurationException pce) {
```


© Copyright IBM Corporation 2009. All rights reserved.

```
        pce.printStackTrace();
      }catch(SAXException se) {
        se.printStackTrace();
      }catch(IOException ioe) {
        ioe.printStackTrace();
      }
    }
}
```

Compare the Java code in Listing 2 with the corresponding Groovy code in Listing 3:

Listing 3. Parsing an XML in Groovy

```
def langs = new XmlParser().parse("languages.xml")
println "type = ${langs.attribute("type")}"
langs.language.each{
  println it.text()
}

//output:
type = current
Java
Groovy
JavaScript
```

The best part about the Groovy code is not that it is significantly shorter than the equivalent Java code — although using five lines of Groovy to parse five lines of XML is a clear win. The thing I like most about the Groovy code is that it is far more expressive. When I write `langs.language.each`, it feels like I am working directly with the XML. In the Java version, you can't see the XML anymore.

String variables and XML

The benefits of working with XML in Groovy become even more evident when you store the XML in a `String` variable instead of a file. Groovy's triple quotes (commonly called a `HereDoc` in other languages) make it effortless to store the XML internally, as shown in Listing 4. The only change from the Groovy example in Listing 3 is flipping the `XmlParser` method call from `parse()` (which handles `File`s, `InputStreams`s, `Reader`s, and `URI`s) to `parseText()`.

Listing 4. Storing XML internally in Groovy

```
def xml = """
<langs type="current">
  <language>Java</language>
  <language>Groovy</language>
  <language>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println "type = ${langs.attribute("type")}"
```


© Copyright IBM Corporation 2009. All rights reserved.

```
langs.language.each{
  println it.text()
}
```

Notice that the triple quotes handle the multiline XML document with ease. The `xml` variable is truly a plain-old `java.lang.String` — you can add `println xml.class` to verify this yourself. The triple quotes also handle the internal quotes of `type="current"` without forcing you to escape them manually with a backslash character as you would in Java code.

Contrast the simple elegance of the Groovy code in Listing 4 with the corresponding Java code in Listing 5:

Listing 5. Storing XML internally in Java code

```
import org.xml.sax.SAXException;
import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.*;

public class ParseXmlFromString {
  public static void main(String[] args) {
    String xml = "<langs type=\"current\">\n" +
            "  <language>Java</language>\n" +
            "  <language>Groovy</language>\n" +
            "  <language>JavaScript</language>\n" +
            "</langs>";

    byte[] xmlBytes = xml.getBytes();
    InputStream is = new ByteArrayInputStream(xmlBytes);

    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.parse(is);

      //print the "type" attribute
      Element langs = doc.getDocumentElement();
      System.out.println("type = " + langs.getAttribute("type"));

      //print the "language" elements
      NodeList list = langs.getElementsByTagName("language");
      for(int i = 0 ; i < list.getLength();i++) {
        Element language = (Element) list.item(i);
        System.out.println(language.getTextContent());
      }
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    }catch(SAXException se) {
      se.printStackTrace();
    }catch(IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

Notice that the `xml` variable is polluted with escape characters for the internal quotes and newlines. More troubling, however, is having to convert the `String` to a `byte` array, and then again to a `ByteArrayInputStream`, before it can be parsed.

© Copyright IBM Corporation 2009. All rights reserved.

Oddly, the `DocumentBuilder` doesn't provide a straightforward way to parse a simple `String` as XML.

Creating XML with MarkupBuilder

The biggest win for Groovy over the Java language comes when you want to create an XML document in code. Listing 6 shows the 50 lines of Java code required to create the five-line XML snippet:

Listing 6. Creating XML with Java code

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;

public class CreateXml {
  public static void main(String[] args) {
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    try {
      DocumentBuilder db = dbf.newDocumentBuilder();
      Document doc = db.newDocument();

      Element langs = doc.createElement("langs");
      langs.setAttribute("type", "current");
      doc.appendChild(langs);

      Element language1 = doc.createElement("language");
      Text text1 = doc.createTextNode("Java");
      language1.appendChild(text1);
      langs.appendChild(language1);

      Element language2 = doc.createElement("language");
      Text text2 = doc.createTextNode("Groovy");
      language2.appendChild(text2);
      langs.appendChild(language2);

      Element language3 = doc.createElement("language");
      Text text3 = doc.createTextNode("JavaScript");
      language3.appendChild(text3);
      langs.appendChild(language3);

      // Output the XML
      TransformerFactory tf = TransformerFactory.newInstance();
      Transformer transformer = tf.newTransformer();
      transformer.setOutputProperty(OutputKeys.INDENT, "yes");
      StringWriter sw = new StringWriter();
      StreamResult sr = new StreamResult(sw);
      DOMSource source = new DOMSource(doc);
      transformer.transform(source, sr);
      String xmlString = sw.toString();
      System.out.println(xmlString);
    }catch(ParserConfigurationException pce) {
      pce.printStackTrace();
    } catch (TransformerConfigurationException e) {
      e.printStackTrace();
    } catch (TransformerException e) {
      e.printStackTrace();
    }
```


© Copyright IBM Corporation 2009. All rights reserved.

```
  }
}
```

I know that some of you are crying, "Foul!" right now. Plenty of third-party libraries can make this code more straightforward — JDOM and dom4j are two popular ones. But none of the Java libraries comes close to the simplicity of using a Groovy `MarkupBuilder`, as demonstrated in Listing 7:

Listing 7. Creating XML with Groovy

```
def xml = new groovy.xml.MarkupBuilder()
xml.langs(type:"current"){
  language("Java")
  language("Groovy")
  language("JavaScript")
}
```

Notice that we are back to the nearly 1:1 ratio of code to XML. More important, I can see the XML again. Oh sure, the pointy braces are replaced by curly-braced closures, and the attributes use colons (Groovy's `HashMap` notation) instead of equals signs, but the basic structure is recognizable in either Groovy or XML. It's almost like a DSL for building XML, don't you think?

Groovy is able to accomplish this `Builder` magic because it is a dynamic language. The Java language, on the other hand, is static: the Java compiler ensures that all methods exist before you can call them. (Java code won't even compile, let alone run, if you try to call a nonexistent method.) But Groovy's `Builder` demonstrates that one language's bug is another language's feature. If you check the API docs for `MarkupBuilder`, you'll find no `langs()` method, `language()` method, or any other element name. Luckily, Groovy can catch these calls to methods that don't exist and do something productive with them. In the case of a `MarkupBuilder`, it takes the phantom method calls and generates well-formed XML.

Listing 8 expands on the simple `MarkupBuilder` example I just showed you. If you want to capture the XML output in a `String` variable, pass a `StringWriter` into the constructor of the `MarkupBuilder`. If you want to add more attributes to `langs`, simply pass more in separated by commas. Notice that the `language` element's body is a value without a name in front. You can add attributes and body in the same comma-delimited list.

Listing 8. An expanded MarkupBuilder example

```
def sw = new StringWriter()
def xml = new groovy.xml.MarkupBuilder(sw)
xml.langs(type:"current", count:3, mainstream:true){
  language(flavor:"static", version:"1.5", "Java")
  language(flavor:"dynamic", version:"1.6.0", "Groovy")
  language(flavor:"dynamic", version:"1.9", "JavaScript")
}
```


© Copyright IBM Corporation 2009. All rights reserved.

```
println sw

//output:
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
```

With these few `MarkupBuilder` tricks under your belt, you can take things in interesting directions. For example, you can quickly build a well-formed HTML document and write it out to a file. Listing 9 shows the code:

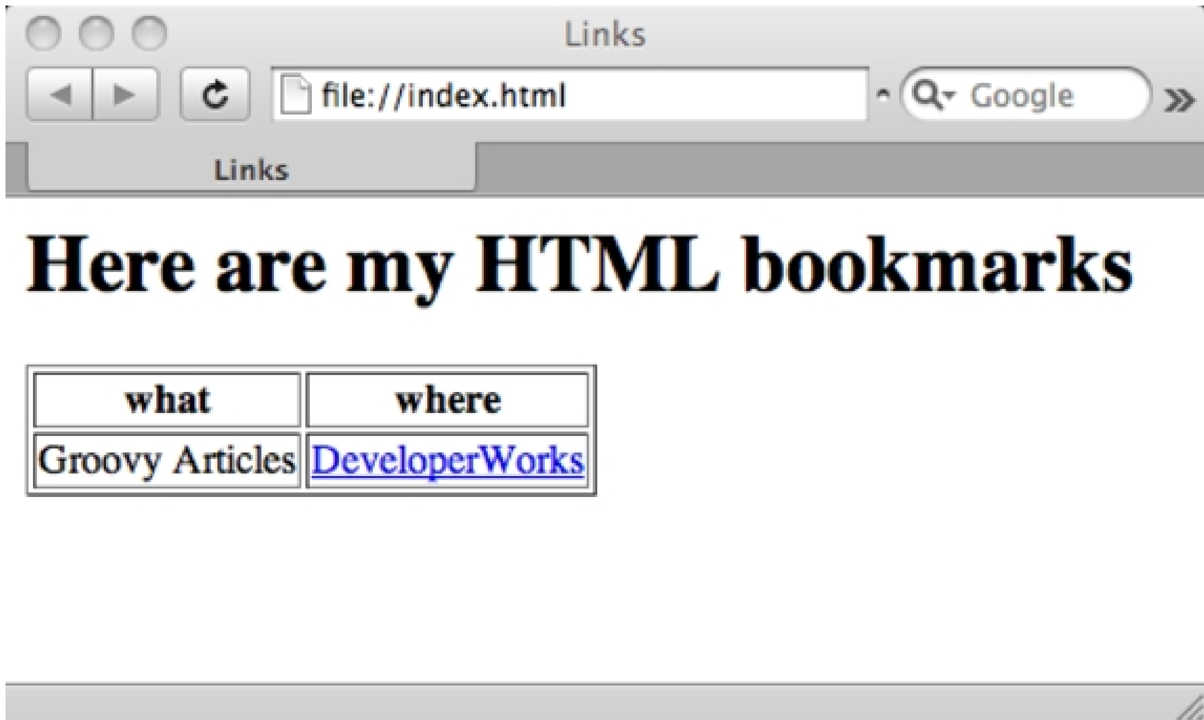
Listing 9. Building HTML with a MarkupBuilder

```
def sw = new StringWriter()
def html = new groovy.xml.MarkupBuilder(sw)
html.html{
  head{
    title("Links")
  }
  body{
    h1("Here are my HTML bookmarks")
    table(border:1){
      tr{
        th("what")
        th("where")
      }
      tr{
        td("Groovy Articles")
        td{
          a(href:"http://ibm.com/developerworks", "DeveloperWorks")
        }
      }
    }
  }
}

def f = new File("index.html")
f.write(sw.toString())

//output:
<html>
  <head>
    <title>Links</title>
  </head>
  <body>
    <h1>Here are my HTML bookmarks</h1>
    <table border='1'>
      <tr>
        <th>what</th>
        <th>where</th>
      </tr>
      <tr>
        <td>Groovy Articles</td>
        <td>
          <a href='http://ibm.com/developerworks'>DeveloperWorks</a>
        </td>
      </tr>
    </table>
  </body>
</html>
```


© Copyright IBM Corporation 2009. All rights reserved.

Figure 1 shows the browser view of the HTML built in Listing 9:

Figure 1. The rendered HTML



Creating XML with StreamingMarkupBuilder

`MarkupBuilder` is great for building simple XML documents synchronously. For more-advanced XML creation, Groovy offers a `StreamingMarkupBuilder`. With it you can add all kinds of XML extras, such as processing instructions, namespaces, and unescaped text (perfect for `CDATA` blocks) using the `mkp` helper object. Listing 10 gives you a quick tour of the interesting `StreamingMarkupBuilder` features:

Listing 10. Creating XML with StreamingMarkupBuilder

```
def comment = "<![CDATA[<!-- address is new to this release -->]]>"
def builder = new groovy.xml.StreamingMarkupBuilder()
builder.encoding = "UTF-8"
def person = {
  mkp.xmlDeclaration()
  mkp.pi("xml-stylesheet": "type='text/xsl' href='myfile.xslt'" )
  mkp.declareNamespace('':'http://myDefaultNamespace')
  mkp.declareNamespace('location':'http://someOtherNamespace')
  person(id:100){
    firstname("Jane")
    lastname("Doe")
    mkp.yieldUnescaped(comment)
    location.address("123 Main")
  }
}
def writer = new FileWriter("person.xml")
```


© Copyright IBM Corporation 2009. All rights reserved.

```
writer << builder.bind(person)

//output:
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type='text/xsl' href='myfile.xslt'?>
<person id='100'
        xmlns='http://myDefaultNamespace'
        xmlns:location='http://someOtherNamespace'>
  <firstname>Jane</firstname>
  <lastname>Doe</lastname>
  <![CDATA[<!-- address is new to this release -->]]>
  <location:address>123 Main</location:address>
</person>
```

Notice that `StreamingMarkupBuilder` doesn't produce the final XML until you call the `bind()` method, passing in the closure that contains the markup and all of the instructions. This allows you to build up various parts of the XML document asynchronously and output them all at once. (See Resources for more information.)

Understanding XmlParser

Groovy gives you two ways to produce XML — `MarkupBuilder` and `StreamingMarkupBuilder` — each with different capabilities. The same is true with parsing XML. You can use either `XmlParser` or `XmlSlurper`.

`XmlParser` offers a more programmer-centric view of the XML document. If you are comfortable thinking of the document in terms of `List`s and `Map`s (for `Element`s and `Attribute`s, respectively), then you should be comfortable with `XmlParser`. Listing 11 deconstructs the `XmlParser` a bit:

Listing 11. A closer look at XmlParser

```
def xml = """
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println langs.getClass()
// class groovy.util.Node

println langs
/*
langs[attributes={type=current, count=3, mainstream=true};
      value=[language[attributes={flavor=static, version=1.5};
                      value=[Java]],
             language[attributes={flavor=dynamic, version=1.6.0};
                      value=[Groovy]],
             language[attributes={flavor=dynamic, version=1.9};
                      value=[JavaScript]]
            ]
]
*/
```


© Copyright IBM Corporation 2009. All rights reserved.

Notice that the `XmlParser.parseText()` method returns a `groovy.util.Node` — in this case, the root `Node` of the XML document. When you call `println langs`, it calls the `Node.toString()` method, returning debug output. To get at the real data, you need to call either `Node.attribute()` or `Node.text()`.

Getting attributes with XmlParser

As you saw earlier, you can get an individual attribute by calling `Node.attribute("key")`. If you call `Node.attributes()`, it returns a `HashMap` that contains all the `Node`'s attributes. Using the `each` closure you learned about in "Reaching for Each," walking through each of the attributes is no big deal. See Listing 12 for an example of this. (See Resources for the API docs on `groovy.util.Node`.)

Listing 12. XmlParser treats attributes as a HashMap

```
def langs = new XmlParser().parseText(xml)

println langs.attribute("count")
// 3

langs.attributes().each{k,v->
  println "-" * 15
  println k
  println v
}

//output:
---------------
type
current
---------------
count
3
---------------
mainstream
true
```

As nice as working with attributes is, `XmlParser` offers even slicker support for dealing with elements.

Getting elements with XmlParser

`XmlParser` offers an intuitive way of querying elements called `GPath`. (It's similar to `XPath`, only implemented in Groovy.) For example, Listing 13 demonstrates that the `langs.language` construct I used earlier returns a `groovy.util.NodeList` containing the results of the query. A `NodeList` extends `java.util.ArrayList`, so it is basically a `List` that has been granted `GPath` superpowers.

© Copyright IBM Corporation 2009. All rights reserved.

Listing 13. Querying with GPath and XmlParser

```
def langs = new XmlParser().parseText(xml)

// shortcut query syntax
// on an anonymous NodeList
langs.language.each{
  println it.text()
}

// separating the query
// and the each closure
// into distinct parts
def list = langs.language
list.each{
  println it.text()
}

println list.getClass()
// groovy.util.NodeList
```

`GPath`, of course, is the complement to `MarkupBuilder`. It uses the same trick of calling phantom methods that don't exist, only this time it is used to query existing XML instead of generating it on the fly.

Knowing that the result of a `GPath` query is a `List`, you can make your code even more concise. Groovy offers a spread-dot operator. In a single line of code, it essentially iterates through the list and performs the method call on each item. The results are returned as a `List`. For example, if all you care about is calling the `Node.text()` method on each item in the query results, Listing 14 shows you how to do it in a single line of code:

Listing 14. Combining the spread-dot operator with GPath

```
// the long way of gathering the results
def results = []
langs.language.each{
  results << it.text()
}

// the short way using the spread-dot operator
def values = langs.language*.text()
// [Java, Groovy, JavaScript]

// quickly gathering up all of the version attributes
def versions = langs.language*.attribute("version")
// [1.5, 1.6.0, 1.9]
```

As slick and powerful as `XmlParser` is, `XmlSlurper` takes it to the next level.

Parsing XML with XmlSlurper

Way back in Listing 2, I said that Groovy makes me feel like I am working directly

© Copyright IBM Corporation 2009. All rights reserved.

with the XML. XmlParser is quite capable, but it still leaves you dealing with the XML programmatically. You are presented with Lists of Nodes and HashMaps of Attributes, and still forced to call methods like Node.attribute() and Node.text() to get at the core data. XmlSlurper removes the last vestiges of method calls, leaving you with the pleasant illusion that you are dealing with the XML directly.

Technically, XmlParser returns Nodes and NodeLists, whereas XmlSlurper returns a groovy.util.slurpersupport.GPathResult. But now that you know, I want you to forget that I ever mentioned the implementation details of XmlSlurper. You'll enjoy the magic show much more if you don't spoil the trick by peeking behind the curtain.

Listing 15 shows an XmlParser and an XmlSlurper side-by-side:

Listing 15. XmlParser and XmlSlurper

```
def xml = """
<langs type='current' count='3' mainstream='true'>
  <language flavor='static' version='1.5'>Java</language>
  <language flavor='dynamic' version='1.6.0'>Groovy</language>
  <language flavor='dynamic' version='1.9'>JavaScript</language>
</langs>
"""

def langs = new XmlParser().parseText(xml)
println langs.attribute("count")
langs.language.each{
  println it.text()
}

langs = new XmlSlurper().parseText(xml)
println langs.@count
langs.language.each{
  println it
}
```

Notice that XmlSlurper drops any notion of method calls. Instead of calling langs.attribute("count"), you call langs.@count. The at sign (@) is borrowed from XPath, but the result is the illusion that you are working with the attribute directly as opposed to calling the attribute() method. Rather than calling it.text(), you simply call it. The assumption is that you want to work directly with the element's contents.

XmlSlurper in the real world

Moving beyond langs and language, here is a real-world example of XmlSlurper in action. Yahoo! offers current weather conditions by ZIP code as an RSS feed. RSS, of course, is a specialized dialect of XML. Type http://weather.yahooapis.com/forecastrss?p=80020 in a Web browser.

© Copyright IBM Corporation 2009. All rights reserved.

Feel free to swap out the ZIP code for Broomfield, Colorado for your own. Listing 16 shows a simplified version of the resulting RSS feed:

Listing 16. Yahoo! RSS feed showing current weather conditions

```
<rss version="2.0"
     xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
     xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
  <channel>
    <title>Yahoo! Weather - Broomfield, CO</title>
    <yweather:location city="Broomfield" region="CO"   country="US"/>
    <yweather:astronomy sunrise="6:36 am"   sunset="5:50 pm"/>

    <item>
      <title>Conditions for Broomfield, CO at 7:47 am MST</title>
      <pubDate>Fri, 27 Feb 2009 7:47 am MST</pubDate>
      <yweather:condition text="Partly Cloudy"
                          code="30"  temp="25"
                          date="Fri, 27 Feb 2009 7:47 am MST" />
    </item>
  </channel>
</rss>
```

The first thing you need to do is get your hands on this RSS programmatically. Create a file named weather.groovy and add the code shown in Listing 17:

Listing 17. Getting the RSS programmatically

```
def baseUrl = "http://weather.yahooapis.com/forecastrss"

if(args){
  def zip = args[0]
  def url = baseUrl + "?p=" + zip
  def xml = url.toURL().text
  println xml
}else{
  println "USAGE: weather zipcode"
}
```

Type `groovy weather 80020` at the command line to verify that you can see the raw RSS.

The most important part of this script is `url.toURL().text`. The `url` variable is a well-formed `String`. All `String`s have a `toURL()` method added to them by Groovy that transforms them into a `java.net.URL`. All `URL`s, in turn, have a `getText()` method added to them by Groovy that performs an `HTTP GET` request and returns the result as a `String`.

Now that you have the RSS stored in the `xml` variable, mix in a little bit of `XmlSlurper` to cherry-pick out the interesting bits, as shown in Listing 18:

Listing 18. Using XmlSlurper to parse RSS

© Copyright IBM Corporation 2009. All rights reserved.

```
def baseUrl = "http://weather.yahooapis.com/forecastrss"

if(args){
  def zip = args[0]
  def url = baseUrl + "?p=" + zip
  def xml = url.toURL().text

  def rss = new XmlSlurper().parseText(xml)
  println rss.channel.title
  println "Sunrise: ${rss.channel.astronomy.@sunrise}"
  println "Sunset: ${rss.channel.astronomy.@sunset}"
  println "Currently:"
  println "\t" + rss.channel.item.condition.@date
  println "\t" + rss.channel.item.condition.@temp
  println "\t" + rss.channel.item.condition.@text
}else{
  println "USAGE: weather zipcode"
}

//output:
Yahoo! Weather - Broomfield, CO
Sunrise: 6:36 am
Sunset: 5:50 pm
Currently:
   Fri, 27 Feb 2009 7:47 am MST
   25
   Partly Cloudy
```

See how natural `XmlSlurper` makes dealing with XML? You print the `<title>` element by referring to it directly — `rss.channel.title`. You peel off the `temp` attribute with a simple `rss.channel.item.condition.@temp`. This doesn't feel like programming. It feels like working directly with the XML.

Did you notice that `XmlSlurper` even ignores the namespaces? You can turn on namespace awareness in the constructor, but I rarely do. Out of the box, `XmlSlurper` slices through XML like a hot knife through butter.

Conclusion

To be a successful software developer in this day and age, you need a set of tools that makes dealing with XML effortless. Groovy's `MarkupBuilder` and `StreamingMarkupBuilder` make it a breeze to create XML on the fly. `XmlParser` does a great job of giving you `List`s of `Element`s and `HashMap`s of `Attribute`s, and `XmlSlurper` makes the code disappear altogether, leaving you with the pleasant illusion that you are working with the XML directly.

Much of the power of XML processing wouldn't be possible without Groovy's dynamic capabilities. In the next article, I'll explore the dynamic nature of Groovy with you in greater detail. You'll learn how metaprogramming works in Groovy, from the cool methods added to standard JDK classes (like `String.toURL()` and `List.each()`) to the custom ones you'll add yourself. Until then, I hope that you find plenty of practical uses for Groovy.

© Copyright IBM Corporation 2009. All rights reserved.

© Copyright IBM Corporation 2009. All rights reserved.

Downloads

Description	Name	Size	Download method
Source code for this article's examples	j-pg05199.zip	6KB	HTTP

Information about download methods

© Copyright IBM Corporation 2009. All rights reserved.

Resources

Learn

- *Practically Groovy* (Andrew Glover and Scott Davis, developerWorks): Read all the installments in this series.
- Groovy: Learn more about Groovy at the project Web site.
- Creating XML using Groovy's StreamingMarkupBuilder: You might find these `StreamingMarkupBuilder` examples in the Groovy user guide helpful.
- `groovy.util.Node`: Check out the API documentation for `groovy.util.Node`.
- AboutGroovy.com: Keep up with the latest Groovy news and article links.
- W3C XML is Ten!: Tim Bray expounds on XML's ubiquity in this press release about the technology's tenth anniversary.
- *Mastering Grails*: Scott Davis's companion series focuses on this Groovy-based platform for Web development.
- *Groovy Recipes* (Scott Davis, Pragmatic Programmers, 2008): Learn more about Groovy and Grails in Scott Davis' latest book.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Groovy: Download the latest Groovy ZIP file or tarball.

Discuss

- Groovy mailing list: Browse, search, or subscribe to the Groovy mailing list.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Scott Davis
Scott Davis is an internationally recognized author, speaker, and software developer. He is the founder of ThirstyHead.com, a Groovy and Grails training company. His books include *Groovy Recipes: Greasing the Wheels of Java*, *GIS for Web Developers: Adding Where to Your Application*, *The Google Maps API*, and *JBoss At*

© Copyright IBM Corporation 2009. All rights reserved.

Work. He writes two ongoing article series for IBM developerWorks: *Mastering Grails* and *Practically Groovy*.

Trademarks

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

© Copyright IBM Corporation 2009. All rights reserved.