Android is a modern, open source operating system and SDK for mobile devices. With it you can create powerful mobile applications. This becomes even more attractive when your applications can access Web services, which means you need to speak the language of the Web: XML. In this article, you will see different options for working with XML on Android and how to use them to build your own Android applications.

Getting started

Learn to build Android applications that can work with XML from the Internet. Android applications are written in the Java™ programming language, so experience with Java technology is a must-have. To develop for Android, you will need the Android SDK. All of the code shown in this article will work with any version of the Android SDK, but SDK 1.5_pre was used to develop the code. You can develop Android applications with just the SDK and a text editor, but it is much easier to use the Android Developer Tools (ADT), an Eclipse plugin. For this article, version 0.9 of ADT was used with Eclipse 3.4.2, Java edition.

XML on Android

The Android platform is an open source mobile development platform. It gives you access to all aspects of the mobile device that it runs on, from low level graphics, to hardware like the camera on a phone. With so many things possible using Android, you might wonder why you need to bother with XML. It is not that working with XML is so interesting; it is working with the things that it enables. XML is commonly used as a data format on the Internet. If you want to access data
from the Internet, chances are that the data will be in the form of XML. If you want to send data to a Web service, you might also need to send XML. In short, if your Android application will leverage the Internet, then you will probably need to work with XML. Luckily, you have a lot of options available for working with XML on Android.

**XML parsers**

<table>
<thead>
<tr>
<th>Frequently used acronyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>API: Application programming interface</td>
</tr>
<tr>
<td>RSS: Really Simple Syndication</td>
</tr>
<tr>
<td>SDK: Software Developers Kit</td>
</tr>
<tr>
<td>UI: User interface</td>
</tr>
<tr>
<td>URL: Universal Resource Locator</td>
</tr>
<tr>
<td>XML: Extensible Markup Language</td>
</tr>
</tbody>
</table>

One of the greatest strengths of the Android platform is that it leverages the Java programming language. The Android SDK does not quite offer everything available to your standard Java Runtime Environment (JRE,) but it supports a very significant fraction of it. The Java platform has supported many different ways to work with XML for quite some time, and most of Java's XML-related APIs are fully supported on Android. For example, Java's Simple API for XML (SAX) and the Document Object Model (DOM) are both available on Android. Both of these APIs have been part of Java technology for many years. The newer Streaming API for XML (StAX) is not available in Android. However, Android provides a functionally equivalent library. Finally, the Java XML Binding API is also not available in Android. This API could surely be implemented in Android. However, it tends to be a heavyweight API, with many instances of many different classes often needed to represent an XML document. Thus, it is less than ideal for a constrained environment such as the handheld devices that Android is designed to run on. In the following sections, you will take a simple source of XML available on the Internet, and see how to parse it within an Android application using the various APIs mentioned above. First, look at the essential parts of the simple application that will use XML from the Internet.

**Android news reader**

The application will take an RSS feed from the popular Android developer site Androidster and parse it into a list of simple Java objects that you can use to back an Android ListView (see **Downloads** for the source code). This is classic polymorphic behavior — different implementations (different XML parsing algorithms) that provide the same behavior. **Listing 1** shows how easily you can model this in Java code using an interface.

**Listing 1. XML feed parser interface**

```java
package org.developerworks.android;
import java.util.List;

public interface FeedParser {
    List<Message> parse();
}
```

In **Listing 2**, the `Message` class is a classic Plain Old Java Object (POJO) that represents a data structure.
Listing 2. The Message POJO

```java
public class Message implements Comparable<Message> {
    static SimpleDateFormat FORMATTER =
        new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss Z");
    private String title;
    private URL link;
    private String description;
    private Date date;

    // getters and setters omitted for brevity
    public void setLink(String link) {
        try {
            this.link = new URL(link);
        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    public String getDate() {
        return FORMATTER.format(this.date);
    }

    public void setDate(String date) {
        // pad the date if necessary
        while (!date.endsWith("00")) {
            date += "0";
        }
        try {
            this.date = FORMATTER.parse(date.trim());
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public String toString() {
        // omitted for brevity
    }

    @Override
    public int hashCode() {
        // omitted for brevity
    }

    @Override
    public boolean equals(Object obj) {
        // omitted for brevity
    }

    @Override
    public int compareTo(Message another) {
        if (another == null) return 1;
        // sort descending, most recent first
        return another.date.compareTo(date);
    }
}
```

Message, in Listing 2, is mostly straightforward. It does hide some of its internal state by allowing dates and links to be accessed as simple strings, while representing them as more strongly typed objects (a `java.util.Date` and a `java.net.URL`). It is a classic Value Object, thus it implements `equals()` and `hashCode()` based upon its internal state. It also implements the `Comparable` interface so you can use it for sorting (by date). In practice, the data always comes sorted from the feed, so this is not necessary.
Each of the parser implementations will need to take a URL to the Androidster feed and use this to open an HTTP connection to the Androidster site. This common behavior is naturally modeled in Java code using an abstract base class as in Listing 3.

### Listing 3. Base feed parser class

```java
public abstract class BaseFeedParser implements FeedParser {

    // names of the XML tags
    static final String PUB_DATE = "pubDate";
    static final String DESCRIPTION = "description";
    static final String LINK = "link";
    static final String TITLE = "title";
    static final String ITEM = "item";

    final URL feedUrl;

    protected BaseFeedParser(String feedUrl) {
        try {
            this.feedUrl = new URL(feedUrl);
        } catch (MalformedURLException e) {
            throw new RuntimeException(e);
        }
    }

    protected InputStream getInputStream() {
        try {
            return feedUrl.openConnection().getInputStream();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

The base class stores the `feedUrl` and uses it to open a `java.io.InputStream`. If anything goes wrong, it simply throws a `RuntimeException`, so that the application simply fails quickly. The base class also defines some simple constants for the names of the tags. Listing 4 shows some sample content from the feed, so that you can see the significance of these tags.

### Listing 4. Sample XML feed

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- generator="FeedCreator 1.7.2" -->
<rss version="2.0">
    <channel>
        <title>android_news</title>
        <description>android_news</description>
        <link>http://www.androidster.com/android_news.php</link>
        <lastBuildDate>Sun, 19 Apr 2009 19:43:45 +0100</lastBuildDate>
        <generator>FeedCreator 1.7.2</generator>
        <item>
            <title>Samsung S8000 to Run Android, Play DivX, Take Over the World</title>
```
As you can see from the sample in Listing 4, an ITEM corresponds to a Message instance. The child nodes of item (TITLE, LINK, and so on) correspond to the properties of the Message instance. Now that you know what the feed looks like, and have all of the common parts in place, take a look at how to parse this feed using the various technologies available on Android. You will start with SAX.

Using SAX

In a Java environment, you can often use the SAX API when you want a fast parser and want to minimize the memory footprint of your application. That makes it very well suited for a mobile device running Android. You can use the SAX API as-is from the Java environment, with no special modifications needed to run on Android. Listing 5 shows a SAX implementation of the FeedParser interface.
Listing 5. SAX implementation

```java
public class SaxFeedParser extends BaseFeedParser {
    protected SaxFeedParser(String feedUrl) {
        super(feedUrl);
    }

    public List<Message> parse() {
        SAXParserFactory factory = SAXParserFactory.newInstance();
        try {
            SAXParser parser = factory.newSAXParser();
            RssHandler handler = new RssHandler();
            parser.parse(this.getInputStream(), handler);
            return handler.getMessages();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

If you have used SAX before, this looks pretty familiar. As with any SAX implementation, most of the details are in the SAX handler. The handler receives events from the SAX parser as it rips through the XML document. In this case, you have created a new class called `RssHandler` and registered it as the handler for the parser, as in Listing 6.

Listing 6. The SAX handler

```java
import static org.developerworks.android.BaseFeedParser.*;

public class RssHandler extends DefaultHandler {
    private List<Message> messages;
    private Message currentMessage;
    private StringBuilder builder;

    public List<Message> getMessages() {
        return this.messages;
    }

    @Override
    public void characters(char[] ch, int start, int length) throws SAXException {
        super.characters(ch, start, length);
        builder.append(ch, start, length);
    }

    @Override
    public void endElement(String uri, String localName, String name) throws SAXException {
        super.endElement(uri, localName, name);
        if (this.currentMessage != null) {
            if (localName.equalsIgnoreCase(TITLE)) {
                currentMessage.setTitle(builder.toString());
            } else if (localName.equalsIgnoreCase(LINK)) {
                currentMessage.setLink(builder.toString());
            } else if (localName.equalsIgnoreCase(DESCRIPTION)) {
                currentMessage.setDescription(builder.toString());
            } else if (localName.equalsIgnoreCase(PUB_DATE)) {
                currentMessage.setDate(builder.toString());
            } else if (localName.equalsIgnoreCase(ITEM)) {
                messages.add(currentMessage);
            }
            builder.setLength(0);
        }
    }
}
```
```java
@Override
public void startDocument() throws SAXException {
    super.startDocument();
    messages = new ArrayList<Message>();
    builder = new StringBuilder();
}

@Override
public void startElement(String uri, String localName, String name,
        Attributes attributes) throws SAXException {
    super.startElement(uri, localName, name, attributes);
    if (localName.equalsIgnoreCase(ITEM)) {
        this.currentMessage = new Message();
    }
}
}
```

The RssHandler class extends the org.xml.sax.helpers.DefaultHandler class. This class provides default, no-op implementations for all of the methods that correspond to the events raised by the SAX parser. This allows subclasses to only override methods as needed. The RssHandler has one additional API, getMessages. This returns the list of Message objects that the handler collects as it receives events from the SAX parser. It has two other internal variables, a currentMessage for a Message instance that is being parsed, and a StringBuilder variable called builder that stores character data from text nodes. These are both initialized when the startDocument method is invoked when the parser sends the corresponding event to the handler.

Take a look at the startElement method in Listing 6. This is called every time an opening tag is encountered in the XML document. You only care when that tag is an ITEM tag. In that case you create a new Message. Now look at the characters method. This is called when character data from text nodes is encountered. The data is simply added to the builder variable. Finally look at the endElement method. This is called when an end tag is encountered. For the tags corresponding to properties of a Message, like TITLE and LINK, the appropriate property is set on the currentMessage using the data from the builder variable. If the end tag is an ITEM, then the currentMessage is added to the list of Messages. This is all very typical SAX parsing; nothing here is unique to Android. So if you know how to write a Java SAX parser, then you know how to write an Android SAX parser. However, the Android SDK does add some convenience features on top of SAX.

**Easier SAX parsing**

The Android SDK contains a utility class called android.util.Xml. Listing 7 shows how to setup a SAX parser with that same utility class.
Listing 7. Android SAX parser

```java
public class AndroidSaxFeedParser extends BaseFeedParser {
    public AndroidSaxFeedParser(String feedUrl) {
        super(feedUrl);
    }

    public List<Message> parse() {
        RssHandler handler = new RssHandler();
        try {
            Xml.parse(this.getInputStream(), Xml.Encoding.UTF_8, handler);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return handler.getMessages();
    }
```

Notice that this class still uses a standard SAX handler, so you simply reused the RssHandler shown above in Listing 7. Being able to reuse SAX handler is great, but it is a somewhat complicated piece of code. You can imagine, if you had to parse a much more complex XML document, that the handler might become a breeding ground for bugs. For example, look back at the endElement method in Listing 6. Notice how it checks if the currentMessage is null before it tries to set properties? Now take a look back at the sample XML in Listing 4. Notice that there are TITLE and LINK tags outside of the ITEM tags. That is why the null check was put in. Otherwise the first TITLE tag can cause a NullPointerException. Android includes its own variant of the SAX API (see Listing 8) that removes the need for you to write your own SAX handler.

Listing 8. Simplified Android SAX parser

```java
public class AndroidSaxFeedParser extends BaseFeedParser {
    public AndroidSaxFeedParser(String feedUrl) {
        super(feedUrl);
    }

    public List<Message> parse() {
        final Message currentMessage = new Message();
        RootElement root = new RootElement("rss");
        final List<Message> messages = new ArrayList<Message>();
        Element channel = root.getChild("channel");
        Element item = channel.getChild("item");
        item.setEndElementListener(new EndElementListener()
        {
            public void end() {
                messages.add(currentMessage.copy());
            }
        });
        item.getChild("title").setEndTextElementListener(new EndTextElementListener()
        {
            public void end(String body) {
                currentMessage.setTitle(body);
            }
        });
        item.getChild("link").setEndTextElementListener(new EndTextElementListener()
        {
            public void end(String body) {
                currentMessage.setLink(body);
            }
        });
        item.getChild("description").setEndTextElementListener(new EndTextElementListener()
        {
            public void end(String body) {
            }
        });
```

Working with XML on Android
As promised, the new SAX parsing code does not use a SAX handler. Instead it uses classes from the android.sax package in the SDK. These allow you to model the structure of your XML document and add an event listener as needed. In the above code, you declare that your document will have a root element called `rss` and that this element will have a child element called `channel`. Then you say that `channel` will have a child element called `ITEM` and you start to attach listeners. For each listener, you used an anonymous inner class that implemented the interface that you were interested in (either `EndElementListner` or `EndTextElementListener`). Notice there was no need to keep track of character data. Not only is this simpler, but it is actually more efficient. Finally, when you invoke the `Xml.parse` utility method, you now pass in a handler that is generated from the root element.

All of the above code in Listing 8 is optional. If you are comfortable with standard SAX parsing code in the Java environment, then you can stick to that. If you want to try out the convenience wrappers provided by the Android SDK, you can use that as well. What if you do not want to use SAX at all? Some alternatives are available. The first one you will look at is DOM.

**Working with DOM**

DOM parsing on Android is fully supported. It works exactly as it works in Java code that you would run on a desktop machine or a server. Listing 9 shows a DOM-based implementation of the parser interface.

**Listing 9. DOM-based implementation of feed parser**

```java
public class DomFeedParser extends BaseFeedParser {

    protected DomFeedParser(String feedUrl) {
        super(feedUrl);
    }

    public List<Message> parse() {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        List<Message> messages = new ArrayList<Message>();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document dom = builder.parse(this.getInputStream());
            Element root = dom.getDocumentElement();
            NodeList items = root.getElementsByTagName(ITEM);
            for (int i=0;i<items.getLength();i++){
                currentMessage.setDescription(body);
            }
        }
    }
}
```
Like the first SAX example, nothing is Android-specific about this code. The DOM parser reads all of the XML document into memory and then allows you to use the DOM APIs to transverse the XML tree, retrieving the data that you want. This is very straightforward code, and, in some ways, simpler than the SAX-based implementations. However, DOM generally consumes more memory as everything is read into memory first. This can be a problem on mobile devices that run Android, but it can be satisfactory in certain use cases where the size of the XML document will never be very large. One might imply that the developers of Android guessed that SAX parsing would be much more common on Android applications, hence the extra utilities provided for it. One other type of XML parser is available to you on Android, and that is the pull parser.

**The XML pull parser**

As mentioned earlier, Android does not provide support for Java's StAX API. However, Android does come with a pull parser that works similarly to StAX. It allows your application code to pull or seek events from the parser, as opposed to the SAX parser that automatically pushes events to the handler. **Listing 10** shows a pull parser implementation of the feed parser interface.

**Listing 10. Pull parser based implementation**

```java
public class XmlPullFeedParser extends BaseFeedParser {
    public XmlPullFeedParser(String feedUrl) {
        super(feedUrl);
    }

    public List<Message> parse() {
        List<Message> messages = null;
        XmlPullParser parser = Xml.newPullParser();
        try {
            // auto-detect the encoding from the stream
            parser.setInput(this.getInputStream(), null);
            int eventType = parser.getEventType();
            ... (code snippet)
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return messages;
    }
}
```
Message currentMessage = null;
boolean done = false;
while (eventType != XmlPullParser.END_DOCUMENT && !done){
    String name = null;
    switch (eventType){
        case XmlPullParser.START_DOCUMENT:
            messages = new ArrayList<Message>();
            break;
        case XmlPullParser.START_TAG:
            name = parser.getName();
            if (name.equalsIgnoreCase(ITEM)){
                currentMessage = new Message();
            } else if (currentMessage != null){
                if (name.equalsIgnoreCase(LINK)){
                    currentMessage.setLink(parser.nextText());
                } else if (name.equalsIgnoreCase(DESCRIPTION)){
                    currentMessage.setDescription(parser.nextText());
                } else if (name.equalsIgnoreCase(PUB_DATE)){
                    currentMessage.setDate(parser.nextText());
                } else if (name.equalsIgnoreCase(TITLE)){
                    currentMessage.setTitle(parser.nextText());
                }
            }
            break;
        case XmlPullParser.END_TAG:
            name = parser.getName();
            if (name.equalsIgnoreCase(ITEM) &&
                currentMessage != null){
                messages.add(currentMessage);
            } else if (name.equalsIgnoreCase(CHANNEL)){
                done = true;
            }
            break;
        }
        eventType = parser.next();
    }
} catch (Exception e) {
    throw new RuntimeException(e);
}
return messages;
}

A pull parser works similarly to a SAX parser. It has similar events (start element, end element) but you have to pull from them (parser.next()). The events are sent as numeric codes, so you can use a simple case-switch. Notice that, instead of listening for the end of elements as in SAX parsing, with pull parsing, it is simple to do most of your processing at the beginning. In the code in Listing 10, when an element starts, you can call parser.nextText() to pull all of the character data from the XML document. This offers a nice simplification to SAX parsing. Also notice that you set a flag (the boolean variable done) to identify when you reach the end of the content that you are interested in. This lets you halt the reading of the XML document early, as you know that the code will not care about the rest of the document. This can be very useful, especially if you only need a small portion of the XML document being accessed. You can greatly reduce the parsing time by stopping the parsing as soon as possible. Again, this kind of optimization is especially important on mobile devices where the connection speed can be slow. The pull parser can have some nice performance advantages as well as ease of use. It can also be used to write XML.
Creating XML

So far, I have concentrated on parsing XML from the Internet. However, sometimes your application might need to send XML to a remote server. You can obviously just use a StringBuilder or something similar to create an XML string. Another alternative comes from the pull parser in Listing 11.

Listing 11. Writing XML with pull parser

```java
private String writeXml(List<Message> messages){
    XmlSerializer serializer = Xml.newSerializer();
    StringWriter writer = new StringWriter();
    try {
        serializer.setOutput(writer);
        serializer.startDocument("UTF-8", true);
        serializer.startTag("", "messages");
        serializer.attribute("", "number", String.valueOf(messages.size()));
        for (Message msg: messages){
            serializer.startTag("", "message");
            serializer.attribute("", "date", msg.getDate());
            serializer.text(msg.getTitle());
            serializer.endTag("", "title");
            serializer.startTag("", "url");
            serializer.text(msg.getLink().toExternalForm());
            serializer.endTag("", "url");
            serializer.startTag("", "body");
            serializer.text(msg.getDescription());
            serializer.endTag("", "body");
            serializer.endTag("", "message");
        }
        serializer.endTag("", "messages");
        serializer.endDocument();
        return writer.toString();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

The XmlSerializer class is part of the same package as the XmlPullParser used in the previous section. Instead of pulling in events, it pushes them out to a stream or a writer. In this case it simply pushes them to a java.io.StringWriter instance. It provides a straightforward API with methods to start and end a document, process elements, and add text or attributes. This can be a nice alternative to using a StringBuilder, as it is easier to ensure your XML is well formed.

Summary

What kind of application do you want to build for Android devices? Whatever it is, if it needs to work with data from the Internet, then it probably needs to work with XML. In this article, you saw that Android comes loaded with lots of tools for dealing with XML. You can pick just one of these as your tool-of-choice, or you can pick and choose based on the use case. Most of the time the safe pick is to go with SAX, and Android gives you both a traditional way to do SAX and a slick convenience wrapper on top of SAX. If your document is small, then perhaps DOM is the simpler way to go. If your document is large, but you only need part of the document, then the XML pull parser might be a more efficient way to go. Finally, for writing XML, the pull parser package
provides a convenient way to do that as well. So, whatever your XML needs are, the Android SDK has something for you.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>AndroidXml.zip</td>
<td>70KB</td>
</tr>
</tbody>
</table>
Related topics

- Android SDK documentation
- The Open Handset Alliance
- The Android SDK download
- Android Open Source Project

© Copyright IBM Corporation 2009, 2013
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)