Many of your Android applications will need to interact with Internet data, which comes in a variety of formats. In this article, build an Android application that works with two popular data formats—XML and JavaScript Object Notation (JSON)—as well as the more exotic protocol buffers format from Google. You'll learn about the performance and coding trade-offs associated with each format.

Frequently used acronyms

- Ajax: Asynchronous JavaScript + XML
- API: Application Programming Interface
- CSV: Comma-separated value
- CSS: Cascading stylesheet
- DOM: Document Object Model
- HTML: HyperText Markup Language
- HTTP: Hypertext Transfer Protocol
- IDL: Interface Description Language
- JSON: JavaScript Object Notation
- SAX: Simple API for XML
- SDK: Software Developer Kit
- UI: User Interface
- URL: Uniform Resource Locator
- XML: Extensible Markup Language
- 3G: Third-generation of mobile phone technology standards

Android applications often must access data that resides on the Internet, and Internet data can be structured in several different formats. In this article, you'll see how to work with three data formats within Android applications:

- XML
- JSON
- Google's protocol buffers

First you'll develop a Web service that converts CSV data into XML, JSON, and protocol-buffers formats. Then you'll build a sample Android application that can pull the data from the Web service in any of these formats and parse it for display to the user.
To perform the exercise in this article, you need the latest Android SDK (see Related topics) and the Android 2.2 platform. The SDK requires that you also have a Java™ Development Kit (JDK) installed; I used JDK 1.6.0_17 for this article. You don't need a physical Android device; all of the code will run on the SDK’s Android emulator. This article doesn't try to teach you Android development per se, so familiarity with Android programming is recommended. However, you can probably follow along just with knowledge of the Java programming language.

You also need a Java web application server to run the Web service that the Android application uses. Alternatively, you can deploy the server-side code to the Google App Engine. See Download to get the complete source code.

**The Day Trader application**

You'll develop a simple Android application called Day Trader. Day Trader lets the user enter one or more stock symbols and retrieve the latest pricing information for the stocks they represent. The user can specify the format to use for the data: XML, JSON, or protocol buffers. A real-world Android application wouldn't typically offer this choice, but by implementing it you'll see how to make your applications work with each format. **Figure 1** shows the Day Trader user interface:
Figure 1. The Day Trader application in action

The text box and the Add Stock button next to it let the user enter the symbol for each stock of interest. When the user presses the Download Stock Data button, the data for all those stocks is requested from the server, parsed in the application, and displayed on the screen. By default, the data is retrieved as XML. With the menu, you can switch the data format between XML, JSON, or protocol buffers.

Listing 1 shows the layout XML used to create the UI in Figure 1:

Listing 1. Day Trader layout XML

```xml
<?xml version="1.0" encoding="utf-8"?>

<grid>
  <row>
    <edittext>
      <desc>Symbol</desc>
    </edittext>
    <button>Add Stock</button>
  </row>
  <row>
    <button>Download Stock Data</button>
  </row>
</grid>
```
Most of the code in Listing 1 is straightforward. You see several widgets to create the inputs and buttons shown in Figure 1. You also see a ListView, the veritable Swiss Army knife of Android widgets. This ListView will be populated with the stock data downloaded from the server. Listing 2 shows the Activity that controls this view:

Listing 2. Day Trader main activity

```java
public class Main extends ListActivity {
    private int mode = XML; // default

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        final EditText input = (EditText) findViewById(R.id.symbol);
        final TextView symbolsList = (TextView) findViewById(R.id.symList);
        final Button addButton = (Button) findViewById(R.id.addBtn);
        final Button dlButton = (Button) findViewById(R.id.dlBtn);

        addButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                String newSymbol = input.getText().toString();
                if (symbolsList.getText() == null ||
                        symbolsList.getText().length() == 0) {
                    symbolsList.setText(newSymbol);
                } else {
                    StringBuilder sb = new StringBuilder(symbolsList.getText());
                    sb.append(",");
                    sb.append(newSymbol);
                    symbolsList.setText(sb.toString());
                }
                input.setText(""");
            }
        });
```

This Activity sets the layout to the XML file in Listing 1, and it wires up a couple of event handlers. First, for the Add Stock button, the code reads the symbols from the text box and adds them to the symList TextView, separating each symbol with a comma. Next, for the Download button, the handler reads the data from that symList TextView, and then—based on the mode variable—uses one of three different classes for downloading the data from the server. The menu sets the value of the mode variable; it is fairly trivial code so I've omitted it from Listing 2. Before you look at the various data downloading/parsing classes, I'll show you how the server provides this data.

**Serving stock data**

The server for your application needs to be able to do two things. First, it must take a list of stock symbols and retrieve data for each of them. Next, it needs to accept a format parameter and encode the data based on that format. For XML and JSON formats, the server will return the stock data serialized as text. For protocol buffers, it must send binary data. Listing 3 shows the servlet that handles these steps:

**Listing 3. The Stock Broker servlet**

```java
public class StockBrokerServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
        String[] symbols = request.getParameterValues("stock");
        List<Stock> stocks = getStocks(symbols);
        String format = request.getParameter("format");
        String data = ";
        if (format == null || format.equalsIgnoreCase("xml")) {
            data = Stock.toXml(stocks);
            response.setContentType("text/xml");
        } else if (format.equalsIgnoreCase("json")) {
            data = Stock.toJson(stocks);
            response.setContentType("application/json");
        } else if (format.equalsIgnoreCase("protobuf")) {
            Portfolio p = Stock.toProtoBuf(stocks);
            response.setContentType("application/octet-stream");
            response.setContentLength(p.getSerializedSize());
            p.writeTo(response.getOutputStream());
            response.flushBuffer();
        }
    }
}
```

Before looking at the various data downloading/parsing classes, I’ll show you how the server provides this data.
This is a simple Java servlet that only supports HTTP GET requests. It reads in the values of the stock and format-request parameters. Then it calls the `getStocks()` method. This method makes a call to Yahoo! Finance to get the stock data. Yahoo! supports only CSV format for the data, so the `getStocks()` method parses it into a list of `Stock` objects. **Listing 4** shows this a simple data structure:

**Listing 4. Stock data structure**

```java
public class Stock {
    private final String symbol;
    private final String name;
    private final double price;
    //getters and setters omitted

    public String toXml(){
        return "<stock><symbol>" + symbol + "</symbol><name><![CDATA[" + name + "]]]></name><price>" + price + "</price></stock>";
    }

    public String toJson(){
        return "{" + 'symbol' : " + symbol + ", 'name':" + name + ", 'price': "+ price + "}";
    }

    public static String toXml(List<Stock> stocks){
        StringBuilder xml = new StringBuilder("<stocks>");
        // code...
    }
}
```
Each Stock has three properties—symbol, name, and price—and convenience methods to convert itself into either an XML string or a JSON string. And it has utility methods for converting lists of Stock objects into XML or JSON. Back in Listing 3, depending on the format-request parameter, the list of Stock objects is converted to XML or JSON strings and sent back to the client.

The XML and JSON use cases are fairly similar and straightforward. For the protocol buffers case, you must generate code-reading and -writing objects in the protocol buffers format. To do this, you need to define the data structure using the protocol buffers specification format. Listing 5 shows an example:

### Listing 5. Protocol buffers messages for stocks

```java
package stocks;

option java_package = "org.developerworks.stocks";

message Quote{
  required string symbol = 1;
  required string name = 2;
  required double price = 3;
}

message Portfolio{
  repeated Quote quote = 1;
}
```

The protocol buffers message format, which is similar to the Interface Description Language (IDL), is meant to be language-independent so that you can use it with various languages. In this case, you run the protocol buffers compiler (protoc) to compile the code in Listing 5 into Java classes that you'll use for the client and the server. For details on compiling protocol buffers messages into Java classes, consult the Protocol Buffers Developer Guide (see Related topics).

In Listing 3, a method called toProtoBuf() converted the list of Stock objects to a Portfolio message. Listing 6 shows the implementation of that method:
Listing 6. Creating a portfolio message

```java
public static Stocks.Portfolio toProtoBuf(List<Stock> stocks){
    List<Stocks.Quote> quotes = new ArrayList<Stocks.Quote>(stocks.size());
    for (Stock s : stocks){
        Quote q =
            Quote.newBuilder()
            .setName(s.name)
            .setSymbol(s.symbol)
            .setPrice(s.price)
            .build();
        quotes.add(q);
    }
    return Portfolio.newBuilder().addAllQuote(quotes).build();
}
```

The code in Listing 6 uses code—the Quote and Portfolio classes—that was generated from the messages in Listing 5. You simply build a quote from each Stock object, and then add it to a Portfolio object that is returned to the servlet in Listing 3. In Listing 3, the servlet opens up a stream directly to the client and uses the generated code to write the binary protocol buffers data to the stream.

Now you have seen how the server creates the data that will be sent to the Android application. Next, you'll look at how the application parses this data.

Working with data formats

The main Activity in Listing 2 needs to work with data in the various formats that your server can send it in. It also needs to request the data in the appropriate format and, once the data is parsed, use it to fill its ListView with data. So, much of the functionality is common regardless of data format.

First, create an abstract base class that encapsulates this common functionality, as in Listing 7:

Listing 7. Base class for data parsers

```java
abstract class BaseStockParser extends AsyncTask<String, Integer, Stock[]>{
    String urlStr = "http://protostocks.appspot.com/stockbroker?format=";

    protected BaseStockParser(String format){
        urlStr += format;
    }

    private String makeUrlString(String... symbols) {
        StringBuilder sb = new StringBuilder(urlStr);
        for (int i=0;i<symbols.length;i++){
            sb.append("&stock=");
            sb.append(symbols[i]);
        }
        return sb.toString();
    }

    protected InputStream getData(String[] symbols) throws Exception{
        HttpClient client = new DefaultHttpClient();
        HttpGet request = new HttpGet(new URI(makeUrlString(symbols)));
        HttpResponse response = client.execute(request);
        return response.getEntity().getContent();
    }
}
```
The base class in **Listing 7** extends `android.os.AsyncTask`. This is a commonly used class for asynchronous operations. It abstracts out the creation of a thread and a handler for making a request off of the main UI thread. It is parameterized based on its input and output data types. For all of your parsers, the inputs are always the same: strings for the stock symbols. And the output is always the same: an array of `Stock` objects. The base class takes `format`, a string that specifies the data format to use. It then has a method for making the appropriate HTTP request and returning a streaming response. Finally, it overrides `AsyncTask`'s `onPostExecute()` method and uses the data returned from the parser to create an `Adapter` for the Activity's `ListView`.

Now that you've seen the functionality that is common to all three parsers, I'll show you the more specific parsing code, starting with the XML parser.

### Parsing XML with SAX

The Android SDK provides several ways to work with XML, including standard DOM and SAX. For some more memory-intensive situations, you can use the SDK's pull-parser. Most of the time, SAX is the fastest way to go, and Android includes some convenience APIs to make it easier to use SAX. **Listing 8** shows the XML parser for the Day Trader application:

```java
private class StockXmlParser extends BaseStockParser {
    public StockXmlParser() {
        super("xml");
    }

    @Override
    protected Stock[] doInBackground(String... symbols) {
        ArrayList<Stock> stocks = new ArrayList<Stock>(symbols.length);
        try {
            ContentHandler handler = newHandler(stocks);
            Xml.parse(getData(symbols), Xml.Encoding.UTF_8, handler);
        } catch (Exception e) {
            Log.e("DayTrader", "Exception getting XML data", e);
        }
        Stock[] array = new Stock[symbols.length];
        return stocks.toArray(array);
    }

    private ContentHandler newHandler(final ArrayList<Stock> stocks) {
        RootElement root = new RootElement("stocks");
        Element stock = root.getChild("stock");
        final Stock currentStock = new Stock();
        stock.setEndElementListener(new EndElementListener() {
            public void end() {
                stocks.add((Stock) currentStock.clone());
            }
        });
    }
}
```

**Listing 8. XML parser implementation**
Most of the code in Listing 8 is in the newHandler() method, which creates a ContentHandler. If you're familiar with SAX parsing, you know that the ContentHandler creates the parsed data by reacting to the various events fired by the SAX parser. The newHandler() method uses the Android convenience APIs to specify the ContentHandler using event handlers. The code simply listens for the events fired when the parser encounters various tags, and then picks out the data to put into a list of Stock objects. Once the ContentHandler is created, the Xml.parse() method is invoked to parse the InputStream provided by the base class and return an array of Stock objects. This is a fast way to parse XML, but—even with the convenience APIs that Android provides—it is still fairly verbose.

Using JSON

XML is a first-class citizen on Android, which is a good thing given how many Web services rely on XML. Many services also support JSON, another popular format. It is usually a little more compact than XML, but it is still human-readable, making it easy to work with and easy to debug applications that use it. Android includes a JSON parser. (It's the same parser that you can get from the JSON.org website, except with a few not-needed-on-mobile classes pruned away.) Listing 9 shows it in action:

Listing 9. JSON parser implementation

```java
class StockJsonParser extends BaseStockParser{
    public StockJsonParser(){
        super("json");
    }
    @Override
    protected Stock[] doInBackground(String... symbols) {
        Stock[] stocks = new Stock[symbols.length];
        try{
            StringBuilder json = new StringBuilder();
            BufferedReader reader =
                new BufferedReader(getData(symbols));
            String line = reader.readLine();
```
while (line != null){
    json.append(line);
    line = reader.readLine();
}
JSONObject jsonobj = new JSONObject(json.toString());
JSONArray stockArray = jsonobj.getJSONArray("stocks");
for (int i=0;i<stocks.length;i++){
    JSONObject object =
        stockArray.getJSONObject(i).getJSONObject("stock");
    stocks[i] = new Stock(object.getString("symbol"),
        object.getString("name"),
        object.getDouble("price"));
}  }
}
catch (Exception e){
    Log.e("DayTrader", "Exception getting JSON data", e);
}
return stocks;
}

You can see how easy it is to use the JSON parser in Android. You convert the stream from
the server into a string that you pass to the JSON parser. You traverse object graph and create
the array of Stock objects. If you have worked with XML DOM parsing, this should look familiar,
because the programming model is almost the same.

Like DOM, the JSON parser can be memory-intensive to use. In Listing 9, all of the data from the
server is represented as a string, then as a JSONObject, and finally as an array of Stock objects.
In other words, the exact same data is represented in three different ways. You can see how this
might be a problem for large amounts of data. Of course, once you get to the end of the method,
two of these three representations of the data fall out of scope and can be reclaimed by the
garbage collector. However, just triggering more-frequent garbage collections can have a negative
effect on the user experience by causing erratic slowdowns. If memory efficiency and performance
are significant, a parser that uses protocol buffers might be a better choice.

Going binary with protocol buffers

Protocol buffers is a language-agnostic data-serialization format developed by Google, designed
to be faster than XML for sending data over a network. It is the de facto standard at Google for any
server-to-server calls. Google made the format and its binding tools for the C++, Java, and Python
programming languages available as open source.

You saw in Listing 3 and Listing 6 that protocol buffers is a binary format. As you might expect, this
makes the data very compact. You can often get similar message size with XML and JSON if you
can enable gzip compression on both the client and server, but protocol buffers still offer some size
advantages. And it is also a format that can be parsed very quickly. Finally, it offers a fairly simple
API. Listing 10 shows an example parser implementation:
Listing 10. Protocol buffers parser implementation

```java
class StockProtoBufParser extends BaseStockParser{
    public StockProtoBufParser(){
        super("protobuf");
    }

    @Override
    protected Stock[] doInBackground(String... symbols) {
        Stock[] stocks = new Stock[symbols.length];
        try{
            Stocks.Portfolio portfolio = Stocks.Portfolio.parseFrom(getData(symbols));
            for (int i=0;i<symbols.length;i++){
                stocks[i] = Stock.fromQuote(portfolio.getQuote(i));
            }
        } catch (Exception e){
            Log.e("DayTrader", "Exception getting ProtocolBuffer data", e);
        }
        return stocks;
    }
}
```

Just as in Listing 3, you use a helper class, generated in this case by the protocol buffers compiler. This is the same helper class used by the server. You can compile it once and share it on both the server and the client. Thus, you can more easily read directly from the stream from the server and turn it into an array of Stock objects. This is simple programming that also happens to have excellent performance characteristics. Now look at how this performance stacks up against XML and JSON.

Performance comparison

Comparing performance usually involves some kind of microbenchmark, and such benchmarks are notoriously easy to bias or get incorrect in some unintentional way. Even when a microbenchmark is designed in a fair way, many random factors can cast doubt on any results. These caveats notwithstanding, I used just such a microbenchmark to compare the performance of XML (about 1300 ms), JSON (about 1150 ms), and protocol buffers (about 750 ms). The benchmark sent a request for 200 stocks to the server and measured the amount of time from when the request went out to when the data was ready to be used to create the Adapter for the ListView. This was done 50 times for each data format, on two different devices: a Motorola Droid and an HTC Evo, both over 3G networks. Figure 2 shows the results:
The graph in Figure 2 shows that protocol buffers (about 750 ms) were almost twice as fast as XML (about 1300 ms) for this microbenchmark. Many factors affect performance of data going over the network and being processed by a handheld device. An obvious factor is the amount of data going over the network. Indeed, protocol buffers, being a binary format, is much smaller over the network than text formats like XML and JSON. However, text formats can be efficiently compressed using gzip, which is a standard technology supported by both web servers and Android devices. Figure 3 shows the size of the data going across the wire, with gzip turned on and off:

Figure 3 should increase your appreciation for the effect of compression on text content like XML and JSON (not to mention web formats, HTML, JavaScript, and CSS). The data of the protocol
buffers (about 6KB) is much smaller than raw XML (about 17.5KB) or JSON (about 13.5KB). But once compression kicks in, JSON and XML (both are about 3KB) are actually smaller than protocol buffers. In this example they are close to half the size of protocol-buffers-encoded messages.

Going back to Figure 2, the difference in speed can certainly not be explained by message size over the network. The protocol-buffers messages are larger than the XML- or JSON-encoded messages, yet by using protocol buffers you can still shave off a half-second of time that the user waits to see data. Does that mean that you should use protocol buffers for your Android application? Such decisions are rarely that cut and dried, of course. If the amount of data being sent is small, then the difference among the three formats is slight. For large amounts of data, perhaps protocol buffers would make a difference. However, a contrived benchmark like this one is no substitute for testing your own application.

**Conclusion**

In this article, you explored the ins and outs of working with two popular data formats used on the Internet, XML and JSON. You also saw a third possibility, protocol buffers. Like everything else in software engineering, choosing a technology is all about trade-offs. Often the consequences of these decisions are amplified when you develop for a constrained environment such as Android. I hope the extra knowledge you now have about those consequences will help you create great Android applications.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Article source code</td>
<td>daytrader.stockbroker.zip</td>
<td>4.81MB</td>
</tr>
</tbody>
</table>
Related topics

- **Android Developers site**: Get the latest news on Android and access the Android API reference.
- **Working with XML on Android** (Michael Galpin, developerWorks, June 2009): Learn about the options for working with XML on Android and how to use them to build your own Android applications.
- **Introduction to Android development** (Frank Ableson, developerWorks, May 2009): Get started with Android in this great introduction to the Android platform and code a basic Android application.
- **Protocol Buffers Developer Guide**: Learn more about developing with protocol buffers.
- **Introducing JSON**: Learn JSON syntax.
- **To XML and Back: Using JSON in Android** (Frank Ableson, Linux Magazine, March 2010): Get more information on using JSON with Android.
- **Open Handset Alliance**: Visit the website of Android's sponsor.
- **Understanding SAX** (Nicholas Chase, developerWorks, July 2003): Become an expert on SAX parsing by reading this tutorial.
- **More articles by this author** (Michael Galpin, developerWorks, April 2006-current): Read articles about XML, HTML 5, Eclipse, Apache Geronimo, Ajax, more Google APIs, and other technologies.
- **Android SDK**: Download the Android SDK. Version 1.5 and later will work for this article's exercise.
- **Android Open Source Project**: Get the open source code for Android.
- **Java Development Kit**: Download the complete platform and runtime environment for JDK 1.6.0_17, used in this article.
- **Protocol Buffers**: Download the protocol buffers compiler.
- **IBM certification**: Find out how you can become an IBM-Certified Developer.
- **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- **IBM product evaluation versions**: Get your hands on application development tools and middleware products.

© Copyright IBM Corporation 2010
**Trademarks**
(www.ibm.com/developerworks/ibm/trademarks/)