ABOUT ILIAS TSAGKLIS

Ilias is a software developer turned online entrepreneur. He is co-founder and Executive Editor at Java Code Geeks.

Android XML Binding with Simple Framework Tutorial

Posted by: Ilias Tsagklis in Android Core February 3rd, 2011

XML is still important in the area of web services even though REST has gained significant attention lately. Exposed APIs via web services is the main reason I have to manipulate XML content in my Android applications. For that reason I need to parse XML documents even though it is a tedious procedure.

In the past, I showed you how to parse XML using the SAX approach and how to boost your Android XML parsing with XML Pull. Both these techniques work, but are rather boring since they qualify as "plumbing code". In this tutorial I am going to show you how to perform XML binding in Android using the Simple framework.

XML data binding is quite popular in Java and there are multiple frameworks that allow binding. Solutions like JAXB and XStream are well established and heavily used. However, these libraries come with a large footprint, something that makes them inappropriate for use in the resources constraint world of mobiles. The good news is that there is a new kid on the block, the Simple framework. As its name implies, it strives to bring some simplicity in the bloated world of XML.

From the official Simple framework site:

Simple is a high performance XML serialization and configuration framework for Java. Its goal is to provide an XML framework that enables rapid development of XML configuration and communication systems. This framework aids the development of XML systems with minimal effort and reduced errors. It offers full object serialization and deserialization, maintaining each reference encountered.

Very nice. You can get started with the framework by visiting the documentation page and you should also read how does JAXB compare to Simple. Simple could change the way you handle XML in your Java applications, give it a try.

The big question is whether Simple is supported in Android's JVM. Android uses Dalvik, a specialized virtual machine optimized for mobile devices. Additionally, Dalvik use a subset of Apache's Harmony project for the core of its Class Library. Not all Java core classes are supported. For example, some of the javax.xml.* subpackages are not included.

Well, Simple CAN work with Android. More specifically, I managed to use **version 2.3.2** which can be found in Simple's download page. The corresponding JAR has a size of 287KB. The release notes for that version mention:

- Addition of DOM provider so that StAX is not a required dependency
- Fix made to ensure property defaults are applied correctly to classes

The first issue is very important because the StAX API is not included in Android's SDK. Note that **the latest versions of Simple (after v2.3.2) also work** and can be used for our purposes.

Let's cut to the chase and see how to perform the binding. As an example I will use an XML document that is returned as a response from the TMDb API which I use in the sample full Android application I build. Here is the document:

Movies search for "Transformers" and (year) "2007"

The response example can also be found here.

First of all download Simple version 2.3.2 and include it in your project's classpath. Then take a quick look at the Simple framework Javadocs.

The most important thing is to create our model objects and map the appropriately to the XML formatted document. If we take a look at the XML file, we shall see that the root element is called OpenSearchDescription and it includes a Query element, a "totalResults" element and a number of movies. Here how our main model class looks like:

```
01 package com.javacodegeeks.xml.bind.model;
02
03 import java.util.List;
```


NEWSLETTER

177,419 insiders are alre
weekly updates and complime
whitepapers!
Join them now to ga
access to the latest news in
as well as insights about Andr
Groovy and other related tech

Email address:

Your email address

Sign up

RECENT JOBS

Java Software Enginer
London, United Kingdom

JavaScript Developer
Dnipro, Ukraine

Java developer
Roth, Germany

Senior Software Engineer
New York City, New York

VIEW ALL

```
04
05 import org.simpleframework.xml.Element;
06 import org.simpleframework.xml.ElementList;
07 import org.simpleframework.xml.Root;
08
09 @Root
10 public class OpenSearchDescription {
11
12  @Element(name="Query")
13  public Query query;
14
15  @Element
16  public int totalResults;
17
18  @ElementList
19  public List<Movie> movies;
20
21 }
```

The Root annotation denotes that the specific class represents a root XML element. We also use the Element and ElementList annotations for the nested elements. Note that Simple can handle both "getters/setters" and "public fields" approaches. I use the latter in this example. One thing to be aware of is that we use the name field (for "Query") in order to provide the corresponding XML element name. This should be done when the XML element has a different name than the Java field, since Simple by default looks for an element with the same name as the field.

Let's now see the Query class:

```
01 package com.javacodegeeks.xml.bind.model;
02
03 import org.simpleframework.xml.Attribute;
04 import org.simpleframework.xml.Element;
05
06 @Element
07 public class Query {
08
09  @Attribute
10  public String searchTerms;
11
12 }
```

This class contains only an attribute called "searchTerms" so the relevant field is annotated with Attribute.

Very easy until now. Let's check the Movie class:

```
01 package com.javacodegeeks.xml.bind.model;
02
03 import java.util.List;
04
05 import org.simpleframework.xml.Element;
06 import org.simpleframework.xml.ElementList;
07
08 @Element(name="movie")
09 public class Movie {
10
11  @Element(required=false)
12  public String score;
13
14  @Element(required=false)
15  public String popularity;
16
17  @Element(required=false)
18  public String name;
19
20  @Element(required=false)
21  public String id;
22
23  @Element(required=false)
24  public String biography;
25
26  @Element(required=false)
27  public String url;
28
29  @Element(required=false)
30  public String version;
31
32  @Element(required=false)
33  public String lastModifiedAt;
34
35  @ElementList
36  public List<Image> images;
37
38 }
```

The only new thing is that the required field is used in order to declare that a field is not required (can be null). This is done because some fields are empty in the API response.

Let's see the Image class:

```
01 package com.javacodegeeks.xml.bind.model;
02
03 import org.simpleframework.xml.Attribute;
04 import org.simpleframework.xml.Element;
05
06 @Element(name="image")
07 public class Image {
08
09  @Attribute
10  public String type;
11
12  @Attribute
13  public String url;
14
15  @Attribute
16  public String size;
17
```

```
    @Attribute
    public int width;

    @Attribute
    public int height;

    @Attribute
    public String id;

}
```

This class includes only attributes so we annotate the fields accordingly.

The final step is to read the source XML and let Simple wire all the classes and populate the fields. This is done by using the Persister class which provides an implementation for the Serializer interface. We shall use its read method which reads the contents of the XML document from a provided source and convert it into an object of the specified type. Note that we have disabled the strict mode. Here is how it looks inside an Android Activity:

```
package com.javacodegeeks.xml.bind;

import java.io.IOException;
import java.io.Reader;
import java.io.StringReader;

import org.apache.http.HttpEntity;
import org.apache.http.HttpResponse;
import org.apache.http.HttpStatus;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.util.EntityUtils;
import org.simpleframework.xml.Serializer;
import org.simpleframework.xml.core.Persister;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;

import com.javacodegeeks.xml.bind.model.OpenSearchDescription;

public class SimpleExampleActivity extends Activity {

    private static final String url =
        "http://dl.dropbox.com/u/7215751/JavaCodeGeeks/AndroidFullAppTutorialPart03/Transformers+2007.xml";

    private DefaultHttpClient client = new DefaultHttpClient();

    @Override
    public void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        try {

            String xmlData = retrieve(url);
            Serializer serializer = new Persister();

            Reader reader = new StringReader(xmlData);
            OpenSearchDescription osd =
                serializer.read(OpenSearchDescription.class, reader, false);

            Log.d(SimpleExampleActivity.class.getSimpleName(), osd.toString());

        }
        catch (Exception e) {
            Toast.makeText(this, "Error Occured", Toast.LENGTH_LONG).show();
        }

    }

    public String retrieve(String url) {

        HttpGet getRequest = new HttpGet(url);

        try {

            HttpResponse getResponse = client.execute(getRequest);
            final int statusCode = getResponse.getStatusLine().getStatusCode();

            if (statusCode != HttpStatus.SC_OK) {
                return null;
            }

            HttpEntity getResponseEntity = getResponse.getEntity();

            if (getResponseEntity != null) {
                return EntityUtils.toString(getResponseEntity);
            }

        }
        catch (IOException e) {
            getRequest.abort();
            Log.w(getClass().getSimpleName(), "Error for URL " + url, e);
        }

        return null;

    }

}
```

This is a typical Android Activity. We retrieve the XML document as an internet resource (check my tutorial on how to use the HTTP API) and then create a StringReader from the response. We feed the Serializer with that and then let Simple perform its magic and return as a full class with the appropriate fields and embedded classes all populated. The specific app will just dump the classes string representations to the system's log, which can be monitored in the DDMS view.

That's all guys. No more manual XML parsing for me. As always, you can download the Eclipse project created for this tutorial.

Happy Android coding! And don't forget to share!

Related Articles:

- "Android Full Application Tutorial" series
- Boost your Android XML parsing with XML Pull
- Android Text-To-Speech Application
- Android Location Based Services Application – GPS location
- Install Android OS on your PC with VirtualBox

Tagged with: ANDROID TUTORIAL | SIMPLE | XML

5 COMMENTS

xe.sun
May 18th, 2012 at 4:06 am

05-18 12:05:35.832: W/System.err(930): org.simpleframework.xml.stream.NodeException: Document has no root element

Reply

esaulogbon
May 21st, 2012 at 12:46 pm

Great tutorial, and thanks a lot. Please, I have a question. If I want to post from my android app the same data to be stored in an external database using the web service, how do I do this?

Reply

51startup
February 13th, 2013 at 8:24 am

Job well done!

Although SimpleXml can work on Android, it's still a little heavy-weight for resource limited mobile device, anyway, SimpleXml originally was not designed with Android in mind.

I've built a light-weight xml binding framework tailored for Android platform, its called Nano, and I have adapted the author's original movie search full android application to use Nano binding instead, if you are interested, you can find Nano and the full adapted movie

search application by following links below:
https://github.com/bulldog2011/nano
http://bulldog2011.github.com/blog/2013/02/12/movie-search-android-app-using-nano/

Reply

wasim
October 14th, 2013 at 7:56 am

how can i read xml from assets folder ?

Reply

Meh
May 10th, 2014 at 11:43 pm

>> XML is still important in the area of web services even though REST has gained significant attention lately.

REST does not imply JSON. You can have a REST API using XML.

Reply

LEAVE A REPLY

Your email address will not be published. Required fields are marked *

Name *

Email *

Website

Sign me up for the newsletter!

Receive Email Notifications?

no, do not subscribe | instantly

Or, you can subscribe without commenting.

Post Comment

KNOWLEDGE BASE

- Courses
- Examples
- Resources
- Tutorials
- Whitepapers

PARTNERS

- Mkyong

THE CODE GEEKS NETWORK

- .NET Code Geeks
- Java Code Geeks
- System Code Geeks
- Web Code Geeks

HALL OF FAME

- "Android Full Application Tutorial" series
- 11 Online Learning websites that you should check out
- Advantages and Disadvantages of Cloud Computing – Cloud computing pros and cons
- Android Google Maps Tutorial
- Android JSON Parsing with Gson Tutorial
- Android Location Based Services Application – GPS location
- Android Quick Preferences Tutorial
- Difference between Comparator and Comparable in Java
- GWT 2 Spring 3 JPA 2 Hibernate 3.5 Tutorial
- Java Best Practices – Vector vs ArrayList vs HashSet

ABOUT JAVA CODE GEEKS

JCGs (Java Code Geeks) is an independent online community focused o
ultimate Java to Java developers resource center; targeted at the techni
technical team lead (senior developer), project manager and junior deve
JCGs serve the Java, SOA, Agile and Telecom communities with daily ne
domain experts, articles, tutorials, reviews, announcements, code snipp
source projects.

DISCLAIMER

All trademarks and registered trademarks appearing on Java Code Geek
property of their respective owners. Java is a trademark or registered tr
Oracle Corporation in the United States and other countries. Examples J
is not connected to Oracle Corporation and is not sponsored by Oracle C