XmlRpcExampleStringArray

Back to XML-RPC

Overview

This page contains some basic example code using a recent version of the xmlrpc libraries (3.1.2) to transfer an array of strings simulating a directory listing and an integer number representing a file count. There were a few examples on the web, but most used much older xmlrpc API's or were incomplete examples. The intent of this page is to provide a more complete starting point for a beginner with XML-RPC and less to go wrong.

This code originated on a linux system, using Java 1.5, Maven 2.2.0 and NetBeans 6.5 running apache-tomcat-6.0.16

This ended up a bit long, but I'm not sure if the server and client programs should be broken out into separate wiki pages. Sometimes it's nice to have it all together. This is my first foray into XML-RPC so improvements are welcome.

Example Configuration

Two programs are included in this example. The first listed is the client and the second part is the server, which will communicate via XML-RPC over HTTP with the server. The client is a command line application, while the server code is intended to run as a Tomcat servlet. These can both be run on the same system.

If you have the NetBeans IDE with web development plugins installed, then you can run the servlet with the integrated Tomcat. Otherwise, you can manually start up tomcat and then deploy the server program (myXmlRpcServer.war) to the `webapps` directory.

---

myXmlRpcClient Code

The code provided below is for a Maven 2 project.

Directory Structure

This is the standard maven project directory layout, it's listed here in case you aren't familiar with it.

```
myXmlRpcClient
 +- pom.xml
 `- src
     `- main
         `- java
             `- gov
                 `- noaa
                     `- eds
                         `- myXmlRpcClient
                             `- App.java
```

App.java Listing

Contents of **`src/main/java/gov/noaa/eds/myXmlRpcClient/App.java`** (Maven expects the file to be nested down in this directory hierarchy unless you configure it differently).

```
/*
 * FILE: App.java
 */
package gov.noaa.eds.myXmlRpcClient;

import java.lang.reflect.Array;
import java.net.MalformedURLException;
import java.net.URL;
import java.util.ArrayList;
import java.util.List;
import org.apache.xmlrpc.XmlRpcException;
import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;


/**
```

```
 * This client will use the myXmlRpcServer. *
 */
public class App {

    /**
     * Utility method to give an ArrayList when the returned XML-RPC
     * object is expected to be an array.
     *
     * @param element result object coming from a client.execute call.
     * @return a List or ArrayList. null is returned if the object is
     * of another type
     */
    public static List decodeList(Object element) {
        if (element == null) {
            return null;
        }
        if (element instanceof List) {
            return (List) element;
        }
        if (element.getClass().isArray()) {
            int length = Array.getLength(element);
            ArrayList result = new ArrayList();
            for (int i = 0; i < length; i++) {
                result.add(Array.get(element, i));
            }
            return result;
        }
        return null;
    }


    public static void main(String[] args) {
        int port = 8080;
        System.out.println("Starting myXmlRpcClient");
        String address = "http://127.0.0.1:" + Integer.toString(port) +
                "/myXmlRpcServer/xmlrpc";
        System.out.println("connecting to " + address);

        XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
        try {
            config.setServerURL(new URL(address));
        } catch (MalformedURLException ex) {
            ex.printStackTrace();
        }
        XmlRpcClient client = new XmlRpcClient();
        client.setConfig(config);
        Object[] params = new Object[] {new String("testDir")};
        try {
            Integer fileCount =
                    (Integer) client.execute("DirList.fileCount",
                    params);
            System.out.println("Client received fileCount=" +
                    fileCount.toString());
        } catch (XmlRpcException ex) {
            ex.printStackTrace();
        }

        try {
            /* OPTION A (next 5 lines):
             * This works, but looks ugly. This is how to get an array without
             * using a method like decodeList.
             */
//            Object[] result = (Object[]) client.execute("DirList.ls", params);
//            ArrayList<String> dirListing = new ArrayList<String>();
//            for (Object o : result) {
//                dirListing.add(o.toString());
//            }

            /* OPTION B (next 2 lines):
             * This works using decodeList()!
             */
```

```
            ArrayList<String> dirListing =
                    new ArrayList<String>(decodeList(
                    client.execute("DirList.ls", params)));

            System.out.println("Listing Length=" + dirListing.size());
            System.out.println("  First 10:");
            for (int i = 0; i < 10; i++) {
                System.out.println("    " + dirListing.get(i));
            }
        } catch (XmlRpcException ex) {
            ex.printStackTrace();
        }
    }
}
```

The port variable should be set to match the tomcat port running `myXmlRpcServer` (8080, 8084 if in NetBeans, maybe 8009 in Eclipse?). Thanks to Stanislav Miklik for the `decodeList` method.

pom.xml Project File (myXmlRpcClient)

The **`myXmlRpcClient/pom.xml`** file defines how the project is built for maven 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>gov.noaa.eds</groupId>
    <artifactId>myXmlRpcClient</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>myXmlRpcClient</name>
    <url>http://maven.apache.org</url>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
            <plugin>
                <!-- Usage: mvn assembly:assembly -->
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <archive>
                        <manifest>
                            <mainClass>gov.noaa.eds.myXmlRpcClient.App</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.xmlrpc</groupId>
            <artifactId>xmlrpc-client</artifactId>
            <version>3.1.2</version>
        </dependency>
    </dependencies>
</project>
```

You can see that the **maven-assembly-plugin** is specified, and the name of the `<mainClass>` "App" is configured here. If there is more than one class with an executable "main" method, this is how it is specified. This creates an executable jar file, with all the dependent libraries packaged in. This is not space efficient, but saves you having to get all the jar file dependencies into the Java class path.

Compiling

You should be at the top level of the source code tree for `myXmlRpcClient`, where the `pom.xml` file is located. Compile the source code with

```
mvn assembly:assembly
```

This will create an executable jar file in the standard `target` directory named `myXmlRpcClient-1.0-SNAPSHOT-jar-with-dependencies.jar`. (You can ignore the shorter/smaller myXmlRpcClient-1.0-SNAPSHOT.jar which does not have the libraries bundled in. Now might also be a good time to learn about filename completion 😉)

Running

Assuming you have successfully gotten tomcat running the myXmlRpcServer servlet (described below), use a command like this to run the example

```
java -jar target/myXmlRpcClient-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Sample Output

On a successful run you should see output like this:

```
Starting myXmlRpcClient
connecting to http://127.0.0.1:8080/myXmlRpcServer/xmlrpc
Client received fileCount=100000
Listing Length=100000
  First 10:
    sample_bwhmjcgwyxlqplhtk
    sample_doebwifflzjkdcyedhevenrp
    sample_jucqase
    sample_mmhogjqyabkueecrezxtgofbpqpwvrbctxhkiwse
    sample_q
    sample_kpzneuekyphzfqakfesfiklnfffctownhvm
    sample_cyeafzfpupqnfqomxjbdbupatdcaxh
    sample_kkjtqwbhsfejkyb
    sample_albkokpxazwuzerodddigdpkbxkmztckyp
    sample_eptoxnmckydxbiwhalljf
```

The first time you run this client against the server it will take longer, since the server generates the filenames on the first request. If you run this client again, you should see the same names, but quicker. If you restart the myXmlRpcServer service or restart tomcat the list of filenames should change.

On my system 1 million files take 6 seconds the first time, and 3 seconds thereafter, so 3 seconds are used to generate 1M random names and about 3 seconds are needed to send 1M files between programs (on the same system). If the client and server are on separate systems, then your network bandwidth should make the transport time go up quite a bit.

If you don't have the port number set right in `App.java` then you will see a long list of exceptions starting with

```
org.apache.xmlrpc.XmlRpcException: Failed to read server's response: Connection refused
```

If you don't have the service name correct in App.java, like "/myXmlRpcSer*~~typo~~*ver/*bad-*xmlrpc", you will see this error:

```
org.apache.xmlrpc.client.XmlRpcHttpTransportException: HTTP server returned unexpected status: /myXmlRpcSer-
typo-ver/bad-xmlrpc
```

---

myXmlRpcServer Code

The code provided below is for a Maven 2 project.

Directory Structure

This is the standard maven project directory layout, it's listed here in case you aren't familiar with it.

```
myXmlRpcServer
 +- pom.xml
 `- src
     +- main
     |   +- java
     |   |   +- gov
     |   |   |   `- noaa
     |   |   |       `- eds
     |   |   |           `- myXmlRpc
     |   |   |               `- DirList.java
     |   |   `- org
     |   |       `- apache
     |   |           `- xmlrpc
     |   |               `- webserver
     |   |                   `- XmlRpcServlet.properties
     |   `- webapp
     |       +- META-INF
     |       |   `- content.xml
     |       +- WEB-INF
     |       |   `- web.xml
     |       `- index.jsp
     `- test                                       \
         `- java                                    |
             `- gov                                 |
                 `- noaa                             > Optional
                     `- eds                         |
                         `- myXmlRpc                |
                             `- DirListTest.java   /
```

DirList.java Listing

Contents of **`src/main/java/gov/noaa/eds/myXmlRpcServer/DirList.java`** (Maven expects the file to be nested down in this directory hierarchy unless you configure it differently).

```
/*
 * FILE: DirList.java
 */
package gov.noaa.eds.myXmlRpc;

import java.util.ArrayList;
import java.util.Random;


/**
 * Provide directory listing functionality.
 */
public class DirList {

    static boolean listInitialized = false; // for lazy initialization
    static int listLength = 100000;
    static ArrayList<String> listing;


    public int fileCount(String dirName) {
        // fileCount for directory dirName
        // Always returns listLength until real code is written
        return listLength;
    }


    /**
     * Return a directory listing.
     * Currently generates made up names.
     * @param dirName directory name for which to get a listing (currently ignored)
     * @return a list of filenames for dirName
     */
    public ArrayList<String> ls(String dirName) {
        if (!DirList.listInitialized) {
            listing = new ArrayList<String>(listLength);
            Random rng = new Random(); // Random Number Generator
            for (int i = 0; i < listLength; i++) {
                int filenameLen = 1 + rng.nextInt(40);
                StringBuffer filename = new StringBuffer("sample_");
                for (int f = 0; f < filenameLen; f++) {
                    filename.append("abcdefghijklmnopqrstuvwxyz".charAt(rng.nextInt(26)));
                }
                listing.add(filename.toString());
            }

            DirList.listInitialized = true;
        }
        return listing;
    }
}
```

Rather than get a real directory listing and have to deal with filesystem specifics, this example creates a list of randomly generated file names, all starting out with "sample_". The list is only created once, so the same list will be returned until the web service is restarted. The second time the client retrieves the list will be without the time needed to generate the random file names, allowing you to judge the performance of just the data transfer over XML-RPC. The `listLength` variable controls how many file names to generate.

XmlRpcServlet.properties Listing

Contents of **`src/main/java/org/apache/xmlrpc/webserver/XmlRpcServlet.properties`**. A resources entry in the pom.xml file (below) includes this into the warfile.

```
## As derived from the example on http://ws.apache.org/xmlrpc/server.html
#Calculator=org.apache.xmlrpc.demo.Calculator

## This connects my class to the "DirList" name used by the client.execute method
DirList=gov.noaa.eds.myXmlRpc.DirList
```

Note that the filename **`XmlRpcServlet.properties`** as well as the directory structure `org/apache/xmlrpc/webserver` should not be changed. Thes `XmlRpcServlet.properties` file should end up in the resulting warfile under `WEB-INF/classes/org/apache/xmlrpc/webserver/XmlRpcServlet.properties`, and the xmlrpc libraries will not find it if these are changed.

web.xml Listing

Contents of **`src/main/webapp/WEB-INF/web.xml`**.

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
    <display-name>Archetype Created Web Application</display-name>
    <servlet>
        <servlet-name>myXmlRpcServer</servlet-name>
        <servlet-class>org.apache.xmlrpc.webserver.XmlRpcServlet</servlet-class>
        <init-param>
            <param-name>enabledForExtensions</param-name>
            <param-value>true</param-value>
            <description>
            Sets, whether the servlet supports vendor extensions for XML-RPC.
            </description>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>myXmlRpcServer</servlet-name>
        <url-pattern>/xmlrpc</url-pattern>
    </servlet-mapping>
</web-app>
```

You can choose a `<servlet-name>` that you like, using the same value in the `<servlet>` and `<servlet-mapping>` sections. Leave the `<servlet-class>` element as-is though. The `<servlet-name>` and `<url-pattern>` will affect the URL you put into the client application (App.java above) so choose URL friendly names, with no spaces or unusual punctuation.

index.jsp Listing

Contents of **`src/main/webapp/index.jsp`**. This is a minor part of the example.

```
<html>
<body>
<h2>XML-RPC Server is running</h2>
</body>
</html>
```

This pops up when running the servlet in NetBeans, otherwise you probably will not see it. You can visit http://127.0.0.1:8080/myXmlRpcServer to see if the servlet is running. It prints the "XML-RPC Server is running" page when it's active.

content.xml Listing

Contents of **`src/main/webapp/META-INF/content.xml`**. This is a minor part of the example.

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/myXmlRpcServer"/>
```

DirListTest.java Listing

Contents of **`src/test/java/gov/noaa/eds/myXmlRpcServer/DirListTest.java`**. This is an optional JUnit test file.

```
/*
 * FILE: DirListTest.java
 */

package gov.noaa.eds.myXmlRpc;

import java.util.ArrayList;
import junit.framework.TestCase;

/**
 * JUnit Test file
 */
public class DirListTest extends TestCase {

    public DirListTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }


    /**
     * Test of fileCount method, of class DirList.
     */
    public void testFileCount() {
        System.out.println("fileCount");
        String s1 = "always100000";
        DirList instance = new DirList();
        int expResult = 100000; // needs to match listLength in DirList.java
        int result = instance.fileCount(s1);
        assertEquals(expResult, result);
    }


    /**
     * Test of ls method, of class DirList.
     * Results are random, so don't look at specifics.
     * 100000 filenames starting with "sample_" will be generated.
     */
    public void testLs() {
        System.out.println("ls");
        String dirName = "100000files";
        DirList instance = new DirList();
        ArrayList<String> result = instance.ls(dirName);
        int errorCount = 0;
        int totalLength = 0;
        int fileCount = 0;
        for (String fn : result) {
            fileCount++;
            totalLength += fn.length();
            if (! fn.startsWith("sample_")) {
                errorCount++;
            }
        }
        int avgLength = totalLength / fileCount;
        System.out.println("  " + fileCount + " files, average name length="
                + avgLength);
        System.out.println("  First 10:");
```

```
        for (int i = 0; i < 10; i++) {
            System.out.println("    " + result.get(i));
        }
        assertEquals(errorCount, 0);
    }

}
```

pom.xml Project File (myXmlRpcServer)

The **`myXmlRpcServer/pom.xml`** file defines how the project is built for maven 2:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>gov.noaa.eds</groupId>
    <artifactId>myXmlRpcServer</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>myXmlRpcServer Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.xmlrpc</groupId>
            <artifactId>xmlrpc-server</artifactId>
            <version>3.1.2</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>myXmlRpcServer</finalName>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                </configuration>
            </plugin>
        </plugins>
        <resources>
            <resource>
                <directory>src/main/java</directory>
                <includes>
                    <include>org/apache/xmlrpc/webserver/*.properties</include>
                </includes>
            </resource>
        </resources>
    </build>
    <properties>
        <netbeans.hint.deploy.server>Tomcat60</netbeans.hint.deploy.server>
    </properties>
</project>
```

The resource entry with `<directory>src/main/java</directory>` and the line with `<include>org/apache/xmlrpc/webserver/*.properties</include>` in the resources section tells maven to include `the XmlRpcServlet.properties` file into the warfile WEB-INF/classes directory with the directory hierarchy that the xmlrpc library is expecting.

Compiling

You should be at the top level of the source code tree for `myXmlRpcServer`, where the `pom.xml` file is located. Compile the source code with

```
mvn clean package
```

This will create a jar file in the standard `target` directory with a name of `myXmlRpcServer.war`. Near the bottom of the output should be a `[INFO] BUILD SUCCESSFUL` message, indicated all is well.

If you did include the JUnit test file (`XmlRpcServlerTest.java`), then the output should contain something like this:

```
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /extra/data/src/java/NetBeans_projects/myXmlRpcServer/src/test
/resources
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to /extra/data/src/java/NetBeans_projects/myXmlRpcServer/target/test-classes
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: /extra/data/src/java/NetBeans_projects/myXmlRpcServer/target/surefire-reports

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running gov.noaa.eds.myXmlRpc.DirListTest
fileCount
ls
  100000 files, average name length=27
  First 10:
    sample_wtzgfxlwdukohv
    sample_vgwsucz
    sample_xwkt
    sample_rryyitdcxs
    sample_sric
    sample_vfaoionq
    sample_tqfqcdwrg
    sample_dmvczjmqrntqlwvthfxqdwcuspvhwnggxmfeze
    sample_rovqppvnofhrmxooqeoigyadts
    sample_jogldvknhzotyt
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.333 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Running

Apache tomcat is fairly easy to install if it's not already on your system, you can usually install it in your home directory for personal use if you don't have root (administrator) permissions. You can run the server in your IDE, like NetBeans with Web Application plugins installed, or in a standalone Tomcat. Note that the port number may vary depending on which method you use, set the port in the client (App.java) to match. Manually starting the service should bring it up on port 8080, inside NetBeans the port chosen tends to be 8084 so as not to conflict with any standalone tomcat that might running on the usual 8080.

Use a command like this to start up tomcat manually if not already running:

```
~/bin/tomcat/bin/startup.sh
```

Adjust the path to match your system. On my system I've set up a link called `tomcat` in my `bin` directory as an alias for `/extra/data/bin2/apache-tomcat-6.0.16/` which is where tomcat is installed.

Put a copy of your application into the webapps directory. The default installation of tomcat will notice the new warfile and start running it. If tomcat on your system is configured differently then you will need to manually request the warfile to start as a new service.

```
cp target/myXmlRpcServer.war ~/bin/tomcat/webapps/
```