LoggingwithXindice

Writing to Xindice with Log4J Appender

by Jim Fuller jim.fuller@ruminate.co.uk Jan 12th, 2004

Log4j allows logging requests to be piped to multiple destinations. A log4j output is known as an appender. Log4j provides a slew of appenders writing to the console, files, remote socket servers, JMS, NT Event Loggers, remote UNIX Syslog daemons, etc... .

An appender is linked to a logger through either calling configuration API or defining a seperate property file. In addition, you can define the layout of an event; e.g. I have been using the XML Layout option for awhile now (even though I really need to hack out the use of CDATA) and I wondered how easy it would be to pipe all log4j events to be written into Xindice.

I wanted to be able to change all my existing log property files to use a new Appender called XindiceAppender, which meant that all logs would be placed within an Xindice db collection. I also required to have an xml file written everytime for backup purposes.

The steps to achieve this are as follows;

- Create addDocument xindice object to write to Xindice: use addDocument.java included in Xindice documentation and amend it for our purposes.
- Create log4j XindiceAppender: use WriterAppender.java included in Log4J documentation and src distribution and amend it for our purposes.
- Define log4j property file: Use an xml log4j property file that attatches our new XindiceAppender to log

As noted above, I will hack up the addDocument.java example that comes with the xindice documentation. I have placed all my objects in the com. example.xindice package.

Firstly we need to have an object that takes care of the writing to Xindice.

addDocument.java

```
package com.example.xindice;

import org.xmldb.api.base.*;
import org.xmldb.api.modules.*;
import org.xmldb.api.*;
import java.lang.RuntimeException;

import java.io.*;

public class addDocument {

  /** Log event. */
  private String _Event= null;

  /** Current server hosting xindice. */
  private String _Host= null;

  /** Deprecated var. */
  private String _FileName=null;

  /** The path of the current collection. */
  private String _Collection=null;

  /** The value to be used for log4j namespace. */
  private String _Namespace=null;


  public void setEvent(String event) {
      String val = event.trim();
      _Event = val;
  }

  public String getEvent()  {
      return _Event;
  }

  public void setFileName(String filename) {
      String val = filename.trim();
      _FileName = val;
  }

  public String getFileName()  {
      return _FileName;
```

```
  }

  public void setCollection(String collection) {
      String val = collection.trim();
      _Collection = val;
  }

  public String getCollection()  {
      return _Collection;
  }

  public void setNamespace(String namespace) {
      String val = namespace.trim();
      _Namespace = val;
  }

  public String getNamespace()  {
      return _Namespace;
  }

  public String getHost()  {
      return _Host;
  }

  public void setHost(String host) {
      String val = host.trim();
      _Host = val;
  }

//because our methods throws exceptions and subAppend doesnt we hack our way through with unchecked exceptions
   public void writeEvent() throws RuntimeException {

   try {
               write(_Event,_Collection,_Namespace,_Host);
    }catch(Exception e){
        System.err.println("Exception occured ");    }
    }

public static void write(String event, String collection, String namespace,String host) throws Exception {

  Collection col = null;
  try {

    String driver = "org.apache.xindice.client.xmldb.DatabaseImpl";
    Class c = Class.forName(driver);
    Database database = (Database) c.newInstance();
    DatabaseManager.registerDatabase(database);
    col = DatabaseManager.getCollection("xmldb:xindice://"+host+":8080"+collection);

    String data ="<?xml version='1.0'?><log4j:log xmlns:log4j='"+namespace+"'>"+event+"</log4j:log>";

    XMLResource document = (XMLResource) col.createResource(null,"XMLResource");
    document.setContent(data);
    col.storeResource(document);
    System.out.println("xml document added");
  }
  catch (XMLDBException e) {
    System.err.println("XML:DB Exception occured " + e.errorCode);
  }
  finally {

  if (col != null) {
      col.close();
    }

  }
}

}
```

The write method does all the heavy lifting, with the entry point for the logger to use writeEvent. There are set methods for defining the event, namespace to use for logger xml, and collection path.

We have had to use unchecked exceptions because of log4j architecture, this is simply through desire to illustrate things simply here. As you will see in the Appender description, we subclass from the log4j Appender class, which would have needed modification to handle our exception events properly.

The next file is the XindiceAppender, its just a hacked up version of WriterAppender.java that comes with log4j. WriterAppender wrote event data to a file, as previously stated I want to retain this capability, but also add writing to Xindice. Log4j can use multiple appenders so this is not neccesary in production, though when testing the code I needed to verify that events were actually being fired off.

XindiceAppender.java

```
/*
 * Copyright (C) The Apache Software Foundation. All rights reserved.
 *
 * This software is published under the terms of the Apache Software
 * License version 1.1.  */

package com.example.xindice;

import java.io.IOException;
import java.io.Writer;
import java.io.FileOutputStream;
import java.io.BufferedWriter;

import org.apache.log4j.*;
import org.apache.log4j.spi.ErrorCode;
import org.apache.log4j.helpers.QuietWriter;
import org.apache.log4j.helpers.LogLog;
import org.apache.log4j.spi.LoggingEvent;

import com.example.xindice.addDocument;

// XindiceWriter is based on modified WriterAppender code
// Author: Jim Fuller <jim.fuller@ruminate.co.uk>
//
// Contibutors: Jens Uwe Pipka <jens.pipka@gmx.de>
//              Ben Sandee

/**
  *  XindiceAppender appends log events to a file and to an Xindice instantiation.
  *
  * @author Jim Fuller
  * @author Ceki G&uuml;lc&uuml;
  * @since 1.1
  */

public class XindiceAppender extends WriterAppender {

  /** Append to or truncate the file? The default value for this
      variable is <code>true</code>, meaning that by default a
      <code>XindiceAppender</code> will append to an existing file and
      not truncate it.

      <p>This option is meaningful only if the XindiceAppender opens the
      file. This has no affect on xindice.
  */
  protected boolean fileAppend = true;

  /**
     The name of the log file. */
  protected String fileName = null;

  /**
     Flag to turn on xindice writing. */
  protected String writeXindice = null;

  /**
     Flag to turn on file writing. */
  protected String writeFile = null;
```

```
  /**
     The namespace to use in generated xml. */
  protected String log4jNamespace = null;

  /**
     The server hosting xindice. */
  protected String Server = null;

  /**
     The name of the collection path. */
  protected String Collection = null;

/**
     instantiate com.epinx.xindice.addDocument object. */
  public addDocument xindicedb = new addDocument();

  /**
     Do we do bufferedIO? */
  protected boolean bufferedIO = false;

  /**
     How big should the IO buffer be? Default is 8K. */
  protected int bufferSize = 8*1024;

  /**
     The default constructor does not do anything.
  */
  public
  XindiceAppender() {
  }

  /**
    Instantiate a <code>XindiceAppender</code> and open the file
    designated by <code>filename</code>. The opened filename will
    become the output destination for this appender.

    <p>If the <code>append</code> parameter is true, the file will be
    appended to. Otherwise, the file designated by
    <code>filename</code> will be truncated before being opened.

    <p>If the <code>bufferedIO</code> parameter is <code>true</code>,
    then buffered IO will be used to write to the output file.

    <p>Writing to xindice is performed in the subAppend method
  */
  public
  XindiceAppender(Layout layout, String filename, boolean append, boolean bufferedIO,
                int bufferSize) throws IOException {
    this.layout = layout;
    this.setFile(filename, append, bufferedIO, bufferSize);
  }

  /**
    Instantiate a XindiceAppender and open the file designated by
    <code>filename</code>. The opened filename will become the output
    destination for this appender.

    <p>If the <code>append</code> parameter is true, the file will be
    appended to. Otherwise, the file designated by
    <code>filename</code> will be truncated before being opened.
  */
  public
  XindiceAppender(Layout layout, String filename, boolean append)
                                                              throws IOException {
    this.layout = layout;
    this.setFile(filename, append, false, bufferSize);
  }

  /**
     Instantiate a XindiceAppender and open the file designated by
    <code>filename</code>. The opened filename will become the output
```

```
  destination for this appender.

  <p>The file will be appended to.  */
public
XindiceAppender(Layout layout, String filename) throws IOException {
  this(layout, filename, true);
}

/**
   The <b>File</b> property takes a string value which should be the
   name of the file to append to.

   <p><font color="#DD0044"><b>Note that the special values
   "System.out" or "System.err" are no longer honored.</b></font>

   <p>Note: Actual opening of the file is made when {@link
   #activateOptions} is called, not when the options are set.  */

public void setFile(String file) {
  // Trim spaces from both ends. The users probably does not want
  // trailing spaces in file names.
  String val = file.trim();
  fileName = val;
}

/**
    set the value of writeXindice flag.
 */
public void setwriteXindice(String writexindice) {
  String val = writexindice.trim();
  writeXindice = val;
}

/**
    set the value of log4jNamespace value.
 */
public void setlog4jNamespace(String log4jnamespace) {
  String val = log4jnamespace.trim();
  log4jNamespace = val;
}

/**
    set the value of server value.
 */
public void setServer(String server) {
  String val = server.trim();
  Server = val;
}

/**
    set the value of writeFile flag.
 */
public void setwriteFile(String writefile) {
  String val = writefile.trim();
  writeFile = val;
}

/**
    set the value of Collection value.
 */
public void setCollection(String dbcollection) {
    String val = dbcollection.trim();
    Collection = val;
}

/**
    Returns the value of the <b>Append</b> option.
 */
public
boolean getAppend() {
  return fileAppend;
```

```
  }


  /** Returns the value of the <b>File</b> option. */
  public
  String getFile() {
    return fileName;
  }

  /**
     If the value of <b>File</b> is not <code>null</code>, then {@link
     #setFile} is called with the values of <b>File</b>  and
     <b>Append</b> properties.

     @since 0.8.1 */
  public
  void activateOptions() {
    if(fileName != null) {
      try {
        setFile(fileName, fileAppend, bufferedIO, bufferSize);
      }
      catch(java.io.IOException e) {
        errorHandler.error("setFile("+fileName+","+fileAppend+") call failed.",
                           e, ErrorCode.FILE_OPEN_FAILURE);
      }
    } else {
      //LogLog.error("File option not set for appender ["+name+"].");
      LogLog.warn("File option not set for appender ["+name+"].");
      LogLog.warn("Are you using XindiceAppender instead of ConsoleAppender?");
    }
  }

 /**
     Closes the previously opened file.
  */
  protected
  void closeFile() {
    if(this.qw != null) {
      try {
        this.qw.close();
      }
      catch(java.io.IOException e) {
        // Exceptionally, it does not make sense to delegate to an
        // ErrorHandler. Since a closed appender is basically dead.
        LogLog.error("Could not close " + qw, e);
      }
    }
  }

  /**
     Get the value of the <b>BufferedIO</b> option.

     <p>BufferedIO will significatnly increase performance on heavily
     loaded systems.

  */
  public
  boolean getBufferedIO() {
    return this.bufferedIO;
  }

  /**
     Get the size of the IO buffer.
  */
  public
  int getBufferSize() {
    return this.bufferSize;
  }

  /**
     The <b>Append</b> option takes a boolean value. It is set to
```

```
   <code>true</code> by default. If true, then <code>File</code>
   will be opened in append mode by {@link #setFile setFile} (see
   above). Otherwise, {@link #setFile setFile} will open
   <code>File</code> in truncate mode.

   <p>Note: Actual opening of the file is made when {@link
   #activateOptions} is called, not when the options are set.
 */
public
void setAppend(boolean flag) {
  fileAppend = flag;
}

/**
   The <b>BufferedIO</b> option takes a boolean value. It is set to
   <code>false</code> by default. If true, then <code>File</code>
   will be opened and the resulting {@link java.io.Writer} wrapped
   around a {@link BufferedWriter}.

   BufferedIO will significatnly increase performance on heavily
   loaded systems.

*/
public
void setBufferedIO(boolean bufferedIO) {
  this.bufferedIO = bufferedIO;
  if(bufferedIO) {
    immediateFlush = false;
  }
}

/**
   Set the size of the IO buffer.
*/
public
void setBufferSize(int bufferSize) {
  this.bufferSize = bufferSize;
}

/**
  <p>Sets and <i>opens</i> the file where the log output will
  go. The specified file must be writable.

  <p>If there was already an opened file, then the previous file
  is closed first.

  <p><b>Do not use this method directly. To configure a XindiceAppender
  or one of its subclasses, set its properties one by one and then
  call activateOptions.</b>

  @param fileName The path to the log file.
  @param append   If true will append to fileName. Otherwise will
      truncate fileName.  */
public
synchronized
void setFile(String fileName, boolean append, boolean bufferedIO, int bufferSize)
                                                          throws IOException {
  LogLog.debug("setFile called: "+fileName+", "+append);

  // It does not make sense to have immediate flush and bufferedIO.
  if(bufferedIO) {
    setImmediateFlush(false);
  }

  reset();

  Writer fw = createWriter(new FileOutputStream(fileName, append));

  if(bufferedIO) {
    fw = new BufferedWriter(fw, bufferSize);
  }
```

```
  this.setQWForFiles(fw);
  this.fileName = fileName;
  this.fileAppend = append;
  this.bufferedIO = bufferedIO;
  this.bufferSize = bufferSize;

  writeHeader();
  LogLog.debug("setFile ended");

}

/**
   Sets the quiet writer being used.

   This method is overriden by {@link RollingXindiceAppender}.
 */
protected
void setQWForFiles(Writer writer) {
   this.qw = new QuietWriter(writer, errorHandler);
}


/**
   Close any previously opened file and call the parent's
   <code>reset</code>.  */
protected
void reset() {
  closeFile();
  this.fileName = null;
  super.reset();
}

  /**
     Actual writing occurs here.

     <p>Most subclasses of <code>WriterAppender</code> will need to
     override this method.


     @since 0.9.0 */

  protected void subAppend(LoggingEvent event) {


      // if writeXindice flag equal to 'yes' then write to xindice db
      if(writeXindice.equals("yes")){

                  xindicedb.setEvent(this.layout.format(event).toString());
                  xindicedb.setCollection(Collection);
                  xindicedb.setNamespace(log4jNamespace);
                  xindicedb.setHost(Server);
                  xindicedb.writeEvent();

          }

      if(writeFile.equals("yes")){

      if(layout.ignoresThrowable()) {
        String[] s = event.getThrowableStrRep();

        if (s != null) {
          int len = s.length;


          for(int i = 0; i < len; i++) {
            this.qw.write(s[i]);

            this.qw.write(Layout.LINE_SEP);
          }
        }
```

```
        }

        if(this.immediateFlush) {
          this.qw.flush();
        }
    }
 }
}
```

The code in subAppend() will write the event, set db collection path, namespace to be used with xml, and finally fire off writeEvent.

```
   xindicedb.setEvent(this.layout.format(event).toString());
   xindicedb.setCollection(Collection);
   xindicedb.setNamespace(log4jNamespace);
   xindicedb.writeEvent();
```

Once again there is plenty of room for optimisation and improvement, though I thought breaking them up would assist in understanding the process. The additional flags and vars handle the extra requirements of writing to Xindice.

- writeXindice: if set to yes will write event to xindice
- writeFile: if set to yes will also write event to file

The Collection and log4jNamespace vars define respectively the db collection path and namespace to use in the generated event xml. Basically XindiceAppender should work in the same manner as WriterAppender with the added benefit of writing to xindice.

As with any log4j you will need to invoke the logging code with the appropriate configuration. OK you will still need to attatch this from within the code that is logging something...which is pure log4j stuff;

```
    Logger root = Logger.getRootLogger();
    DOMConfigurator.configure("c:\\log4j.properties.xml");
```

note- remember to use DOMConfigurator with xml log4j prop files

So lets now show an example property configuration file.

property.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    <appender name="myAppender" class="com.epinx.xindice.XindiceAppender">
        <param name="File" value="log.xml" />
        <param name="Collection" value="/db/test" />
        <param name="writeXindice" value="yes" />
        <param name="writeFile" value="no" />
        <param name="Server" value="127.0.0.1" />
        <param name="log4jNamespace" value="http://jakarta.apache.org/log4j/" />
        <layout class="org.apache.log4j.xml.XMLLayout" />
    </appender>
    <root>
        <priority value="info" />
        <appender-ref ref="myAppender" />
    </root>
</log4j:configuration>
```

The com.example.xindice.XindiceAppender appender should write to whatever collection you have defined in xindice. In the case above, the logger is looking for a /db/log collection. Appropriate errors will be thrown for malformed xml.

With the addition of a server variable, we can direct output to a variety of remote Xindice repositories. Note that each event gets written as a single document within xindice, and since I have ommited explicitly naming the documents...I settled for xindice's method of automatically naming a file.

I have done little testing though everything seems to work fine, there will perhaps be conflicts with Xindice's own usage of Log4j, in addition it remains to be seen if xindice can handle such high volume, small transaction type data handling.

Improvements for the future could be;

- prescribe xindice xml document name
- append option to append to existing file

- add additional xml meta data
- add xslt interception
- add port option
- add authentication

Good luck, Jim Fuller