Jsub XML

Josh Hone
June 14, 2002

Introduction:

I have written and modified some Java classes for the purpose of generating Jsub request scripts from information encoded inside a JsubXML form. Jsub is the Jefferson Lab scripting language that submits a programmer's programs to the processor farm for fast processing of their "jobs". A typical JsubXML form might contain data looking like this:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE JOBREQUEST SYSTEM "http://www.jlab.org/~hone/XML/Jsub.dtd">
<JOBREQUEST>
  <COMMAND>run_a1c</COMMAND>
  <PROJECT>clas</PROJECT>
  <QUEUE NAME="low_priority"/>
  <JOB>
    <JOBNAME>run1.f</JOBNAME>
    <QUEUE NAME="production"/>
  </JOB>
  <JOB>
    <JOBNAME>mark_test</JOBNAME>
  </JOB>
</JOBREQUEST>
```

Once this markup and its information are passed to the Java program, it generates zero to many Jsub scripts (in direct correlation to the number of <JOB> tags) by parsing the markup with a Java-based Apache XERCES parser, using the SAX parsing functions. Documentation for the parser can be found at http://xml.apache.org/xerces-j/index.html and linking to either the API docs or the others listed in the left-hand column as you have need.

XML

The Jsub eXtensible Markup Language (XML) is a markup structure designed to hold all information that the Jsub submission system could possibly use in running a script.

What is XML?

XML is a natural choice to encode this information because the language allows a programmer to create a markup language to match almost any situation-specific lingo. It employs descriptive tag names, describes relationships among tags, and can ensure that the data in a document is used in more or less the fashion for which it was intended.

A Document Type Definition (DTD) uses XML syntax to define tag names, attributes, interrelationships, and other properties. They are referred to in a declaration at the beginning of every XML document. An XML document that matches with the rules provided by the DTD is said to be valid, and, if it is written in good markup style, well-formed. The parsing program checks for both. For example, a DTD can declare that the legal value a tag may contain is PCDATA, which stands for parsed character data. Then it may declare that a tag's attribute may contain CDATA, which is just PCDATA for attributes, indicating that in attributes the character data is not parsed, or removed of leading and trailing whitespace. However, it should not allow bad markup grammar, such as ending an outer element before an inner element is ended.

What is Jsub XML?

The Jsub DTD can be found, thoroughly commented, by a link at http://www.jlab.org/~hone/ , and in the CLAS CVS repository under tools/JsubParser. This DTD defines the structure and the main intent of JsubXML. All Jsub keywords have been encoded in Jsub XML and can be specified by using this markup language. A list of Jsub keywords and their respective functions can be found at: http://cc.jlab.org/docs/scicomp/how-to/keywords.html. So, to *use* this XML structure, we will write a web interface with easy-to-use forms that a potential farm user will fill out.

Jsub XML-Jsub Command Keywords

There is more or less a direct connection between the names of the Jsub XML tags and the names and/or functions of the corresponding Jsub keywords. First, a <JOBREQUEST> is the root tag for this type of document. It contains all (ANY) other tags. The Tag Name column lists the set of existing JsubXML tag names. The Jsub Keyword column lists the corresponding keyword in the JSUB scripting language, if any. The Tag Value-Keyword Value column maps the possible values of the tag, as allowed by the tag definitions in the DTD, to the type of information that it becomes in the resulting JSUB script. The Description field is a set of comments on the meaning of the tag. Here is a list to match the rest together:

Tag Name	Jsub Keyword	Tag Value - Keyword Value	Description
COMMAND	COMMAND	PCDATA-character data	Points to the user script.
COMMAND – COPY att.	COMMAND_COPY	TRUE/FALSE – present/not present	Copy the command file to the local

			farm disk.
PROJECT	None	EXTENSION	Holds PROJECT info.
PROJECT – NAME att.	PROJECT	CDATA-character data	Project name allotted the farm time.
EXTENSION	None	EXTENSION	Utility tag for specifying subprojects.
EXTENSION – NAME att.	None	PCDATA-none	Subproject name.
USER	None	PCDATA-none	Utility to specify the user name.
JOBNAME	JOBNAME	PCDATA-character data	Gives jobs a label inside Jsub.
QUEUE	None	Empty	Contains queue name information.
QUEUE – NAME att.	QUEUE	IDLE, LOW_PRIOIRITY, PRIORITY, PRODUCTION	Specifies the job queue on the farm.
TIME	TIME	PCDATA- integer value	Maximum time the job can run on the farm.
TIME – UNITS att.	Converts TIME value.	SECONDS, MINUTES, HOURS, DAYS	Unit of time the value is specified in.
OS	None	Empty	Contains OS specification
OS – NAME att.	OS	ANY, AIX, SOLARIS, LINUX, HPUX	Tells which OS your farm job uses.
MAIL	MAIL	PCDATA-character data	Email addresses to which Jsub can mail job status reports.
OPTIONS	OPTIONS	PCDATA-character data	Gives the command line options to the user script.
INPUT	INPUT_FILES	PCDATA-character data	Names of job input files stored on the tape drive.

INPUT – LOCAL att.	INPUT_DATA	CDATA-character data	Name every input file should be called locally on each farm node.
REPETITIONS	MULTI_JOBS	PCDATA- integer value	Number of times a job should run if zero or one INPUT_FILE is given.
OTHER_FILES	OTHER_FILES	PCDATA-character data	Other job input files not coming from the tape silo.
OTHER_FILES-LOCAL att.	None	CDATA-none	Possible future inclusion to specify what each other file should be called on the local farm node.
OUTPUT	OUTPUT_TEMPLATE	PCDATA-character data	Name of the output file copied back to the silo.
OUTPUT – LOCAL att.	OUTPUT_DATA	CDATA-character data	Name of the output file on the local farm node.
OUTPUT – DEST att.	TOWORK, TOTAPE	WORK, TAPE – TOWORK, TOTAPE	Destination of output files – the working disk or the tape silo.
JOB	None	ANY	Each separate JOB becomes a Jsub script. Local qualities can be described apart from global ones here.
JOB – SINGLE_JOB att	SINGLE_JOB	FALSE, TRUE – not present, present	If present, process all INPUT_FILES with one job.

Java

Why Java?

Perl might seem like a more natural choice to accomplish this task, since it is very good at outputting and formatting text and in connecting to some databases. Java was chosen for a number of reasons. One is that most people use Java to do XML parsing, and it is easy to find examples and usable source code for Java XML parsers on the web and in programming books. Second, it seemed like a more natural tool for writing a web tool that can be well-supported in the future. Third, Java is platform-independent, since you use a Java compiler from Sun to compile the code. A program compiled on any machine can run on any other machine as long as they use compatible Java Runtime Environments (JREs). Fourth, we can use Java DataBase Connectivity (JDBC) classes to connect, with the same source code in the calls, to any database. This partially arises from the platform independence of Java. The intent is to connect to any of JLab's many databases with this program to build more useful Jsub scripts from the same web interface request.

What happens?

A command line user of this program will simply say:

java JsubParser.JsubParser [name of XML file to be parsed]

The program will generate Jsub scripts, ready for submission, in the local directory. The syntax of this can be explained piece by piece:

java: This is the command to execute a Java program.

JsubParser.JsubParser: This syntax says that you want to execute the main function of a class called JsubParser(the second name) in package JsubParser(the first name). To execute this program, strict file name and directory name guidelines must be maintained. See "Setting CLASSPATH" for rules and regulations of this process.

Setting CLASSPATH

Usage of the Java classes from the command line begins with setting an environment variable called CLASSPATH. On a typical Unix system you would add a line to your .cshrc file (C shell resource, I think…)

setenv CLASSPATH /pathname/jarname.jar:/pathname/JavaPackageContainer

Then you run the command:

source .cshrc

This ensures that the changes are implemented by your environment.

What does all this mean? The environment variable CLASSPATH, which can also be set on the *java* command line using the flag –classpath, tells the runtime environment where to find the various classes that are referenced in the source code. It is also used in

compilation with the *javac* command to find all classes that the main method in the compiled source file uses.

A peculiarity in Java is that the source file which contains a class must be named only a certain way: classname.java (whatever the class name is). This extends to directories and packages as well. A package, which is a collection of classes, must be contained in a directory of the same name as the package. For example, the JsubParser package contains all classes that make up the package JsubParser: JsubContentHandler, JsubErrorHandler, JsubParser, and JsubScript. They must be in a directory called JsubParser. When CLASSPATH is set, it must point to the directories containing the classes and packages that a program needs. Then, when the program compiles the class with the main method that the programmer wants to invoke, it will automatically compile all classes that this program needs. It can find those classes by the paths provided by the CLASSPATH.

It is easy to point to too little, but it is also easy to point to too much. The java compiler will compile only the classes that are referenced by the main routine, directly or indirectly. From those it will compile all found from the CLASSPATH paths. Then, in the runtime environment, the program will encounter whatever problems are incurred from not including code definitions of compiled classes. However, if a CLASSPATH points to two files with the same name, then the compiler will not know which the program is meant to include. Neither of the conflicting files will be compiled, and your program may suffer while running from the absence of the machine code from those files.

What is in the Java code?

The base source code comes from an Apache XERCES parser that can parse in either a SAX (Simple API for XML) or DOM (Document-Object Model) fashion. The formula to make either work is to override underlying parsing functions with your own code that handles events as they happen. However, a SAX parses XML"nodes": it will simply load, node by node, the document in sequence. A programmer rewrites the functions that deal with each node. Examples of nodes include elements, character data, and processing instructions. A DOM loads the entire document into memory and the functions move around the document tree. A SAX Parser is used in this code.

Starting with the sample Apache program, SAX functions were written to parse the beginning and end of elements, the handling of any element's attributes, and the character data inside an element. Global variables were added to the JsubContentHandler class, where the functions were written, to add DOM functionality. These carried names of current elements and some information about the tree structure further up from the node currently being parsed. This way the simple SAX sequenced processing of nodes was also given some history about the current node.

A container class was developed to parse each element and attribute encountered. Each JsubScript is built as the SAX moves along. The program first builds a global

JsubScript, which contains the properties of the request that will be shared by all Jsub scripts. The global is then copied to a local JsubScript, created by each new instance of a <JOB> tag, and the program appends the properties specific to that request to the local JsubScript. This class has member functions that parse each element and attribute in JsubXML into meaningful Jsub script keywords. Therefore, to change what an element means, change the corresponding member function. This class offers a localized way to interpret JsubXML tags.

The other two classes are important but less complicated. The class containing the main function, JsubParser, simply reads in a URI which locates the document to parse, creates instances of XMLReaders, ContentHandlers, and ErrorHandlers, and applies them appropriately to the document at the URI. The ErrorHandler describes how to deal with mistakes in an XML document's structure, and it defines three levels of error: warning, error, and fatal error. The parser cannot recover from a fatal error and must stop parsing. So, if your document looks really bad, it will stop at your first bad error. Otherwise, it will output your errors and warnings to the standard output device, wherever you choose to direct it.

The Future

Xerces, the Java XML parser, is just one piece of software in the Apache Jakarta project, as noted above. The general development goal of Jakarta is to write Java packages that work very well with XML and can run inside Apache web servers. Cocoon, Jetspeed, Ant, Velocity, Turbine, and Tomcat are all programs that aid in creating spaces on web servers for applets and servlets to run and to parse and interpret XML along the way. The JsubXML project will be extended using these. The goal of this work is to create web pages that detail what is happening of interest to a particular user inside a processor farm. The web server will receive XML from the farm as jobs run that can be sent in the form of web pages to qualified users who are allowed to see the data. The web pages will be personalized, much like My Yahoo and other popular web sites, by allowing a user to specify what type of information they are interested in. Out of a channel, or stream of data, the information relevant to a registered user will be selected and then configured according to the style they prefer. The immediate next few steps include XML structures to define the next few layers of abstraction away from JsubXML, as we work to solve user needs further from the server side and closer to the client side. One example is an XML structure that would detail the pieces of a file name on tape that is considered CLAS data. A user filling out this form can then use the parsing process as a search engine to find all runs they are interested in, from matching either name patterns or run characteristics from the online database. Another project is to create a relational database to hold this XML, to make searching and recovering important JsubXML easier.

Acknowledgements

Thanks to Larry Dennis (dennisl@phy.fsu.edu) and Mark Ito (marki@jlab.org) for providing the motivation and supervision of this project.