JoranConfigurator DRAFT

Ceki Gülcü, 2003-2004, © All rights reserved

Joran stands for a cold north-west wind which, every now and then, blows forcefully on Lake Leman. Contrary to what its name might lead to believe, the Leman lake bears little relation to lascivious debauchery. Located right in the middle of Europe, it happens to be the continent's largest sweet water reserve.

Introduction

One of the most powerful features of the Java language is reflection. Reflection makes it possible to configure software systems declaratively. For example, many important properties of an EJB are configured with the *ejb.xml* file. While EJBs are written in Java, many of their properties are specified within the *ejb.xml* file. Similarly, log4j logging settings can be specified in a configuration file, expressed either in key=properties format or in XML.

The `DOMConfigurator` that ships with log4j version 1.2.x can parse configuration files written in XML. The `DOMConfigurator` is written in Java such that each time the structure of the configuration file changes the `DOMConfigurator` must be tweaked accordingly. Moreover, the modified code must be recompiled and redeployed. Just as importantly, the code of the `DOMConfigurator` consists of loops dealing with children elements containing many interspersed if/else statements. One can't help but notice that that particular code reeks of redundancy.

The digester project[1] has shown that it is possible to parse XML files using pattern matching rules. At parse time, digester will apply the rules that match previously stated patterns. Rule classes are usually quite small and specialized. Consequently, they are relatively easy to understand and to maintain.

Joran is largely inspired by the commons-digester project but uses a slightly different terminology. In commons-digester, a rule can be seen as consisting of a pattern and a rule, as shown by the `Digester.addRule(String pattern, Rule rule)` method. I find it unnecessarily confusing to have a rule to consist of itself, not recursively but with a different meaning. In Joran, a rule consists of a pattern and an action. An action is invoked when a match occurs for the corresponding pattern. This relation between patterns and actions lies at the core of Joran.

[1] See *http://jakarta.apache.org/commomns/digester* for the digester project.

Quite remarkably, one can deal with quite complex situations by using simple patterns, or more precisely with exact matches and wildcard matches. For example, the pattern "`a/b`" will match a `<b>` element nested within an `<a>` element but not a `<c>` element, even if nested within a `<b>` element. It is also possible to match a particular XML element, regardless of its nesting level, by using the "*" wildcard character. For example, the pattern "`*/a`" will match an `<a>` element at any nesting position within the document. Other types of patterns, for example "`a/*`", are not currently supported by Joran.

SAX or DOM?

Due to the event-based architecture of the SAX API, a tool based on SAX cannot easily deal with forward references, that is, references to elements which are defined later than the current element being processed. Elements with cyclical references are equally problematic. More generally, the DOM API allows the user to perform searches on all the elements and make forward jumps.

This extra flexibility initially led me to choose the DOM API as the underlying parsing API for Joran. After some experimentation, it quickly became clear that dealing with jumps to distant elements while parsing the DOM tree did not make much sense when the interpretation rules were expressed in the form of patterns and actions. *Joran only needs to be given the elements in the XML document in a sequential, depth-first order.*

Joran was first implemented in DOM. However, the author migrated to SAX in order to benefit form the location[2] information provided to the user, that is, to an `org.w3.sax.ContentHandler`. With the help of location information, it becomes possible to display convenient error reports to the user which include exact line and column. This extra information turns out to be convenient in hunting down problems.

Availability

Joran will ship as part of log4j version 1.3 which is not yet released as of April 10th 2004. Joran and accompanying examples are only available from the log4j project

[2] The location information is the line and columns numbers corresponding to various elements or attributes within the XML document.

CVS repository[3]. However, an *alpha* release of log4j 1.3 is expected to be available within the next few days.

Actions

Actions extend the `org.apache.joran.action.Action` class which consists of the following abstract methods.

```
public abstract class Action {

 /**
  * Called when the parser first encounters an element.
  */
  public abstract void begin(ExecutionContext ec,
                             String name,
                             Attributes attributes);

 /**
  * Called when the parser encounters the element end. At
  * this stage, we can assume that child elements, if any,
  * have been processed.
  */
  public abstract void end(ExecutionContext ec, String name);
```

Thus, every action must implement the `begin` and `end` methods.

Execution context

To allow various actions to collaborate, the invocation of `begin` and `end` methods include an execution context as the first parameter. The execution context includes an object stack, an object map, an error list and a reference to the Joran interpreter invoking the action. Please see the `org.apache.joran.ExecutionContext` class for the exact list of fields contained in the execution context.

Actions can collaborate together by fetching, pushing or popping objects from the common object stack, or by putting and fetching keyed objects on the common object map. Actions can report any error conditions by adding error items on the execution context's error list.

[3] See http://logging.apache.org/site/cvs-repositories.html

A hello world example

The *examples/src/joran/helloWorld/* directory includes a trivial action and Joran interpreter setup which just diaplays “hello world” when a `<hello-world>` element is encountered in an XML file.

Look into this example to learn about the basic steps which are necessary to set up and invoke a Joran interpreter.

Collaborating actions

The *examples/src/joran/calculator/* directory includes several actions which collaborate together through the common object stack in order to accomplish simple computations.

New-rule action

Joran includes an action which allows the Joran interpreter to lean new rules on the fly while interpreting the XML file containing the new rules.

See the *examples/src/joran/newRule/* directory for sample code.

Implicit actions

The rules defined thus far are called explicit rules because they require an explicit pattern, hence fixing the tag name of the elements for which they apply.

In highly extensible systems, the number and type of components to handle are innumerable so that it would become very tedious or even impossible to list all the applicable patterns by name.

At the same time, even in highly extensible systems one can observe well-defined patterns linking the various parts together. Implicit rules come in very handy when processing components composed of sub-components unknown ahead of time. For example, Apache Ant is capable of handling tasks which contain tags unknown at compile time by looking at methods whose names start with *add*, as in `addFile`, or `addClassPath`. When Ant encounters an embedded tag within a task, it simply instantiates an object that matches the signature of the task class' *add* method and attaches the resulting object to the parent.

Joran includes similar capability in the form of implicit actions. Joran keeps a list of implicit actions which can be applied if no explicit pattern matches the current

XML element. However, applying an implicit action may not be always appropriate. Before executing the implicit action, Joran asks an implicit action whether it is appropriate in the current context. Only if the action replies affirmatively does Joran interpreter invoke the (implicit) action. This extra step makes it possible to support multiple implicit actions or obviously none, if no implicit action is appropriate for a given situation.

For example, the `NestedComponentIA` extending `ImplicitAction`[4], will instantiate the class specified in a nested component and attach it to the parent component by using setter method of the parent component and the nested element's name.

Refer to the *examples/src/joran/implicit* directory for an example of an implicit action.

Non goals

The Joran API is not intended to be used to parse documents with thousands of elements.

[4] Both `ImplicitAction` and `NestedComponentIA` are located in the `org.apache.joran.action` package.