24 Parsing complex log files

Processes such as webservers generate log files with long complex lines. We sometimes need to extract data from these, but TDI does not have a 'webserver logfile' parser. This problem can be solved using the script parser.

1. Create a new AL called ProcessWebLog.
2. Add a filesystem connector in iterator mode, and set it to read the file `web-log-sample`
3. Add a Script parser to the connector. In the parser, click the *Edit Script* button and look at the template that has been supplied.
   You should find three methods (functions): *writeEntry, readEntry,* and *querySchema. writeEntry* is only used in output mode: we are interested in the other two.
4. Go to the input map and connect to the data source. You should see one attribute appear in the schema: *line* – this is the only one mentioned in the *querySchema* method.
5. Click the *Next* button: you should see a new line from the logfile in the *line* attribute each time. This is not very useful as a parser, but it does show that the connector is working.
6. Edit the parser script again, and change the *querySchema* method to look like this:

```
function querySchema ()
{
   var attrs = ["line","domain","ip"];
   for (var i=0; i< attrs.length; i++) {
      var e = new com.ibm.di.entry.Entry();
      e.addAttributeValue("name",attrs[i]);
      e.addAttributeValue("syntax","String");
      list.add(e);
   }
   result.setStatus (1);
}
```

   The idea is to create several Entry objects, each describing an attribute for the schema. All the attributes will be strings.
7. Return to the input map, close the connection and re-connect. You should see new schema attributes *domain* and *ip*.
8. Now we need to start parsing the actual log lines. Regular Expressions provide a powerful text-matching facility that will help.
   Look at the *readEntry* method. You will see that it reads a line of text into *str* and then does some tests before calling *entry.setAttribute* to put the line into the *line* attribute in the *entry* object. The *Script Parser* chapter in the TDI Reference Guide describes how the parser communicates with the rest of the system.
   Add this after the line that says `counter++;`

```
var re = new RegExp("^([^ ]+) ([^ ]+) ");
var attrs = re.exec(str);

entry.setAttribute ("domain", attrs[1]);
entry.setAttribute ("ip", attrs[2]);
```

This does a lot of work in a few lines, so it is worth looking at it carefully.

The first line defines a regular expression that parses the first few items on the line. It saves some of the matched text into variables that can be used later. A rough description of the expression is:
Starting with the first character of the line, match at least one non-space character and save the matched characters.
Match exactly one space.
Match at least one non-space character and save the matched characters.
Match exactly one space.
Ignore the rest of the line.
For a full description of how regular expressions work, see the JavaScript Core Reference. Most books covering Perl or Unix command-line utilities will also have sections on regular expressions.

The second line executes the expression against the input line (*str*). If the pattern matches, the saved text is copied into elements of the array *attrs*. In this case there will be two elements because there are two 'saves'. The elements we want are indexed 1 and 2, as the whole string is copied into element 0 of the array.

We now have the first two element of the log line in the *attrs* array. The last two lines of the code fragment copy this data into the *domain* and *ip* attributes in *entry*.

9. Save the script and return to the input map. Close and re-connect, then click the *Next* button to see what happens. You should see data from the logfile appearing in the Sample Value column.
10. Extend the parser to extract the date field as well.
11. Add a DumpEntry script component to your assembly line, and run it to check that you get *domain, ip,* and *date* in the results.

24.1 Handling active logfiles

Logfiles on an active system grow continuously, and it is often useful to analyse them in real time. This means that we need to read all the lines in the file and then wait for more to be appended, rather like the Unix `tail -f` command.

1. Go to the Connection tab of your ReadLogFile connector. Expand the Advanced section at the bottom.

2. Click on the question-mark beside the Timeout box and read the description. Set a timeout of 0 (zero) so that the connector will wait forever for new lines to appear in the file.
3. Change the file path to `/home/student/files/log`
4. Now we need to simulate a webserver logfile. The script `dripfeed` is provided for this purpose:

```
cd ~/files
dripfeed web-log-sample > log
```

   The effect of this is to copy one line from the sample file to *log* every two seconds. You can see what is going on by running this in another window:

```
tail -f ~/files/log
```

5. Run your assembly line. You should see some entries reported very quickly, then the rate will slow down to one every two seconds.
6. In a practical system, instead of just dumping the work object we would probably analyse the entries and keep running counts. Another assembly line could make the data available via HTTP or SNMP.