StatisticsAndLog4jIdeas

Statistics and log4j

Some info and ideas about log4j and appenders etc that might be relevant

What I'm groping for here, in reference to Richard's proposal, is a 'third way' - a way to use log4j and a db appender of some sort to log events and their properties directly to the db. It's looking good, too - we might have to wait for the next version of log4j, but that shouldn't be long.

Now log4j - current version is 1.2.12 - has a JDBCAppender class, but warns that "This version of JDBCAppender is very likely to be completely replaced in the future."

About version 1.3 log4j it says-
Version 1.3 is going to contain some very extensive changes and new features. You should expect a number of api changes.
Expected timeframe: October, 2005
More info is here - http__--logging.apache.org-log4j-docs-plan.html.

So there is a 1.3 alpha out there now - which has instead org.apache.log4j.db.DBAppender and org.apache.log4j.db.DBAppender2 (not sure what the latter is for...).

The new DBAppender logs to three predefined tables that we would have to add to our schema. Namely -
logging_event, logging_event_property, logging_event_exception

logging_event_property is for storing the properties of LoggingEvents (org.apache.log4j.spi.LoggingEvent). So these LoggingEvent properties sound kind of like what we need - they are just like the parameters that Richard highlights in his proposal - so that's looking interesting. But it wasn't too clear how you actually put one into a LoggingEvent. (The DBAppender is responsible for getting them in the db of course.)

After much getting my head round of some advanced log4j concepts I finally realised that using a Mapped Diagnostic Context (MDC) is the best way to do this. This stores properties which are thread local - so we can define them for just one logging event or for a longer period - in a map (obviously). If we can get them into the event this way, the DBAppender will take care of the rest. But will this be a nice way of logging for people? Well it's pretty easy - here's an e.g.

```
                        // gather some client specific information
                        String remAddr = req.getRemoteAddr();
                        String remHost = req.getRemoteHost();

                        //put it into the MDC for logging
                        MDC.put("RemoteHost", remHost);
                        MDC.put("RemoteAddress", remAddr);

                        // get the comments
                        String comments = req.getParameter("comments");

                        if(comments != null){

                                log.info("Comments: " + comments);
                                req.setAttribute("comments", comments);
                        }else{

                                log.error("Comments are null !!");
                        }

                        //get it out of the MDC
                        MDC.remove("RemoteHost"); //or could remove it later... or not at all... or do an MDC.
clear()
                        MDC.remove("RemoteAddress");
```

(slightly adapted from tutorial here - http://www.onjava.com/pub/a/onjava/2002/08/07/log4j.html?page=last

```
which uses an older version of log4j etc though)
```

Anyway, the relevant logging event properties are then automagically put into the db when the event buffer is flushed and the logs written (or so we're led to believe... actually the logj4 code does look as if it will do that...).

There's an important distinction between actions and parameters for our DSpace usage, that isn't made by these event properties. Most significant logging events will reflect a particular action, and we don't want to log real events (like an access of an item) multiple times by accident. We need a particular loggingEvent property that defines the action. If we were to call the MDC directly it would look like this -

```
MDC.put("action", "items_by_author");
Log.info("browse of items by author");
MDC.remove("action");
}}}

So where does all that leave our dspace logging lines like this?

<pre>
log.info(LogManager.getHeader(context, "items_by_author", logInfo  + ",result_count=" + browseInfo.
getResultCount())); }}}

Not sure. We could still do with a LogManager class I guess but it would have work a bit differently. I suppose
a call to -
<pre>
 LogManager.setContext(context);
```

could be made to ensure various data are in the MDC.

Then you do -

```
LogManager.setAction("items_by_author");
LogManager.setParam("result_count", browseInfo.getResultCount());
log.info(logInfo); actual logging line...arg may be strangely redundant
LogManager.clearAction();
```

Not sure what else we would need to have LogManager do, I guess this -

```
LogManager.clearParam("result_count");
```

```
LogManager.clearAllParams()
```

could clear everything that is not gleaned from the context and that we always want to retain

```
 LogManager.clearAll();
```

could clear the action and all the params, to save typing - but again leave the stuff gleaned from the context etc.

Note that -

```
LogManager.setParam(key, value)
```

might just delegate to MDC but could check a registry of param keys on the way.
Equally

```
 LogManager.setAction("view_stuff")
```

could check a registry of **actions**.

Okay - what would all that look like in a moderately complex DSpace case (ItemsByAuthorServlet):

```
        // Get the HTTP parameters
         String author = request.getParameter("author");
         String order = request.getParameter("order");

         LogManager.setContext(context);


         // How should we order the items?
         boolean orderByTitle;

         if ((order != null) && order.equalsIgnoreCase("title"))
         {
             orderByTitle = true;
             LogManager.setParam("order", "title");
         }
         else
         {
             orderByTitle = false;
             LogManager.setParam("order", "date");
         }

         // Get the community or collection scope
         Community community = UIUtil.getCommunityLocation(request);
         Collection collection = UIUtil.getCollectionLocation(request);

         if (collection != null)
         {
             LogManager.setParam("collection_id", collection.getID());
             scope.setScope(collection);
         }
         else if (community != null)
         {
             LogManager.setParam("community_id", community.getID());
             scope.setScope(community);
         }

         // Ensure author is non-null
         if (author == null)
         {
             author = "";
         }

         LogManager.setParam("author_string", author);

         // Do the browse
         scope.setFocus(author);

         BrowseInfo browseInfo = Browse.getItemsByAuthor(scope, orderByTitle);

         LogManager.setParam("result_count", browseInfo.getResultCount());

         //log the browse
         LogManager.setAction("items_by_author");
         log.info("browsing by author - but you know that!!");
         LogManager.clearAll();
```

(This is not significantly more complex than it already is.)

I think that's basically that - so it's a plan of sorts - maybe the log4j mailing lists can help too, and we should post there to check it's not an insane idea. And this post has various ways of doing interesting stuff with logs - though not exactly what we want -

http://marc.theaimsgroup.com/?l=log4j-user&m=108019994415866&w=2