StatisticsFurtherSpeculations

<?xml version="1.0" encoding="utf-8"?>
<html>

Logging Notes

From looking at Liam's thoughts here, I would agree that it makes sense to use the log4j system with the DAppender to manage the deposit of logs in the database. This should basically give us the pros from both the previous suggestions I made without any significant cons.
Further to this, then, I have been thinking about how this might be best utilised in DSpace, and think that taking Liam's LogManager suggestions further might be the best solution. That is, we would encapsulate all logging actions inside our own LogManager class, so as to insulate the user from interfacing directly with log4j and it's MDC concept. A class speicifcation might be:
{{

Unknown macro: {LogManager -- public static int INFOpublic static int WANNpublic static int EOpublic static int DEUGpublic Logger log -- public LogManager(Class class)public void setContext(Context context)public void setParam(String key, String value)public void setAction(String action)public void log(int level, String comment)public void clearParams()public void clearParam(String param)public void clearAction()public void clearAll()private String getHeader()}}

}
So this would basically provide an interface to the MDC class, but also will provide an interface to the logging action. The purpose of this is to move knowledge of how logging actually works, and it's specific relationship to log4j away from the user (i.e. the module writer). Shamelessly stealing Liam's example, this would then implement as:
{{

Unknown macro: {LogManager log = new LogManager(request.class);log.setContext(context);log.setAction("items_by_author");log.setParam("result_count", browseInfo.getesultCount());log.log(LogManager.INFO, logInfo);log.clearAll();}}

}

Constrained actions and parameters

I am having trouble deciding if constraining parameters and actions for logging is a good idea or not. It would enforce consistent logging, at the cost of perhaps annoying the programmer, having to jump through some loops to log an activity, and not being able to change a logging message structure without making modifications elsewhere (probably in another class). Nonetheless, here is something a bit more solid regarding this idea, should it be neccessary.
The following database table is the trivial method of maintaining a link between allowed actions and parameters
{{

Unknown macro: {log_properties----------action -- the name of the allowed actionproperty -- the name of the allowed property for that action}}

}
se we just hold a basic map of actions to properties which we can validate against. I imagine the following LogParams class mediating with this table in various ways
{{

Unknown macro: {LogParams -- public static boolean validate(String action, String param)public static boolean validate(String action)public static boolean validate(LogManager log)public static String getParams(String action)public static void register(String action, String param)}}

}
So during a logging process the following calls to this class might be made:
{{

Unknown macro: {public class LogManager{public void setParam(String key, String value){if (!LogParams.validate(MDC.get("action"), key)){throw new LogParameterException(); // for example}else{MDC.put(key, value);}}public void setAction(String action){if (!LogParams.validate(action)){throw new LogParameterException(); // for example}else{MDC.put("action", action);}}}}

}
I've also put a register() method in there so that modules could provide XML files containing their logging actions and parameters, which could be read in at build time, and pushed into the database through this class.
The point of this, as I see it, is that we get to enforce consistent logging standards for each logging action, and also have a registry of actions and parameters that will allow for the building of reports more easily. For example, we could have a search interface on the reports section that allows you to select an action, and make a search based on one of the parameters. With a registry of actions and parameters this would be a straightforward exercise. Also, it may be useful for building tables of results showing actions and selected parameters, and combining different logging events with eachother through like parameters.

Does this section go to far?

I don't know if this is a step too far, but here it is for consideration anyway.
As DSpace becomes more modular, it may be sensible to abstract this logging process even further, so that modules can implement their own customised logging features. I briefly brought this up in my last post on this, but here it is a little more formalised. asically I am wondering whether we should encourage each module to extend the LogManager class to create a ModuleLogger (e.g. TapirLogger). The idea would be to have this do all the leg work with regard to logging on a module by module basis, and to keep logging code clear from the main logic. For example:
{{

```
Unknown macro: {public class TapirLogger extends LogManager{public void editItem(int itemID){this.setAction("edit_item");this.setParam("item_id", Integer.toString(itemID));this.log(LogManager.INFO, "an item has been edited");this.clearAction();this.clearParam("item_id");}}}}
```

}
Stealing one of Liam's examples, again, this would make the main source code look like this during the logging process:
{{

```
Unknown macro: {rowseLogger log = new rowseLogger(rowseServlet.class);// Get the HTTP parameters String author = request.getParameter("author");String order = request.getParameter("order");log.setContext(context);// How should we order the items?boolean orderTitle;if ((order != null) && order.equalsIgnoreCase("title")){orderTitle = true;}else{orderTitle = false;}// Get the community or collection scopeCommunity community = UUtil.getCommunity(request);Collection collection = UUtil.getCollection(request);if (collection != null){log.setParam("collection_id", collection.getID());scope.setScope(collection);}else if (community != null){log.setParam("community_id", community.getID());scope.setScope(community);}// Ensure author is non-null;if (author.b1 = null){author = "";}// Do the browseBrowseInfo browseInfo = rowse.getItemsyAuthor(scope, orderyTitle);//log the browselog.browse(orderyTitle, author, browseInfo.getesultCount());}}
```

}
This combines actions of both directly setting parameters, and passing them wholesale to the module specific log manager.
Question is, is this just too much abstraction, to the point where it's just a pain to use? Answers on the back of a Wiki ... Yet Further Statistics & Logging Speculations this time from Liam ==
In response to the above I have an number of comments/ideas....
Wrapping the actual "log" call in the LogManager means that log4j would not be able to record line numbers (they would all be line number of code in the LogManager class....). ut that's probably not important in the whole scheme of things - I don't think we log the line nos now anyway.... and I think reducing the pollution of functional code by logging gubbins is probably worth this drawback.
I kind of liked the idea of the subclasses of LogManager - but after some consideration I think it's creating too much of a burden on the module programmer. I think adding log actions should be just a config change, that install-configs should take care of.
ut we should take steps to make logging as convienient as possible. asically we want the logging an action to be a single line to be possible - though we also want people to be able to gather params as they go if they want. For single line I suggest -
{{

```
Unknown macro: {log.logActionWithParams("browse_items_by_author","author_id", "1234","collection_id", "1234"); }}
```

}
i.e.
{{

```
Unknown macro: {log.logActionWithParams(<action id><param1 id>, <param1 value>,.... )}}
```

}
achieved using the Java 5.0 (String ... args) syntax - it's gonna be validated anyway.... so should be ok...
for logging over multiple lines I suggest -
{{

```
Unknown macro: {log.addActionParam("collection_id", "1234"); log.addActionParam("community_id", "1234"); log.logAction("items_by_author", "message if any");}}
```

}
Where logAction() will validate first, and clear the action and all action params but not contextual params after it's done. And one should be able to use both techniques at the same time. So this way the validation "signature" (i.e. which params are required) is in the config/db and not duplicated in a java method signature, but we still have more or less the same convenience factor.
I think we need to distinguish easily between action-related parameters and ones that are really context-related (thread level) and allow people to put contextual ones in easily, at any time, as it may be useful. So a separate -
{{{ log.addContextualProp("session_id", "12345"); }}}
would be good, and anything added like this should not be flushed. One way would be for the LogManager to have a Stack containing which logging action params have been added, and pop everything off it, clearing them in the MDC as it goes, when it's done logging the action (so leaving everyhting else).
We might also want to facilitate addhoc logging for debugging purposes with a log.debug() method that logs at debug level and doesn't have an "action" property (but records any contextual props plus any action params already set...).
The allowed actions and their params should probably be stored in memory while the app runs and not require a database trip - e.g. Map of LogAction objects each containing a list of params, read from the D at app initialisation (i.e. this is the data of LogParams class - though might be better to call it LogActionegistry or something). We could validate the MDC directly, just prior to a log call - i.e. pull the action value, and check that all required params are there. E.g. LogActionegistry.validate(MDC)
Just to clarify how the LogManager class would work, given all of that. The client class instantiates a static reference to an instance of the LogManager e.g.:
private static LogManager log = LogManager.getInstance(MyClass.class);
Each instance of LogManager needs a reference to the log4j logger for the class it's logging for -
{{

Unknown macro: {private Logger logger4j;logger4j = Logger.getLogger(inClass);}}

}
ut all the rest of the data owned by LogManager is either static or thread-level. The LogActionegistry is static. The paramStack would be thread level and managed directly, and MDC properties are thread level but managed by the MDC. I think that will work okay - the clients will use LogManager much like they use log4j anyway.
Just had another thought - again inspired by the browse servlet - some parameters are intrinsic to the action, some are optional and their presence may indicate a subtype of the action (e.g. collection id in the example). In order to assist (potentially interactive) reporting, all significant subtypes of actions should normally be differentiated by the value of a parameter specifically for that - e.g. if there was simply "browse" action it would need a "browse_type" param to indicate whether it was by date/author/titles. This would be okay as long as the parameter set stayed roughly the same between the different action subtypes (so not sure the browse thing is a good example but you get the idea). To accommodate the minor differences we can have optional parameters like community_id or whatever - but they should also be registered, to facilitate interactive reporting. They allow finer discrimination on reporting. ut it might actually be good to have a standard action_subtype field in the database table itself, in order to group semantically similar actions with substantially different parameter sets - otherwise you'd have to choose between losing this grouping (or basing it on a partial string match), or breaking the process of validation of params. I dunno really - someone who's actually written a stats system already would be best off deciding about this...
Another thought - re logging levels - at what log level an action is logged could be configurable. I.e. some admins might never need to report on certain actions normally so make them only get recorded at debug level - and you shouldn't need to change the code for this. Assuming the database trip is the expensive thing, this would fit in with the above. When the log call is made, it uses the log level associated with that action that it's got out of the logActionegistry config.
Putting all this together it terms of the db we've got something like -
{{

Unknown macro: {log_action_parameters log_action_parameter_id -- PK action_id -- the id of the allowed actionparameter_name -- the name of the allowed property for that actionrequired -- boolean -- does validation check this is actlog_actions log_action_id -- PK log_action_name -- name of actionlog_action_subtype -- ??? subtype of action for which this property appliesslog_level -- under what conditions do we want this logged}}

}
(emember this is not accessed on the fly by the running app)
One last thing - we need to decide whether to call them logging params or logging properties as we keep changing our minds (at least I do).... I guess the right way to think of it is: log actions have params, while the MDC log lines have properties (which are a superset that includes the "action":<action name> name-value pair, all the log action params and contextual properties...). So I guess some confusion is unavoidable but we can limit it by taking care over the names....

</html>