CCLRC

Technical Report
DL-TR-2004-002

ENHANCING THE DATA PORTAL TO A PRODUCTION LEVEL ENVIRONMENT

G. Drinkwater

19th August 2004

Council for the Central Laboratory of the Research Councils

Abstract: *The project aims to provide easy, transparent access to experimental, observational, simulation and visualisation data kept on a multitude of systems and sites. Further more it will provide links to other web/grid services, which will allow the scientists to further use the selected data, e.g. via data mining, simulations or visualisation. The Data Portal will aim to work as a broker between the scientists, the facilities, the data and other services. The problem addressed is that currently the scientific data is stored distributed across a multitude of sites and systems. Scientists have only very limited support in accessing, managing and transferring their data or indeed in identifying new data resources. In a true Grid environment, it is essential to ease many of these processes and the aim of the Data Portal is to help with automating many of these tasks.*

TABLE OF CONTENTS:

TABLE OF FIGURES:

1. INTRODUCTION

The storage and management of scientific data generated at its facilities is an important responsibility of CCLRC. The full value of these data resources will only be realised if they are easily searchable, accessible and reusable. The aim of this project is to develop the means for a scientist to explore these data resources, discover the data they need and retrieve the relevant datasets through one interface and independent of the data location.

The Data Portal is currently on its third version. The first was a prototype built on J2EE technology, mainly servlets. The code was redesigned from scratch for the next version but still used J2EE with a Model, View, Controller paradigm (MVC). Java Server Pages (JSP) for the presentation, JavaBeans as business logic (Model) and servlets for the Controller. The current release used the implementation of the previous release but split the Data Portal in to areas of functionality, each with a web service interface. This allowed the code to be modularised and to take advantage of web service technology but created many problems, including integration of the modules, the additional complexity to install and configure all the modules, installation of all the new software and libraries and the added problem of a UDDI registry server for the lookup of Data Portal web services.

The solution to the problems that the web services architecture caused and the enhancement of the Data Portal to a production level piece of software are explained and examined in this paper. It outlines what are the code and software engineering problems and what improvements are needed.

2. CURRENT PROBLEMS WITH VERSION 3.X

2.1 Logging

One problem with Data Portal 3.x is the use of logging. This is because each module was written independently from other modules with different logging techniques and ways to log errors, thus creating problems with the Data Portal as different modules report their errors independently and log similar errors with different importance levels, causing confusion and complications with debugging.

Logging is currently being used in several different ways. The standardisation of the modules logging system with clearly defined error messages and logging of these messages is needed. The current manners of logging within the Data Portal are as follows:

- *Log4j*. This is a module created by Apache within the Jakarta project [1]. It allows logging at runtime without modifying the application binary. Logging behaviour is controlled by editing a configuration file, without touching the application binary. The target of the log output can be a file, an `OutputStream`, a `java.io.Writer`, a remote log4j server, a remote Unix Syslog daemon, a swing GUI, or even a NT Event logger among many other output targets, which gives the module great flexibility and power with little effect on the speed.

- *System.out.print() statements*. Used within some modules. These statements are used for logging, debugging etc, which cannot be disabled once the system goes live, or in production.

- *exception.printStackTrace() methods*. Same effect and problems as the previous method.

Log4j is currently the preferred way of debugging and logging of errors. Log4j gives the possibility of levels of logging with the ability of some of the levels to be disabled by a change of a configuration file when the system comes out of debugging.

2.1.1 Logging levels within Log4j

Here are the five logging levels for log4j with a description:

- FATAL. The FATAL level designates very severe error events that will presumably lead the application to abort

- ERROR. The ERROR level designates error events that might still allow the application to continue running, other runtime errors or unexpected conditions

- WARN. Use of deprecated APIs, poor use of API, Almost errors, other runtime situations that are undesirable or unexpected, but not necessarily "wrong".

- INFO. The INFO level designates informational messages that highlight the progress of the application at coarse-grained level. Interesting runtime events (startup/shutdown).

- DEBUG. The DEBUG Level designates fine-grained informational events that are most useful to debug an application.

The strict level of debugging can allow different targets of logging. In development, most targets can be files. In production, FATAL and ERROR messages can be targeted to a server port, this can allow the server to log the messages to a database, and/or email a member of staff about a problem or to LogFactory5 (section 3.1.3). WARN messages can be logged to files and INFO to files or a UNIX daemon, or even turned off with a DEBUG level.

2.2 Exception handling

Much of the code within the version of the Data Portal v3.x only catches `Exceptions` or `Throwables`. Some code does not even catch and log errors, and therefore errors from within the code are never recorded and ultimately never dealt with.

2.2.1 Dealing with Errors

Much of the Data Portal code does not fully deal with errors. Catching errors is not only about logging the error but also dealing with the problem.

The following code is taken from the current Data Portal:

```
DbAccess db = new DbAccess();
ResultSet rs =db query("SELECT * from userTable where userName='"+userId+"'");
if (rs.next()){
    return r  = db.buildXML(rs),
} else {
    // if user does not have an account in the facility then it returns the privileges of
    // a demo user which is stored in the database as Demo
    ResultSet rs1 = db.query("SELECT * from userTable where userName='demo'");
    rs1.next(),
   return r  = db buildXML(rs1),
}
```

Figure 1. Code taken from the current Data Portal.

This method creates a connection to a database, and then builds an XML document from the results. No exceptions are caught, dealt with or logged. Even if the code closed with a `db.shutdown()` method to close the connection and release the `ResultSet` and `Statement` objects for garbage collection any exception from a `NullPointerException` to a `SQLException` would skip the shutdown method and therefore keep a connection open (It is another topic altogether whether to check for `RuntimeExeptions` or to catch them. There is no CPU overhead for a try block if no exception is thrown but it makes bad coding to always check for `NullPointerExceptions`. See http://java.sun.com/docs/books/tutorial/essential/exceptions/runtime.html or http://www-106.ibm.com/developerworks/java/library/j-jtp05254.html for a discussion.).

See section 4.6.1 on profiling for further examples from this code.

A `try / catch` block is needed to catch exceptions. Maybe a `NullPointerException` means that the user is not in the database and therefore returns an empty document representing the outcome. A `SQLException` would represent an error with the database or the connection. This error would need to be logged as an ERROR and dealt with accordingly.

Since the two exceptions deal with closing the connection, it is then best to put this in the `finally` clause of the `try / catch` block. This is always executed, whether an exception is thrown or not. Therefore, the final code might look like figure 2.

NB: Database connections. When closing a `Connection`, not all database drivers release the `ResultSet` and `Statement` objects, so it is essential before they go out of scope these are closed and set to `null`.

```
 try{
     DbAccess db = new DbAccess(),
     ResultSet rs =db.query("SELECT * from userTable where userName='"+userId+"'");
     if (rs.next()){
         return r  = db.buildXML(rs);
     } else {
         // if user does not have an account in the facility then it returns the privileges of
         // a demo user which is stored in the database as Demo
         ResultSet rs1 = db query("SELECT * from userTable where userName='demo'");
         rs1.next();
        return r  = db buildXML(rs1),
     }
  catch(NullPointerException npe){
      log.warn("User "+userId+" not in privileges table"):
      return "";
  }
  catch(SQLException se){
      log.error("Uable to obtain "+userId+" privileges",se);
      return "";
  }
  finally{
      try{
          db.shutdown();
      }
      catch(Exception ignore){}
  }
}
```

Figure 2: Corrected code to deal with database connections.

2.2.1.1 Web service exception handling

Web services exception handling is inadequate. Other remote method invocation architectures, i.e. Java RMI ensures that the remote method API at compilation time is local to the client code, so that the Java Virtual Machine can check the error handling for the client code. Web services however, the exceptions thrown that are known at compile time are `AxisExceptions` (If Axis [2] is been used) and therefore it is unknown what the exception represents and how to deal with this problem.

When writing a client to a web service it is therefore impossible to know the difference with the errors unless it is explicitly known what exceptions are thrown and what they entail. E.g., the Web Interface checks for a `SessionTimedOutException` from the SessionManager because the Web Interface knows this through human knowledge. In a true web service environment, this would be impossible and the client would not know the difference between a `SessionTimedOutException` from a `ConnectionRefusedException` and therefore has to treat them the same.

3. SOLUTIONS TO SECTION 2

3.1 New architecture for Data Portal logging

All Data Portal modules to be re-engineered to standardise the logging levels. Each level used must conform to a level of seriousness that the error will effectively have on the Data Portal, and not the seriousness on the module itself. This only with the core Data Portal modules, i.e. not the XMLWrapper.

3.1.1 Usage and Examples

- FATAL. Only to be used when the error causes the Data Portal stop completely. Examples: UDDI server down, connection refused to Authentication Module, Session Manager database goes down. This is usually for the core modules of the Data Portal.
- ERROR. When the error causes something to fail but the Data Portal can still recover the fault. Example, Data Transfer service down, all other services still work.
- WARN. An error has occurred but this is an expected error. Examples, user's certificate expires, wrong password given, session timed out.
- INFO. Used with code but not in a catch statement. Simple information, i.e. the DN of user logging on.
- DEBUG. Used only to debug and test. E.g. checking null pointers, values etc, i.e. the DN of user logging on.

3.1.2 Log4j properties file

All Data Portal modules to use a global LOG4J logger properties file. This makes all the logging statements and targets commcn. Figure 3 shows a proposed LOG4J properties file:

```
log4j rootCategory=debug, console, server, LF5

# LF5 is the DailyRollingFileAppender that outputs to a rolling log
# file called dataportal log
log4j.appender.LF5.Threshold=debug
log4j appender LF5=org apache.log4j DailyRollingFileAppender
log4j appender.LF5.File=dataportal log
Users must also define a Pattern Layout for the log file  That is, which pieces of information you
want logged to the log file and in which order. Information on the various conversion specifiers (i.e.
d, p, t, l, m, n) are documented in the PatternLayout class of the Log4J API.

log4j.appender.LF5.layout=org.apache.log4j.PatternLayout
# Next line is editited to fit.
log4j appender LF5 layout ConversionPattern=[slf5s start]%d{DATE}[slf5s DATE]%n\
  %p[slf5s.PRIORITY]%n%x[slf5s NDC]%n%t[slf5s THREAD]%n%c[slf5s.CATEGORY]%n\
  %l[slf5s LOCATION]%n%m[slf5s.MESSAGE]%n%n

# console is appender to the catalina out/console
log4j appender console Threshold=debug
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender console layout=org apache log4j.PatternLayout
log4j appender console layout ConversionPattern=%p  %d{dd-MM HH:mm} %F:%L - %m%n
log4j.appender.console.ImmediateFlush=true

# server appender to a socket.  This emails about errors and fatala
log4j appender server Threshold=error
log4j.appender.server=org apache.log4j.net.SocketAppender
log4j appender.server.RemoteHost=localhost
log4j appender.server.Port=8888
```

Figure 3. Proposed Log4j properties file.

This file has three different logging outputs:

1. The first output appends to a `DailyRollingFileAppender`. This will append all log levels to a file per day in a certain format. This format is a standard for LOG4J that allows the LogFactor5 application to view the contents.(Section 3.1.3)

2. The next output logs all levels to the console appender. This is the catalina.out file generate from Tomcat, allowing for normal debugging and viewing.

3. The last is a socket appender. The socket append will send ERROR and FATAL messages to socket that is written by the Data Portal team. This socket can reside on any machine and could be free to do anything that it is programmed to do. E.g. log the messages to a database or file on the server, but since the `DailyRollingFileAppender` achieves this, it will be suggested that the server logs the errors and emails the errors with a unique id for the error to a list of administrators, who can go to the LogFactor5 GUI and locate the error.

3.1.3 LogFactor5 Application

NB: A commercial company gave LogFactor5 to the Apache Jakarta Log4j team. The log4j team liked the application so much that they wrote an open source version very similar to LogFactor5 called Chainsaw. This had all the functionality of LogFactor5 but with additions. At the time of writing this paper, Chainsaw version 2 [3] was not released, but all the information and ideas of LogFactor5 still apply to Chainsaw.

LogFactor5 is a Swing based GUI that leverages the power of log4j logging framework and provides developers with a sophisticated, feature-rich, logging interface for managing log messages.

Benefits:

- Quickly isolate problems in applications.
- Enable only the categories that you are interested in without affecting other messages.
- Filter out priority levels.
- Filter out records based on NDC (Nested Diagnostic Context, see below).
- Reduce the time required to locate specific messages.
- Read in and view log4j log files from either a file or a URL.
- Start the LogFactor5 GUI up independent of the main application.

Features:

- Real-time category and log level filtering.
- Read log files from a file.
- Read log files from a URL.
- Customizable Log Table view.
- Category level tree navigation.
- NDC record filtering.
- Full text searching on logged messages.
- Save configuration and filtering settings for later sessions.
- Configure the number of log records to be displayed.
- Customizable record colours.
- Configurable font face and size.
- Customizable Log Table column layout.

- Full support for all log4j levels.
- Full support for custom levels.
- Dynamic message counting.

The list of benefits and features is very rich and links in with the socket server/email solution. LogFactor5 allows a unique NDC (Nested Diagnostic Context) to be attached to every message sent. This message could contain the user's session id with a unique id time stamp. This message is then sent to the log files and to the Socket Server. The Socket Server would filter any Error/Fatal log4j messages and the unique NDC number sent via email to a list of administrators. The administrator could then filter the table of errors messages using LogFactor5 and locate the error that has just occurred with the unique id in the email.

Log4j NDC affects the NDC of the current thread only, i.e. they are managed on a per thread basis. Therefore, the rules for writing the NDC into the code are as follows

- Use `NDC.push(`_give unique id here as a string_ `);` This method, if no current NDC are on the thread, creates and puts it on a stack. This needs to be at the start of the code.

- Use `NDC.pop();` This removes it from the stack. There is no exception thrown for forgetting to pop the NDC out of the stack but you will end up with the previous id appended with the current one. This needs to be at the end of the code just before the NDC.remove(). This needs to be executed however the flow of the code goes.

- Use `NDC.remove()` at the end of the thread, i.e. at the end of the JSP or Servlet. This frees up the memory used for the NDC within the thread. This is needed with large heavy-duty applications, but with servlets, since another thread will execute the class the previous thread been executing, (i.e. a servlet) it is not necessary to use this method.

Sections of the code for the server can be seen in figure 6. This class creates a socket and listens for incoming connections. Once a connection is established with the Data Portal's Log4j server event logger, the application loops listening for events (ERROR and FATAL) to be sent to the server. Once the information about the logging event are extracted from the `LoggingEvent` class that Log4j sends, the information is sent to a list of administrators that the server is configured to email.

Log4j is configured with a properties file (in either XML or normal key/value pair). This means that configuration of Log4j is done on a ClassLoader basis. Since each web application within a servlet container is loaded via a separate ClassLoader each application has to configure itself to the global Log4j file. This can be achieved using the load-on-startup element (See figure 5) in the web.xml file of each application. This ensures that on startup of the servlet container, each application loads each servlet instance in a certain order and that the servlets that are to be loaded in order are loaded on startup and not when the servlet is executed via its first http request. Firstly, a servlet that the application wishes to load first must be slightly modified and its `init()` method overridden (See figure 4). This method is executed when the servlet is first loaded into memory and therefore the properties file is located and the Logj4 is configured for this application whenever the application is reloaded or installed.

```
//set static log for the class
Logger log = Logger.getLogger(this.getClass().getName());

//get context path

private String workingDir = null;

/** Initializes the servlet
 */
public void init(ServletConfig config) throws ServletException {
    //get and set the working dir
    ServletContext sc = config.getServletContext();
    workingDir = sc.getRealPath("");
    //set the log4j properties file for the whole web application
    PropertyConfigurator.configure(workingDir+File.separator+"WEB-INF"+File.separator+"log4j.properties");

}
```

Figure 4: Code obtaining the context path from a servlet to configure Log4j.

To load the servlet first upon the application being installed or reloaded this code snippet needs to be added in the web.xml file in each application. The load-on-startup element's value one means that it will be loaded up before any servlets with a higher number in their load-on-startup element.

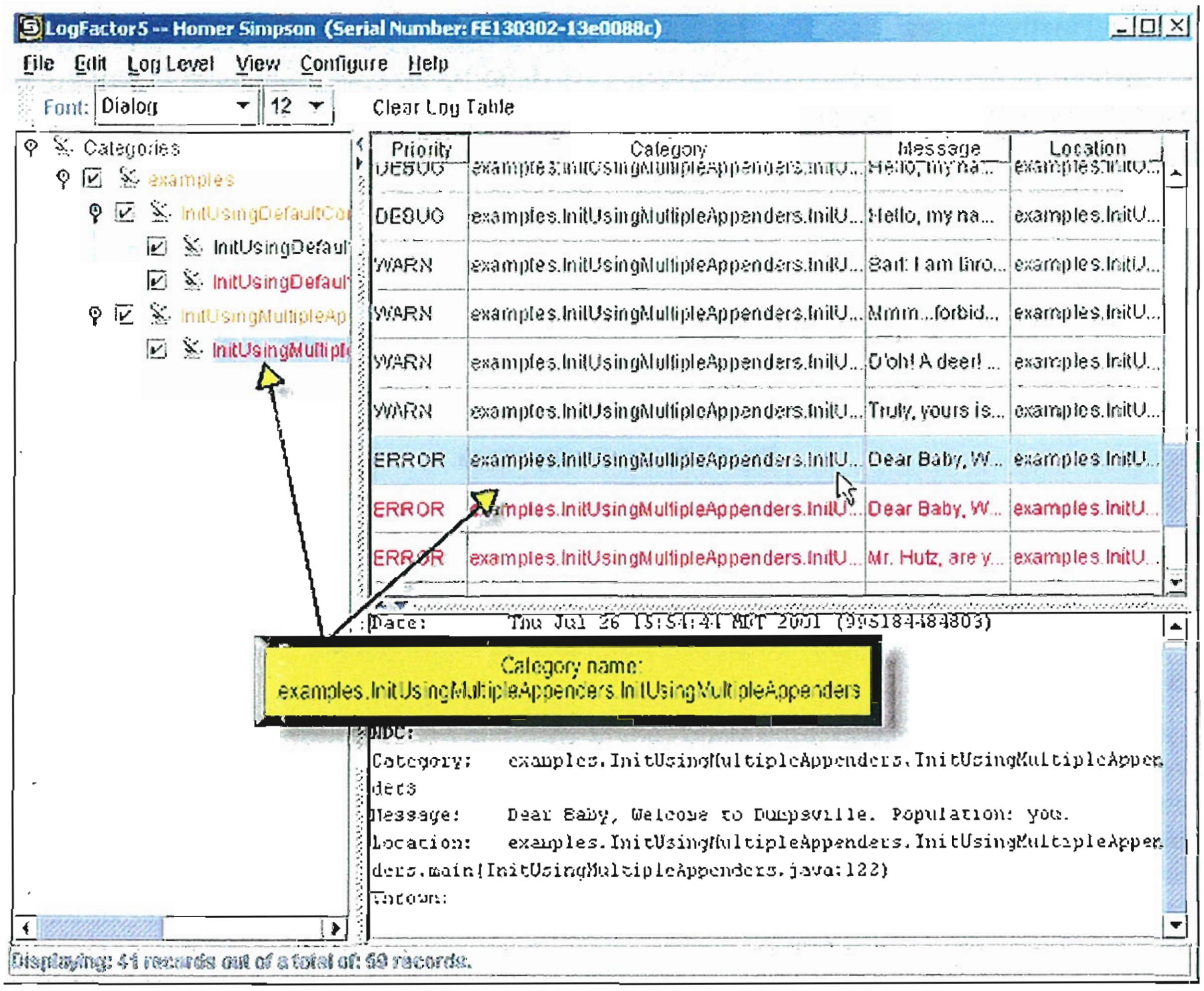
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee web-app_2_4.xsd" version="2.4">
    <display-name>Web Interface</display-name>
    <!-- <listener>
        <listener-class>SessionListener</listener-class>
    </listener> -->
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <display-name>LoginServlet Servlet</display-name>
        <servlet-class>

    uk.ac.dl.web.LoginServlet

   </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
```

Figure 5 : Section from a web.xml file configuring a servlet to load up first.

```
try{
    server = new ServerSocket(port,max_wait);
    log.info("Started server on port "+port);


    while(true){
        log.info("Awaiting a connection...  ");
        socket = server.accept();
        log.info("Connection accepted from "+socket.getInetAddress().getHostName());

        //output = new ObjectOutputStream(socket.getOutputStream());

        input = new ObjectInputStream(socket.getInputStream());

        LoggingEvent event;
        //do while loop will never gomout of since while is just set to true
        //only bottom catch can be got to if there is a problem above
        do{
            try{
                //get logging event
                event = (LoggingEvent)input.readObject();
                log.info("Logging Event incoming ... ");

                //get Exception message as a string[]
                ThrowableInformation throwable = (ThrowableInformation)event.
                getThrowableInformation();
                String[] exception = {"No Exception thrown with this logging event."};
                exception = (throwable != null) ? throwable.getThrowableStrRep() : exception;
        //get the level of the log4j event
        Level _level = event.getLevel();
        String level = _level.toString();

        props_conf.load(new FileInputStream(new File(prop_file_name)));
        String confLevel = props_conf.getProperty("level", "FATAL");
        boolean email = false;
        //only fatal and error message can get send to emails.
        if(confLevel.equalsIgnoreCase("FATAL")) email =
        (level.equalsIgnoreCase("FATAL"))? true : false;
        else if(confLevel.equalsIgnoreCase("ERROR")){
            email = (level.equalsIgnoreCase("ERROR") ||level.equalsIgnoreCase("FATAL")
            true : false;
        }

        log.debug("Logging level is "+level+" . Sending email(s): "+email);
        //get the time of the event
        long time = event.getStartTime();
        Date date = new Date(time);

        //get the unique NDC
        String ndc  = event.getNDC();

        //get the message
        String message = event.getRenderedMessage();

        if(email) sendEmail(ndc,date,level,exception, message);

    }
    catch(NoClassDefFoundError ncdfe){
        log.fatal("Unable to run server without Class Def. Shutting down",ncdfe);
        System exit(0);
    }
    catch(ClassNotFoundException cnfe){
        log fatal("Unable to run server without Class Def  Shutting down",cnfe);
        System.exit(0);
    }
    catch(SocketException se){
        log.error(se);
        //
        try{
            log.warn("Waiting for 2 minutes.   then will try to reconnect...");
            input.close();
            socket.close();
            Thread.sleep(120000);
        }
        catch(Exception ignore){}
        break;
    }
```

Figure 6: Code from the Log4j server to email Data Portal administrators of problems.

LogFactor5 has other benefits. It displays loggers in a hierarchical tree-like fashion. Loggers are identified with dot-separated names similar to Java package and class names. The name components are mapped to the levels in the hierarchy in LogFactory5. For example, the logger name `org.apache.applications.log4j.InitUsing` Log4JProperties is displayed in figure 7. This allows the view of all the error levels from with in the Data Portal, filters view certain levels, i.e. FATAL and ERROR. It allows a view of all the errors from each module in turn, because of the package name. This tree like structure allows errors to be viewed, not only from the Data Portal to the module level, but further to the class level. It would be possible to view all the FATAL errors from within one class within a module from any time scale, allowing monitoring of persistent failure points etc.


LogFactor5 -- Homer Simpson (Serial Number: FE130302-13e0088c)
File Edit Log Level View Configure Help
Font: Dialog 12
Clear Log Table
Categories
examples
InitUsingDefaultCo
InitUsingDefaul
InitUsingDefaul
InitUsingMultipleAp
InitUsingMultipl
Category name:
examples.InitUsingMultipleAppenders.InitUsingMultipleAppenders

Priority	Category	Message	Location
DEBUG	examples.InitUsingMultipleAppenders.InitU...	Hello, my na...	examples.InitU...
DEBUG	examples.InitUsingMultipleAppenders.InitU...	Hello, my na...	examples.InitU...
WARN	examples.InitUsingMultipleAppenders.InitU...	Bart: I am litro...	examples.InitU...
WARN	examples.InitUsingMultipleAppenders.InitU...	Mmm...forbid...	examples.InitU...
WARN	examples.InitUsingMultipleAppenders.InitU...	D'oh! A deer! ...	examples.InitU...
WARN	examples.InitUsingMultipleAppenders.InitU...	Truly, yours is...	examples.InitU...
ERROR	examples.InitUsingMultipleAppenders.InitU...	Dear Baby, W...	examples.InitU...
ERROR	examples.InitUsingMultipleAppenders.InitU...	Dear Baby, W...	examples.InitU...
ERROR	examples.InitUsingMultipleAppenders.InitU...	Mr. Hutz, are y...	examples.InitU...

```
Date:        Thu Jul 26 15:54:41 MDT 2001 (995184484803)
NDC:
Category:    examples.InitUsingMultipleAppenders.InitUsingMultipleAppen
ders
Message:     Dear Baby, Welcome to Dumpsville. Population: you.
Location:    examples.InitUsingMultipleAppenders.InitUsingMultipleAppen
ders.main(InitUsingMultipleAppenders.java:122)
Thrown:
```

Displaying: 41 records out of a total of: 59 records.

Figure 7: View of the LogFactor5 GUI showing Log4j messages.

3.2 Exception Handling

3.2.1 Techniques

3.2.1.1 Private methods

Private methods should not log errors. This would lead to a single error being logged twice or more. A private method should, if needed, catch the exception to clean up and deal with the error. E.g. database and file connections. The code in the class that calls this private method should log the error.

With web services, the main web service method should log the error in most cases, if the error is not recoverable. The exception should be thrown upwards in the method call trace and not logged by the private methods so the same error is not logged more than twice.

The exceptions thrown should also be more helpful. A `NullPointerException` from a UDDI lookup for a URL should be checked for it's value. A `NullPointerException` then should be checked and an exception thrown with a message, i.e. new `NullPointerException` ("Url for "+facilityName +" cannot be "+facilityUrl) would help in the debugging instead of a `NullPointerException()` with no message.

3.2.1.2 Try-Finally clauses.

When catching multiple exceptions the code that needs to be executed whatever execution flow can use the `try-finally` clause.

```
try{
    //block of code with multiple exit points
}
finally{
    //Block of code that is always executed when the try block is exited
    //no matter how the try block is exited, even if a exception is thrown
}
```

Figure 8: Try-finally block.

This is particularly useful when a database connection has been created within the `try` block. If the code needed to be caught by multiple catches then instead of shutting down the connection in the `try` and all of the `catch` blocks it is desirable to place it into the `finally` clause.

3.2.1.3 What to catch

`Exception` or `Throwable` classes are mostly caught within the Data Portal application. `Throwable` classes should never be caught because this contains `Error` classes which are thrown when a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an `Error`. Catching `Exceptions` is not a problem if the code checks for null pointers (i.e. most `RuntimeExceptions` associated with the code) and the error handling is to be the same for all errors. However, different errors sometimes need to be treated differently, connections closed, different responses, return values etc.

For example, the Shopping Cart web service retrieves the users contents with a session id that has been given. The web services checks exceptions for a `SessionTimedOutException` thrown from the SessionManager. This is a checked exception that needs to be treated differently from a `SQLException`. A checked exception is a exception that a `try-catch` must either be nested around the call to the method that throws the exception *or* the method must explicitly indicate with `throws` that it can generate this exception.

3.2.1.4 Dealing with errors

Refer to section 2.2.1

4. PRODUCTION LEVEL ENHANCEMENTS FOR DEPLOYMENT

4.1 Monitoring of users

Monitoring of users by direct logging of all web service invocations within the Data Portal. Since web services are slower and more memory intensive than normal J2EE applications a better solution would be to use a API based interface, like Quartz [4]. Quartz provides multi-threading capabilities and other benefits, see section 4.5.2.

This API would create a new thread to log the invocation by passing the session id (sid). The sid would be sent at the start of every web service call with the time, method called etc to a database to be logged against a user within the database. A thread is more desirable than static method call because of better exception handling and possible blocking / timeout issues with the database connection, and hence would not affect the Data Portal's logging or performance. Since this logging would create large volumes of connection calls, a connection pool is advantageous, which the servlet container like Tomcat would monitor.

Possibility of creating a GUI to view sessions and services used by the users. This would allow the administrators of the application to view/create usage charts and to see which users are logging on to the Data Portal and which web services they are using.

The architecture of the database schema needs to be investigated.

4.2 Monitoring of services

Introduction of a monitoring service to the Data Portal. There is a monitoring tool called Big Brother [5]. It is free under academic use. The tool uses a client and server approach. Another similar tool is Nagios [6], it is an open source host, service and network monitoring program.

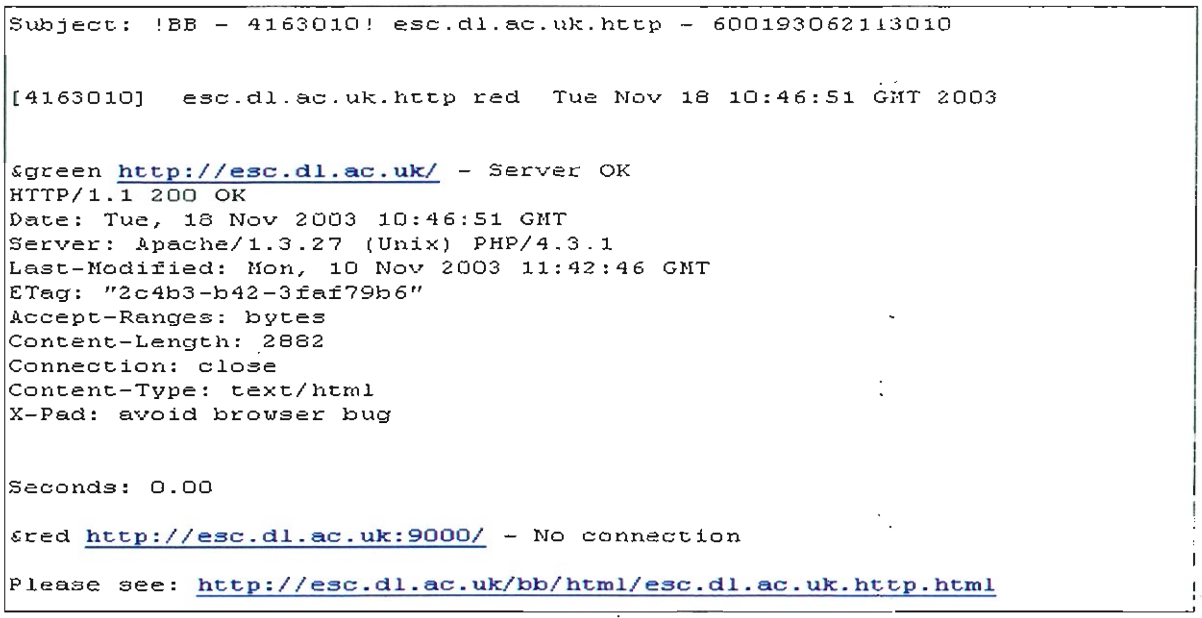
These monitoring tools are designed to inform the administrator of network problems before clients, end-users or managers do. The monitoring daemon intermittent checks on hosts and services you specify using external "plugins" which return status information. When problems are encountered, the daemon can send notifications out to administrative contacts in a variety of different ways (email, instant message, SMS, etc.). Current status information, historical logs, and reports can all be accessed via a web browser. This will allow administrators to locate and fix problems before users to the Data Portal encounter them.

4.2.1 Big Brother

The server is installed on a Linux machine and interfaces to an Apache web server, which displays the status information of machine. The client is run on each computer that needs to be monitored. The server every 5 minutes contacts the clients through a single port 1948 and requests information about the machine. This can be to monitor CPU and memory usage, http connections, DNS, FTP, POP3, databases etc. Once a service falls under a certain level, (disk over 95%, http connection lost) an email is sent to a list of administrators.

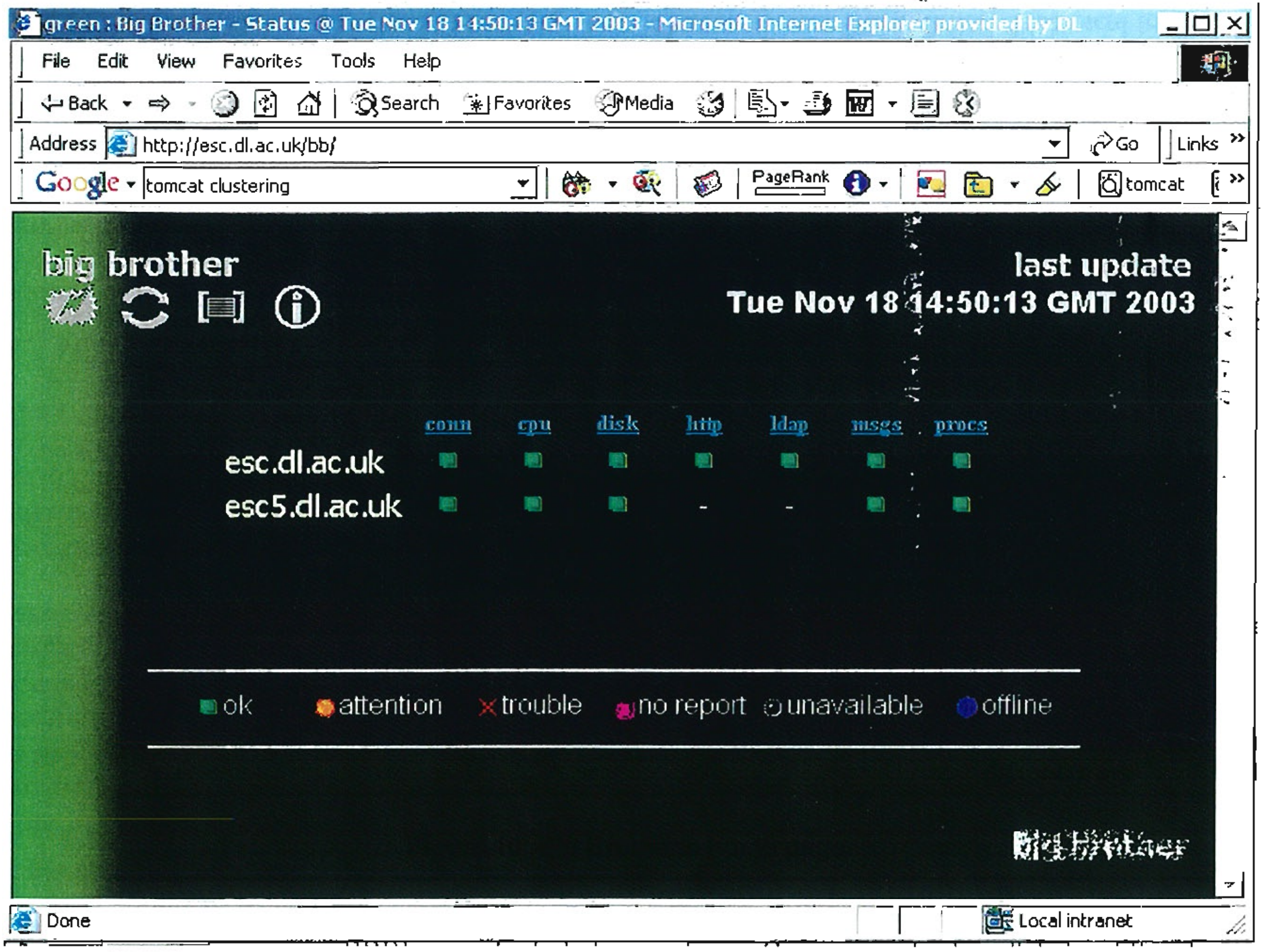
One of these clients could sit on our Data Portal node and monitor the tomcat server, database and UDDI server for lost of connections. Groups can be collated and therefore a Data Portal group could be established to monitor all the Data Portal services, which reside on different machines.

Figure 10 below shows a screen shot of the Big Brother web front end that is monitoring esc and esc5 on our sub net. The server contacts the clients on the two machines, which executes a number of shell scripts on each machine. Standard scripts are available for http, databases, memory, ldap etc. Green means that the service is ok. If a service goes down, administrators are contacted via an email in the format in figure 9.



```
Subject: !BB - 4163010! esc.dl.ac.uk.http - 600193062113010


[4163010]  esc.dl.ac.uk.http red  Tue Nov 18 10:46:51 GMT 2003


&green http://esc.dl.ac.uk/ - Server OK
HTTP/1.1 200 OK
Date: Tue, 18 Nov 2003 10:46:51 GMT
Server: Apache/1.3.27 (Unix) PHP/4.3.1
Last-Modified: Mon, 10 Nov 2003 11:42:46 GMT
ETag: "2c4b3-b42-3faf79b6"
Accept-Ranges: bytes
Content-Length: 2882
Connection: close
Content-Type: text/html
X-Pad: avoid browser bug


Seconds: 0.00

&red http://esc.dl.ac.uk:9000/ - No connection

Please see: http://esc.dl.ac.uk/bb/html/esc.dl.ac.uk.http.html
```

Figure 9: Contents of a Big Brother alert email.



green : Big Brother - Status @ Tue Nov 18 14:50:13 GMT 2003 - Microsoft Internet Explorer provided by DL

Address http://esc.dl.ac.uk/bb/

big brother

last update
Tue Nov 18 14:50:13 GMT 2003

	conn	cpu	disk	http	ldap	msgs	procs
esc.dl.ac.uk	🟩	🟩	🟩	🟩	🟩	🟩	🟩
esc5.dl.ac.uk	🟩	🟩	🟩	-	-	🟩	🟩

🟩 ok ● attention ✗ trouble ● no report ○ unavailable ● offline

Figure 10: Big Brother's home page.

4.3 Clustering of Tomcat instances

Tomcat 5, the new release of Tomcat implements Servlet 2.4 and JavaServer Pages 2.0 specifications from Sun. One of the newest enhancements is session replication. This allows clusters of Tomcat instances to replicate their session data. Therefore, if a tomcat instance fails the other tomcat instance can take over the session request to process it as normal. This clustering of tomcat servlet containers allows the Data Portal service to be more reliable and therefore more accessible to the users.

4.4 Load Balancing of Tomcat instances

Tomcat can be used as a back end server behind a single Apache Web Server. This web server acts as a load balancer for the multiple instances of tomcat (a cluster using session replication). If an instance fails, the session information is sent to another instance within the cluster. Because they share the session information with more tomcat instances, the user would then carry on as normal with an improved response time.

The load balancer checks the tomcats periodically to see which of the instances are available for a job (receive a page request), and checks how many of the instance is has received from the load balancer. If an instance goes down the load balancer (lb) will delegate the request to the rest of the cluster until the tomcat is up again and ready to receive requests. At this point it will enter the cluster again, show itself to the lb, and receive requests.

With clustering, it is possible to work on upgrading the Data Portal code or bug fixes without stopping the service. One tomcat is taken out of the cluster and upgraded. Then after that the other (depends if there are 2 tomcats) is taken down and the first re-established to the clusters. This has been demonstrated with the development Data Portal blade server. It was simple to configure and works well. This also removes the port number when logging on to the Data Portal, as they would go through the normal port 80 in which browsers do automatically. Figure 11 shows a simple worker2.properties file for Apache for load balancing with 50/50 without session replication.

```
[logger.apache2]
level=DEBUG

[shm]
file=c:\Program Files\Apache Group\Apache2\logs\jk2.shm

# Example socket channel, override port and host.
[channel.socket:gjd37vig.dl.ac.uk:8009]
tomcatId=tomcat1
group=lb_1

#define the worker
[ajp13:gjd37vig.dl.ac.uk:8009]
channel=channel.socket:gjd37vig.dl.ac.uk:8009
lb_factor=1

# Example socket channel, override port and host.
[channel.socket:gjd37vig.dl.ac.uk:9009]
tomcatId=tomcat2
group=lb_1

#define the worker
[ajp13:gjd37vig.dl.ac.uk:9009]
channel=channel.socket:gjd37vig.dl.ac.uk:9009
lb_factor=1

[lb:lb_1]
#info=Default load balancer
#debug=0

worker=ajp13:gjd37vig.dl.ac.uk:8009
worker=ajp13:gjd37vig.dl.ac.uk:9009

[uri:/*]
group=lb:lb_1
```

Figure 11: A workers2.propteries file, configuration for apache load balancing onto tomcat.

The tomcatId corresponds to the jvmRoute attribute of the Engine element in the tomcats server.xml file. (NB: it is also possible to take away the tomcatId from is file (worker2.properties) and apache will just alternate and not load balance 50/50).

4.5 House keeping

4.5.1 Cron jobs

4.5.1.1 Databases

Modules like the session manager create large amounts of data within their databases. Most of the data is removed when a users logs off the Data Portal. However, when a user does not log off the Data Portal and their session times out the data within the database is retained. Depending on the amount of users and the frequency of their usage, this information could become very large.

A cron job (or something similar) to periodically check the databases should clear the database of any unnecessary data.

4.5.1.2 File Systems

Similar to section 4.5.1.1. File systems contained user workspace to be cleared from the entire Data Portal. E.g. results.xml files etc.

4.5.1.3 Log files

Tomcat log files in a production environment can get very large. Jobs should be set up to weekly compress the catalina.out file and access file and replace them.

4.5.1.4 Weekly/Monthly/Daily records

Investigate into the possibility of producing weekly, monthly reports of the usage, down time etc of the Data Portal through the session manager info and the access log files.

I.e. the weekly log file should be created and compressed for storage to save on space on the file system log directory.

These tasks could be created and executed with Quartz. See section 4.5.2.

4.5.2 Quartz

Quartz is a job scheduling system that can be integrated with, or used along side virtually any J2EE or J2SE application. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components or EJBs.

Sample uses of job scheduling with Quartz:

- Driving Workflow: As a new order is initially placed, schedule a Job to fire in exactly 2 hours, that will check the status of that order, and trigger a warning notification if an order confirmation message has not yet been received for the order, as well as changing the order's status to 'awaiting intervention'.

- System Maintenance: Schedule a job to dump the contents of a database into an XML file every business day (all weekdays except holidays) at 11:30 PM.

This can be used instead of a cron job for more complex tasks that do not depend on the Data Portal, i.e. Tasks to clean up the File System from timed out sessions, explained before, whenever the Data Portal has zero sessions. Tomcat can listen for sessions and counts them when sessions are created and destroyed.

Quartz is java based and would be easily implemented in the Data Portal. The coding is every simple, the following two examples give an indication of the simplicity.

```
SchedulerFactory schedFact = new org quartz impl.StdSchedulerFactory().
Scheduler sched = schedFact getScheduler().
sched start();

JobDetail jobDetail = new JobDetail("myJob", sched.DEFAULT_GROUP,  JobClass.class);
JobDetail jobDetail2 = new JobDetail("myJob2", sched DEFAULT_GROUP,  Another.class).

long endTime = System.currentTimeMillis() + 6000000L.

SimpleTrigger strigger = new SimpleTrigger("mytrigger", sched.DEFAULT_GROUP, new Date(),new Date(endTime).
SimpleTrigger REPEAT_INDEFINITELY, 600L*100L).
CronTrigger trigger = new CronTrigger("myTrigger", "myGroup","myJob2",| "myGroup","0 * 12 ? * WED*");

sched.scheduleJob(jobDetail, trigger).
sched.scheduleJob(jobDetail2, trigger2).
```

Figure 12: Simple Quartz code to set up two jobs.

The first job schedule is a simple trigger. The trigger fires on the exactly specified intervals. Here it fires every hour from now for 6 hours. The second one is more like a cron job schedule, a job-firing schedule that recurs based on calendar-like notions. Here it fires off every Wednesday at 12 pm. The scheduler executes a class that is defined in the job detail. This class has to implement Job and therefore must have a method `execute`, which is the method that is executed.

4.6 Profiling

Java JVM come with a option of profiling the application that the JVM is running with a argument -Xint -Xrunjprofiler:port=8849 or -Xrunhprof etc. This has the effect of printing out information about the heap, classes, and instances etc of the application. For example

```
Object allocated from:

  java.util.TimeZoneData.<clinit>(()V) :
        TimeZone.java line 1222
  java.util.TimeZone.getTimeZone((Ljava/lang/String;)
         Ljava/util/TimeZone;) :
         TimeZone.java line (compiled method)
  java.util.TimeZone.getDefault(
         ()Ljava/util/TimeZone;) :
         TimeZone.java line (compiled method)
  java.text.SimpleDateFormat.initialize(
         (Ljava/util/Locale;)V) :
         SimpleDateFormat.java line (compiled method)
```

Or

```
CPU TIME (ms) BEGIN (total = 11080)
                       Mon Mar  29 16:40:59 2004
rank   self    accum   count  trace   method
 1   13.81%  13.81%       1   437    sun/
   awt/X11GraphicsEnvironment.initDisplay
 2    2.35%  16.16%       4   456    java/
   lang/ClassLoader$NativeLibrary.load
 3    0.99%  17.15%      46   401    java/
   lang/ClassLoader.findBootstrapClass
```

Figure 13: Sample profile output from a JVM.

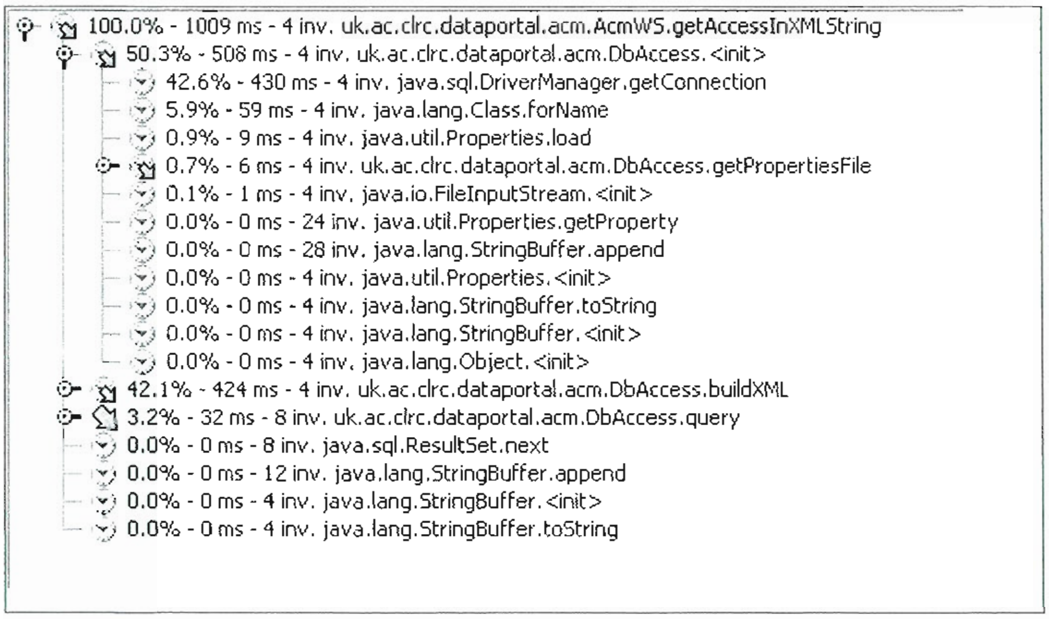
This information is very difficult to understand. Applications can read these outputs and display them in a GUI, i.e. LogFactor5 reads the output from Tomcat.

JProfiler from ej-technologies' [7] is one good product on the market which is cheap ($199 for one license). It helps find performance bottlenecks, pin down memory leaks and resolve threading issues, which once rectified makes the application faster, more reliable and robust.

The code in section 2.2.1 was run through JProfiler.

4.6.1 CPU usage

The following shows the CPU usage of the ACM module (a Data Portal module) calling the method to get the user's access rights as a string.



```
100.0% - 1009 ms - 4 inv. uk.ac.clrc.dataportal.acm.AcmWS.getAccessInXMLString
  50.3% - 508 ms - 4 inv. uk.ac.clrc.dataportal.acm.DbAccess.<init>
    42.6% - 430 ms - 4 inv. java.sql.DriverManager.getConnection
    5.9% - 59 ms - 4 inv. java.lang.Class.forName
    0.9% - 9 ms - 4 inv. java.util.Properties.load
    0.7% - 6 ms - 4 inv. uk.ac.clrc.dataportal.acm.DbAccess.getPropertiesFile
    0.1% - 1 ms - 4 inv. java.io.FileInputStream.<init>
    0.0% - 0 ms - 24 inv. java.util.Properties.getProperty
    0.0% - 0 ms - 28 inv. java.lang.StringBuffer.append
    0.0% - 0 ms - 4 inv. java.util.Properties.<init>
    0.0% - 0 ms - 4 inv. java.lang.StringBuffer.toString
    0.0% - 0 ms - 4 inv. java.lang.StringBuffer.<init>
    0.0% - 0 ms - 4 inv. java.lang.Object.<init>
  42.1% - 424 ms - 4 inv. uk.ac.clrc.dataportal.acm.DbAccess.buildXML
  3.2% - 32 ms - 8 inv. uk.ac.clrc.dataportal.acm.DbAccess.query
  0.0% - 0 ms - 8 inv. java.sql.ResultSet.next
  0.0% - 0 ms - 12 inv. java.lang.StringBuffer.append
  0.0% - 0 ms - 4 inv. java.lang.StringBuffer.<init>
  0.0% - 0 ms - 4 inv. java.lang.StringBuffer.toString
```

Figure 14: JProfiler examples of CPU usage on a Data Portal module.

The results show that getting the connection and results set took 50% and building the XML from the results set took 42.1% of the time. Drilling down into the classes and methods it is possible to identify bottlenecks and performance issues. For example, the creation of the `DbAccess` class (init method) can be broken down, and majority of the CPU time is taken up by the `getConnection` method to the database. Since this is expected with database connections, there is little code enhancements that would speed it up. Nevertheless, if this time is too much, it is possible to improve the performance, e.g. use connection pooling etc.

4.6.2 Memory usage

With JProfiler it is possible to view the instances of classes, sizes etc. Again, the code profiled and a heap snap shot and the information inspected. The diagram in figure 15 identifies an instance within the java heap, which is of the class `DbAccess`. The arrows to the right identify the outgoing references that the instance holds directly. Normally, each instance should have a root to the Garbage Collection (GC). This root identifies to the JVM which instances are ready for garbage collection, these have no strong or weak references to them.

JProfiler can then look into the class instance and find out the size of the `DbAccess` class and all the sizes of the information referenced by the `DbAccess`. The total size of the `DbAccess` and all the sub information is 150kb. This information can be used to locate memory leaks within a java application. Since this instance of `DbAccess` has no root to the JVM that it is running under and no other references from `DbAccess` has any roots to the JVM means that this is ready for GC. The GC runs normally under a lower thread priority and therefore a class instance that is ready for GC has to wait until memory is needed or the application has no other higher priority threads to execute.

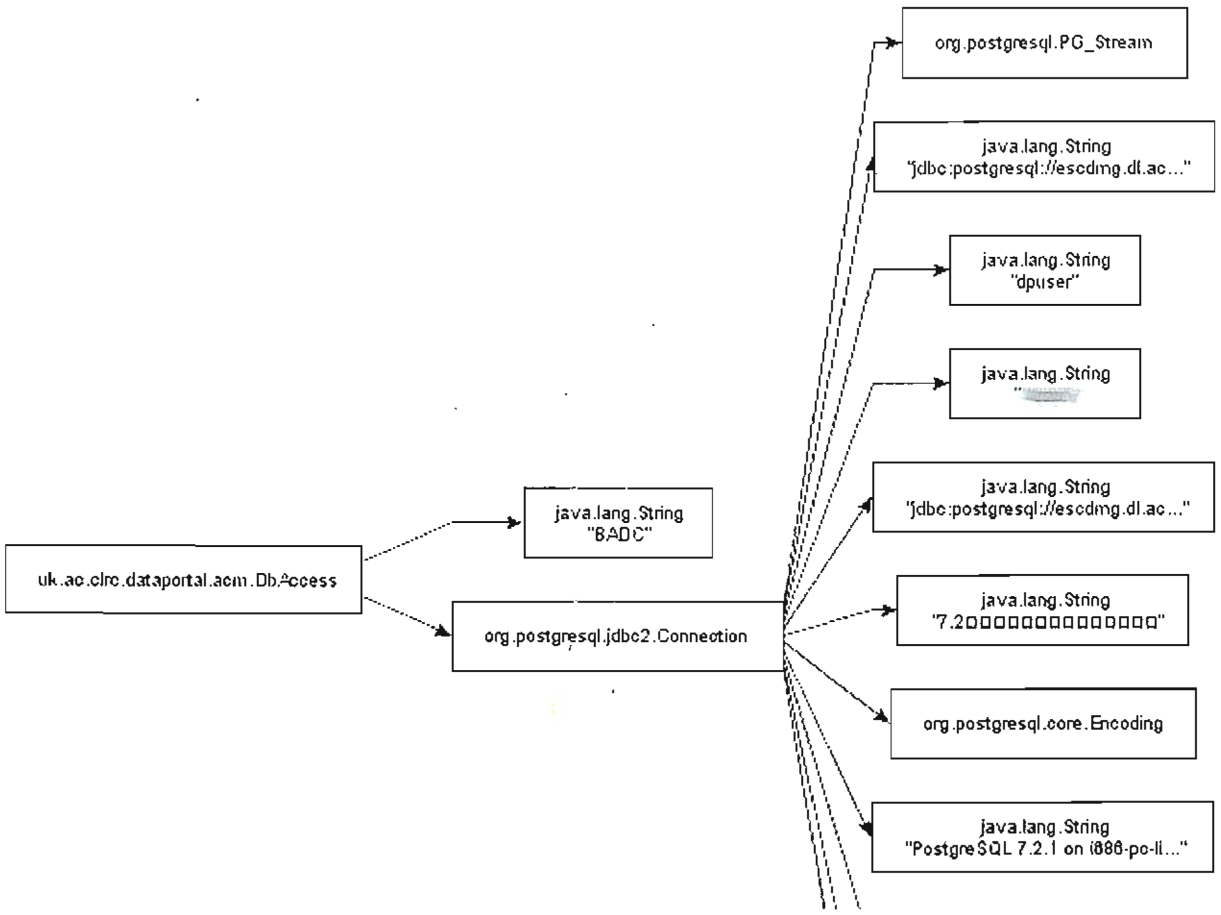

org.postgresql.PG_Stream
java.lang.String "jdbc:postgresql://escdmg.dl.ac..."
java.lang.String "dpuser"
java.lang.String
java.lang.String "jdbc:postgresql://escdmg.dl.ac..."
java.lang.String "BADC"
uk.ac.clrc.dataportal.acm.DbAccess
org.postgresql.jdbc2.Connection
java.lang.String "7.2□□□□□□□□□□□□□□"
org.postgresql.core.Encoding
java.lang.String "PostgreSQL 7.2.1 on i686-pc-li..."

Figure 15: Memory heap taken from JProfiler.

Below there is an example of an obvious java memory leak, but highlights the steps to track down the leak:

Servlet A instantiates a static `Hashtable` in it's constructor. Servlet A then proceeds to add classes to that `Hashtable` without removing anything at the end of the request. Next request comes in and because the servlet is pooled between requests, the static `Hashtable` is the same one from the first request and still has all the content stored in it. Servlet A adds even more classes while servicing this request, but still has not removed any of the old classes by the end of processing. This servlet repeats this until the JVM is full and there is no more memory for anything else.

In this example, on profiling profiled this application of a heap snap shot, it would reveal that Servlet A has a very large memory heap, inspecting more, the static `HashTable` would be the culprit. Next, apply to look into the class and the `HashTable` to find that the `HashTable` has a root to the JVM that runs the Servlet container and therefore will never be GC until the container is shut down.

Here are some articles explaining java memory leaks.

http://www-106.ibm.com/developerworks/java/library/j-leaks/

http://www.adtmag.com/java/articleold.asp?id=165

http://sys-con.com/story/?storyid=44716&DE=1

4.7 Stress testing

Once profiling of the code has been optimised and investigated it is now possible to start to load test functional behaviour and measure performance (stress testing). This can be done using a performance testing application. A good free one is JMeter [8] from Apache, it is a 100% Java desktop application used to test performance both on static and dynamic resources (files, Servlets, Perl scripts, Java Objects, Data Bases and Queries, FTP Servers and more). It can be used to simulate a heavy load on a server, network or object to test its strength or to analyze overall performance under different load conditions. JMeter can generate graphical analysis of performances or to test the server/script/object behaviour under heavy concurrent load.

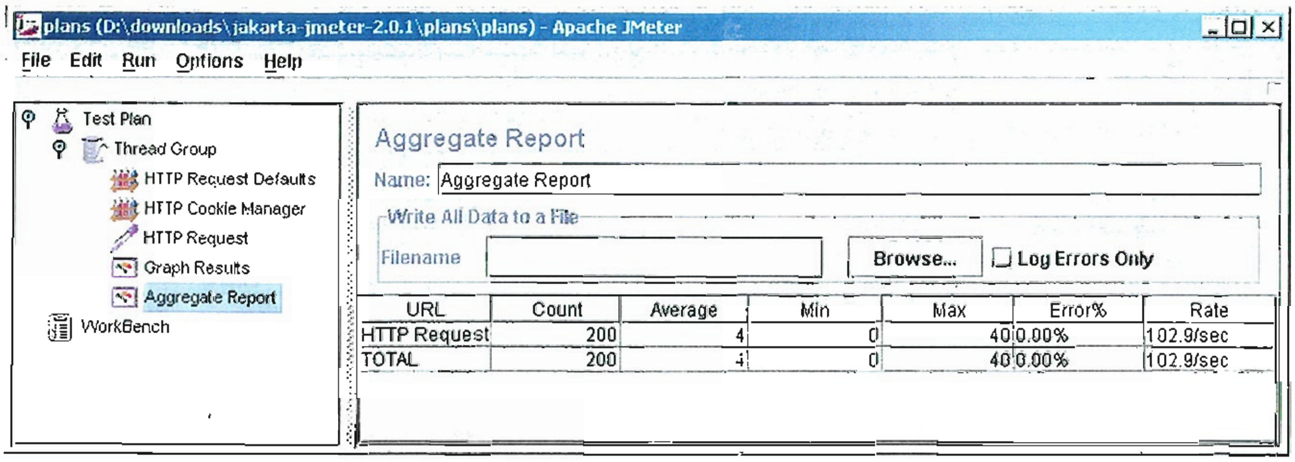
Apache JMeter features include:

- Can load and performance test HTTP and FTP servers as well as arbitrary database queries (via JDBC)
- Complete portability and 100% Java purity .
- Full Swing and lightweight component support (precompiled JAR uses packages javax.swing.*).
- Full multithreading framework allows concurrent sampling by many threads and simultaneous sampling of different functions by seperate thread groups.
- Careful GUI design allows faster operation and more precise timings.
- Caching and offline analysis/replaying of test results.
- Highly Extensible:
    - Pluggable Samplers allow unlimited testing capabilities.
    - Several load statistics may be choosen with pluggable timers.
    - Data analysis and visualization plugins allow great extendibility as well as personalization.
    - Functions (which include JavaScript) can be used to provide dynamic input to a test
    - Scriptable Samplers (BeanShell is supported in version 1.9.2 and above)

4.7.1 Testing Data Portal Web Interface

A simple test was performed on the Data Portal, 200 users concurrently opened the front page to the Data Portal and the timing and performance parameters were measured. To test the Data Portal properly, there needs to be a comprehensive Test Plan created. Since the Data Portal needs to be logged on to test the pages like the Shopping Cart, there needs to be an initial logging on to the Portal before the test can be initiated. This can be achieved using JMeter.

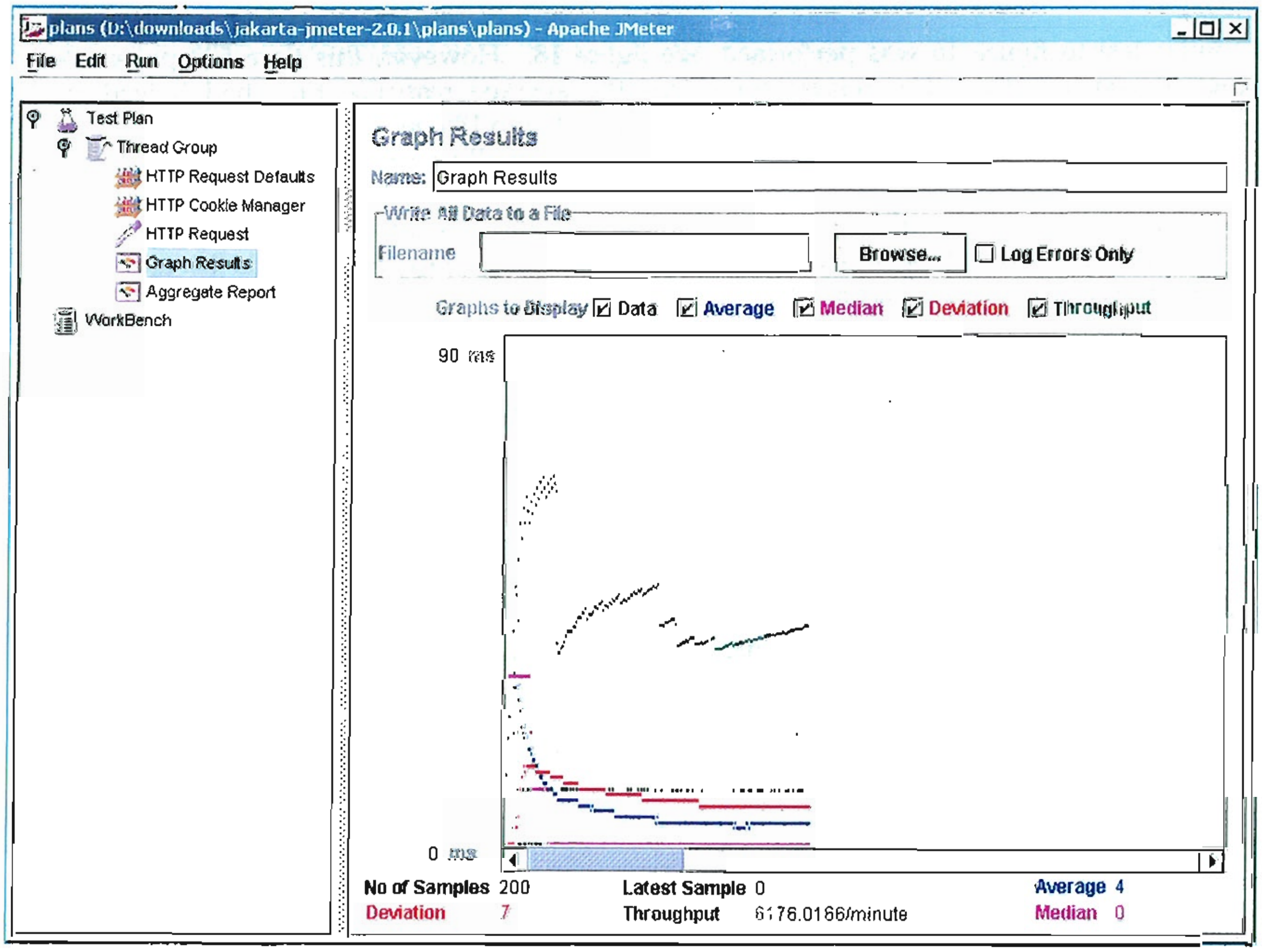
Figure 16 shows an Aggregate Report from 200 concurrent HTTP requests for the Login page of the Data Portal. The average time for the request was 4 milliseconds, with no errors and the throughput rate of over 100 requests per second.



Name: Aggregate Report

URL	Count	Average	Min	Max	Error%	Rate
HTTP Request	200	4	0	40	0.00%	102.9/sec
TOTAL	200	4	0	40	0.00%	102.9/sec

Figure 16: JMeter Aggregate Report from a simple Data Portal test.

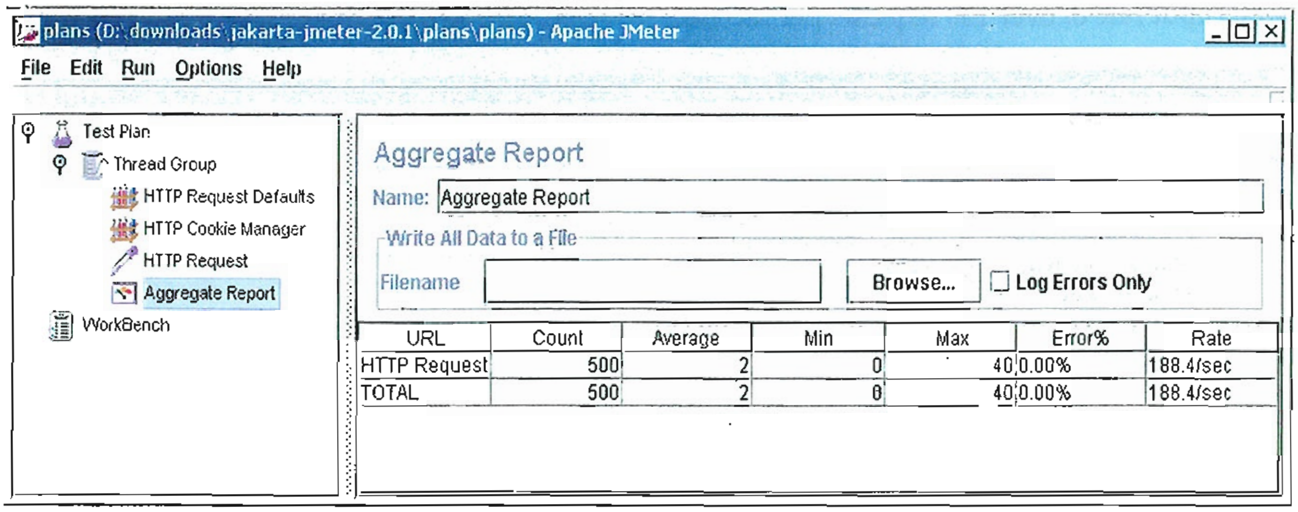


Name: Graph Results

No of Samples 200
Deviation 7
Latest Sample 0
Throughput 6176.0166/minute
Average 4
Median 0

Figure 17: JMeter Graph Result from a simple Data Portal test

Figure 17 shows the graph of the same test and simple results as before. Again, the average was 4 milliseconds and the same throughput rate per minute, instead of per second. Since Java is a multi-threaded programming language, it is not surprising to see that the average response time decreases as the amount of request increases.



Name: Aggregate Report

URL	Count	Average	Min	Max	Error%	Rate
HTTP Request	500	2	0	40	0.00%	188.4/sec
TOTAL	500	2	0	40	0.00%	188.4/sec

Figure 18: Results show that with more requests the response time decreases (See Figure 16)

A similar test to figure 16 was performed, see figure 18. However, this time 500 pages were requested concurrently. The results show that the average response time had halved to 2 milliseconds and the throughput rate had gone up from 103 to 188 per second. This is typical for a multi-threaded java application. These applications are normally extremely scaleable compared to CGI or other web based applications. The overhead of spawning a thread onto a servlet class is insignificant to the process of originally creating and loading the servlet into memory. Hence, once the class is loaded into memory, to a certain point, adding concurrent threads (i.e. users) does not effect the response time or error rate.

4.7.2 Testing Web Services

JMeter has the capability to test Web Services as well as the java objects, HTTP requests, FTP Servers etc. It is as simple as testing a HTTP response as theoretically a Web Service invocation can be a HTTP request (it can also be a SMTP request).

Again, after unit testing the Web Service functionality for correct results, and then profiling the code for bottlenecks, pin down memory leaks and resolve threading issues to increase the speed of the invocation, it would be advisable to load test functional behaviour and measure performance of the web service. Using JMeter or another profiling application, to simulate a heavy load on the server to test its strength or to analyze overall performance under different load types allows a view of the web services speed under different load types and its scalability. The speed represents the time it is able to respond to a request and scalability measures how many simultaneous requests a web service application can process before its speed deteriorates to an unacceptable level.

Profiling and Stress Testing might seem a little blurred, and in some ways, this is true. Some developers would skip profiling and review their code without granular performance data, looking for obvious inefficiencies. Profiling however, helps to save developers time by showing them exactly where the greatest performance gains can be realised, and changing their code before they hit performance and scalability problems future down the development line, when they have to try to spot the inefficiencies within the code and then redevelop it.

4.8 Portals and other Frameworks

Research the possibility of other portals and frameworks. For example OpenCMS , Sakai Portal.

5. CODE ENHANCEMENTS OF CODE FOR DEPLOYMENT

5.1 Unit Testing

Each web service and helper classes within each module of the Data Portal should be unit tested, preferably using JUnit [9]. Each module's package hierarchy should contain a folder unittest. This should contain a class file for each class that is being tested, with different tests depending on which methods are being tested.

Achieving this is simple, by using some code within the web service. The code within a method would return the directory from where to find the properties file. See figure 19.

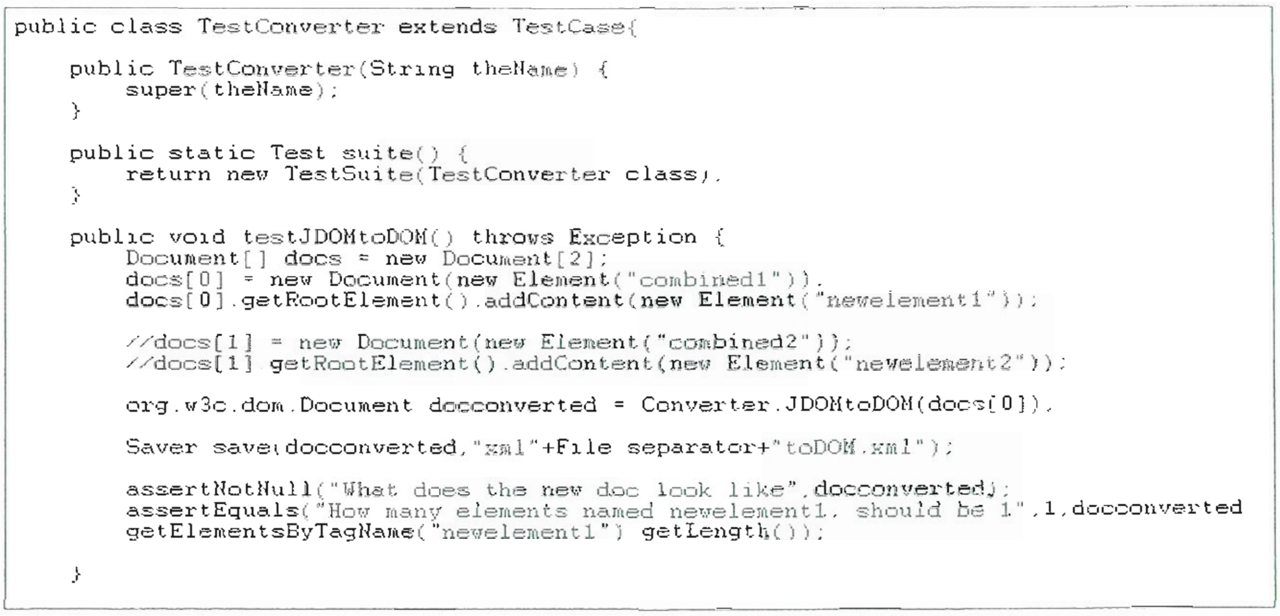
```
private static String path ="path to your properties file";

public static String getContextPath(){
    String propertiesFileName;
    MessageContext messageContext = MessageContext.getCurrentContext();
    if (messageContext != null) {
        // Get the servlet request
        HttpServletRequest request = (HttpServletRequest)messageContext.getProperty(HTTPConstants.
        MC_HTTP_SERVLETREQUEST);

        // Strip off the web service name off the end of the path
        // and append our properties file path
        propertiesFileName = request.getPathTranslated().substring(0,request.getPathTranslated().
        lastIndexOf(File.separator));
        path = propertiesFileName + File.separator + "WEB-INF" + File.separator;
    }
    return path;
}
```

Figure 19: Code allowing the context path from a web services to be found.

The code above sets a path to your properties file's directory. If the code is running as a web service, the messageContext will not be null and the real file location for the properties file directory will be returned. If the code is been unit tested, the messageContext will be null and the path returned would be the hard coded one your originally put in. The static path means that any other web service using this after the first initial web service invocation keeps the path. Only the first web service has a messageContext. Therefore, any other code using this method will not have a `MessageContext` but the path has already been set.

Unit testing code makes the code mores productive and stable. This is because:

- it *greatly* increases confidence in the correctness of your code
- it often improves the design of the class you are testing - since you spend much more time thinking about how an object is actually *used*, instead of its implementation, defects in its interface become more obvious
- failure of a test is glaringly obvious
- the positive feedback provided by successful tests produces unmistakably warm, fuzzy feelings even in the embittered heart of an experienced programmer



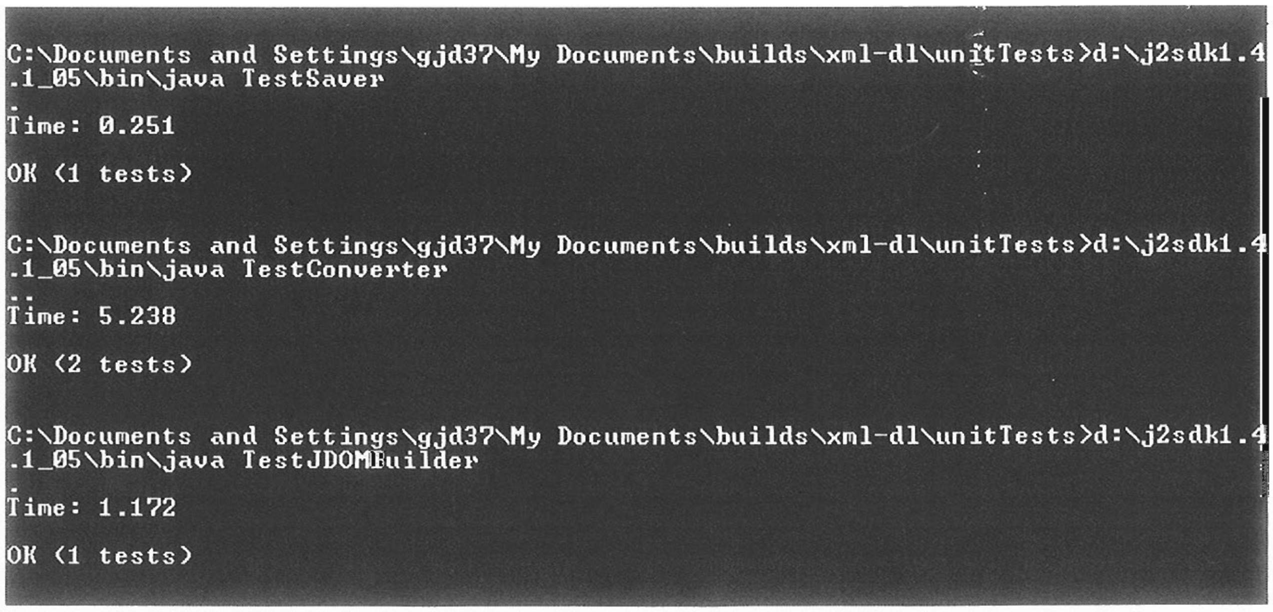
```
public class TestConverter extends TestCase{

    public TestConverter(String theName) {
        super(theName);
    }

    public static Test suite() {
        return new TestSuite(TestConverter.class);
    }

    public void testJDOMtoDOM() throws Exception {
        Document[] docs = new Document[2];
        docs[0] = new Document(new Element("combined1"));
        docs[0].getRootElement().addContent(new Element("newelement1"));

        //docs[1] = new Document(new Element("combined2"));
        //docs[1].getRootElement().addContent(new Element("newelement2"));

        org.w3c.dom.Document docconverted = Converter.JDOMtoDOM(docs[0]);

        Saver save(docconverted,"xml"+File.separator+"toDOM.xml");

        assertNotNull("What does the new doc look like",docconverted);
        assertEquals("How many elements named newelement1, should be 1",1,docconverted.
        getElementsByTagName("newelement1").getLength());

    }
```

Figure 20: Sample Unit test code.

5.1.1 Examples of a unit test code.

Unit test code is very simple. Create a class that extends `TestCase`, create a constructor with a super method and a static method suite which returns a `TestSuite`. Then create methods that test functionality of the methods that is being unit tested. At the end of the methods run assertions on the results. Figure 20 shows an example of a unit test. The test method is `testJDOMtoDOM`. The code is run and executes the methods within the class. In figure 20, it is testing that the converted document is not null and that the number of elements named newelement1 is one.

The output is similar to figure 21. It runs through each of the methods and reports if the assertions are true or false. The two assertions in the method `testJDOMtoDOM` in class `TestConverter` give two OK valued tests taking 5.238 seconds to complete the method.



```
C:\Documents and Settings\gjd37\My Documents\builds\xml-dl\unitTests>d:\j2sdk1.4
.1_05\bin\java TestSaver
.
Time: 0.251

OK (1 tests)

C:\Documents and Settings\gjd37\My Documents\builds\xml-dl\unitTests>d:\j2sdk1.4
.1_05\bin\java TestConverter
..
Time: 5.238

OK (2 tests)

C:\Documents and Settings\gjd37\My Documents\builds\xml-dl\unitTests>d:\j2sdk1.4
.1_05\bin\java TestJDOMBuilder
.
Time: 1.172

OK (1 tests)
```

Figure 21: Results from figure 17 code's unit test.

5.2 Build.xml files

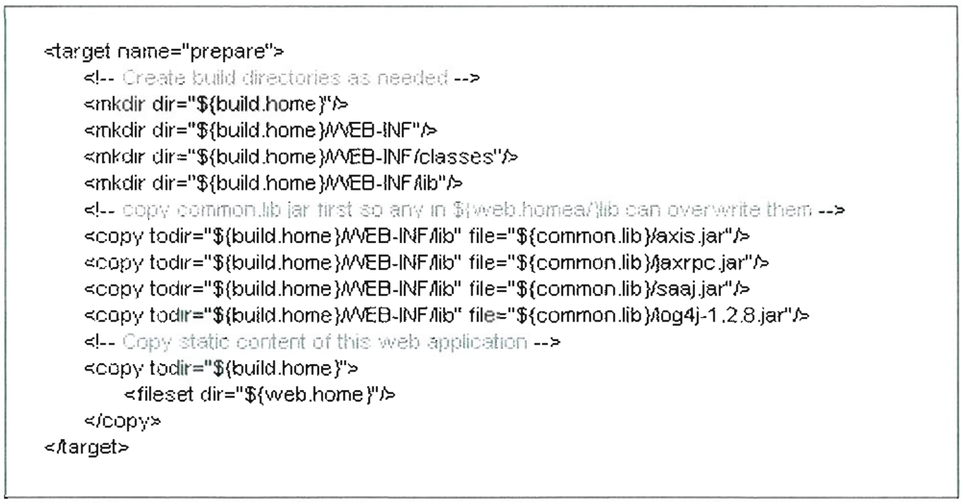
5.2.1 Problems

Currently most modules use their own jar files places in the WEB-INF/lib directory. For small applications, this is OK, but for large applications like the Data Portal, this can lead to extensive repetition of jar files throughout the application.

Large jar files like the `axis.jar` are over 1.1 megabyte each. With approximately 20 modules for an instance of the Data Portal, there is a approximately 20 megabytes of repetition adding to the size of the Data Portal. The normal download of Axis (files needed for using SOAP and web services) are `axis.jar, saaj.jar, log4j.jar, commons loggging/discovery.jar` and `jaxrpc.jar` total nearly 1.5 megabytes therefore the Data Portal is creating a 30 megabytes of extra space which is unnecessary.

5.2.2 Solution

The build.xml file has been modified to solve this problem. A new 'common' folder is added to the Data Portal, which contains all the common jar files developers use. Any jar files that are specific to a module should stay within the WEB-INF/lib folder, these are normally versioned jar files and therefore the module needs a specific version of the jar for it to function. Any other jars should be referenced within the build.xml file. Figure 22 shows a snippet from the new build.xml file. This task copies in the jar files the module needs from the common folder before it compiles the code and creates it classpath instance. This section needs to be edited to add the jar files for the module.



```
<target name="prepare">
    <!-- Create build directories as needed -->
    <mkdir dir="${build.home}"/>
    <mkdir dir="${build.home}/WEB-INF"/>
    <mkdir dir="${build.home}/WEB-INF/classes"/>
    <mkdir dir="${build.home}/WEB-INF/lib"/>
    <!-- copy common.lib jar first so any in ${web.homea/}lib can overwrite them -->
    <copy todir="${build.home}/WEB-INF/lib" file="${common.lib}/axis.jar"/>
    <copy todir="${build.home}/WEB-INF/lib" file="${common.lib}/jaxrpc.jar"/>
    <copy todir="${build.home}/WEB-INF/lib" file="${common.lib}/saaj.jar"/>
    <copy todir="${build.home}/WEB-INF/lib" file="${common.lib}/log4j-1.2.8.jar"/>
    <!-- Copy static content of this web application -->
    <copy todir="${build.home}">
        <fileset dir="${web.home}"/>
    </copy>
</target>
```

Figure 22: Section from an ant build file.

5.3 Cog - kit and Axis classloader problems

5.3.1 Problem

`http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=1242`

The problem is that Globus Cog Kit [10] calls `ClassCastExceptions` when two applications load the same jce-jdk*.jar file. This is a problem because different web applications could be using different versions of the cog kit.

This means that after the second module loads the Globus Cog Kit after the first the module, it cannot work and throws this exception indicating that the web service has failed.

5.3.2 Solution

Tomcat deploys two class loaders, a Standard class loader and a `WebClassLoader`. The standard loads the $TOMCAT_HOME/server/lib etc classes and the WebApp loads the classes in the WEB-INF/lib.

When the first application loads the Globus APIs, it will verify the certificate signature that signed the jar file. To do this the GSI library gets the signature algorithm implementation using the Security class/global `HashTable`. The Signature implementation class checks if the key passed into its `engineInitVerify()` method is of the method is of the right type. Since the Signature implementation class was loaded using the `WebappsClassLoader` it loads the key type class it refers to using the same loader as the Signature implementation. However, the key type class passed into the `engineInitVerify()` method is loaded using the `StandardClassLoader`. Therefore, the type verification fails (since its fully qualified name and its defining class loader identify a class in Java).

The key problem in this case is that one application has its own copy of the classes loaded by one class loader and it obtains a reference (indirectly) to the same class loaded by a different class loader. That happens because of the global `HashTable`. Therefore, all security providers should be loaded from one class loader. However, because axis is the class that loads the Globus classes, any class within axis loaded by the `WebappsClassLoader` that needs a class that in now from the `StandardClassLoader` (i.e. Globus) will have the same problem.

Therefore, all Axis and Globus jar files needs to be located in the common/lib directory of tomcat. Tomcat states that no application jar file is to be placed in the common directory but in the shared directory but if tomcat is instanced by using the same tomcat_base, the second instance cannot see the jar file within the shared directory and therefore needs to be put in the common.

5.4 Concurrent Versioning System (CVS)

Each module must be tagged during the development cycle with a version. All tagged modules are managed using the CVS system.

A proposed addition to the build.xml file for a CVS checkout of a version of the module. The task should checkout the cvs and copy and compile the code and copy to build directory. This should not interfere with tomcat. Since tomcat is clustered, the new update can be done without stopping or interfering with the Data Portal.

5.5 Application design using a Multi-tiered architecture

5.5.1 Presentation Tier

5.5.1.1 Web Interface

Recently JSP 2.0 and Servlet 2.4 specs have been released by Sun, which has been implemented by Apache with Tomcat 5. In addition, Java Standard Tag Libraries 1.1 [11] and Apache Supported Tag Libraries [12] have been released. JSP 2.0 includes JSTL, which provides a set of four standard tag libraries (core, internationalization/format, XML, and SQL) and supports Expression Language [13] (EL). A primary design goal for JSTL/Supported Tag Libraries and the EL was to simplify Webpage development and implementation, by separating of business logic from presentation.

5.5.1.2 Separation of business logic and presentation

The separation of business logic and presentation gives Web development the ability to remove nearly all Java code from within a JSP. This helps with modularisation of the code increasing reusability, stops the repetition of code within different JSPs and helps to separate presentation from business logic.

On any web/ enterprise project, multiple role and responsibilities will exist. E.g. a HTML designer and a software engineer. On small scale projects the role might be the same person, but on larger projects where these are to be filled my multiple individuals, who might not have overlapping skills, are less productive if made too dependant on the workflow of each other.

The separation means that development of the Java code and HTML code can be truly separate and therefore easier to work with and develop. Any combination of the roles and development means that the client layer becomes fatter and moves away from the original concepts of the web or client-server architecture.

Figure 23 shows a simple JSP. The page is valid XML, it contacts a database, applies and select statement and displays the results in a table. In this environment, a HTML developer is concealed from the complex java and concentrates on the HTML design and the software designer developers only with java, creating better code.

```
<%@ taglib uri="http://java.sun.com/jstl/core_rt" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jstl/sql_rt" prefix="sql" %>
<sql:setDataSource var="database" driver="org.postgresql.Driver" url="dbc:postgresql://*****.dl.ac.uk:5432/*****" user="***" password="*****" />

<sql:query var="users" dataSource="${database}">
select username, facility from usertable
</sql:query>

<html>
    <head>
        <title>JSP 2.0 - Basic Operations</title>
    </head>
<body>
<table>
    <c:forEach var="row" items="${users.rows}">
        <tr>
            <td>${row.username}</td>
            <td>${row.facility}</td>
        </tr>
    </c:forEach>
</table>
</body>
</html>
```

Figure 23: JSP expression language to hide the complex java code from the HTML designer.

5.5.1.2.1 Expression Language

A primary feature of JSP technology version 2.0 is its support for an expression language (EL). An expression language makes it possible to access application data stored in JavaBeans, Session, Page components. For example, the JSP expression language allows a page author to access a bean using simple syntax such as `${name}` for a simple variable or `${name.foo.bar}` for a nested property.

The JSP expression language defines a set of implicit objects. For example, printing out the Boolean value of a user's session attribute 'isLoggedIn':

```
        ${session.isLoggedIn}
```

Whereas with JSP 1.2 the code would look like:

```
<%  boolean isLoggedIn = (bolean)session.getAttibute("isLoggedIn");
out.prinln(isLogginIn);   %>
```

Here are some example EL Expressions

Expression	**Result**
`${1 < (4/2)}`	true
`${'hip' gt 'hit'}`	false
`${3 div 4}`	0.75
`${!empty param.Add}`	True if the request parameter named `Add` is null or an empty string
`${header["host"]}`	The Host
`${departments[deptName]}`	The value of the entry named `deptName` in the `departments` map
`${sessionScope.cart.numberOfItems}`	The value of the `numberOfItems` property of the session-scoped attribute named `cart`

5.5.1.2.2 Standard Tag Libraries

The JavaServer Pages Standard Tag Library (JSTL) encapsulates, as simple tags, core functionality common to many JSP applications. For example, simplifies coding when iterating over lists using a scriptlet or different iteration tags from numerous vendors, JSTL defines a standard tag that works the same everywhere. This standardization provides a single tag and use it on multiple JSP containers. Also, when tags are standard, containers can recognize them and optimize their implementations.

JSTL has support for common, structural tasks such as iteration and conditionals, tags for manipulating XML documents, internationalization and locale-sensitive formatting tags, and SQL tags. It also introduces a new expression language to simplify page development, and it provides an API for developers to simplify the configuration of JSTL tags and the development of custom tags that conform to JSTL conventions.

5.5.1.2.2.1 Core tag library

Here are some examples of the core JSTL, which covers flow control, exception handling etc which is vital to all other Tag Libraries.

5.5.1.2.2.1.1 Displaying/setting values and exception handling

The core library's most basic tag is the `c:out` tag, which displays an EL expression's value in a page. An example expression that uses `c:out` might look like this:

We have `<c:out value="${applicationScope.product.inventoryCount}" escapeXml="true" default="0"  />` of those items in stock.

In the above, the value attribute is the expression that is send to the page output. It also showns the optional escapeXml attribute, which specifies whether XML characters (<, >, &, and .) should convert to corresponding character entity codes (so they show up as those characters in an HTML page), and the default attribute, which is used if the EL can't evaluate the value or the value evaluates to null.

Note that when EL support is fully implemented in JSP 2.0, there won't be any need to use the `c:out` action; just embed JSP expressions directly in the page.

Another commonly used core action is `c:set`, which sets a variable in a page. Use `c:set` in two ways. The first way sets the variable defined in the var attribute to the value defined in the value attribute, as shown below:

```
        <c:set var="customerID" value="$param:customerNumber" scope="session" />
```

The optional scope attribute above specifies that it wishes to set the variable customerID in the session scope; if scope is not specified, it defaults to page scope.

5.5.1.2.2.1.2 Exception Handling

JSTL has made exception handling a bit easier. In typical JSP pages, there are two approaches for handling exceptions: `try/catch` blocks in scriptlet code embedded directly in the page or with a JSP errorPage directive that provides a nice catch-all way to handle any possible exception on a page. JSTL offers a good alternative with the `c:catch` action, which provides an effective way to handle exceptions with a bit more granularity without embedding Java code in your pages. A `c:catch` action might look like this:

```
        <c:catch>
        <!-- some set of nested JSTL tags below which would be hit on an
exception-->
       </c:catch>
```

The `c:catch` action has an optional attribute, a variable that references a thrown exception.

It is not common to use the `c:remove` tag. This tag has attributes for a variable name and a scope, and removes the specified variable from the specified scope.

5.5.1.2.2.1.3 Flow control

The `c:if` action handles simple conditional tests. The Boolean expression's value in the test attribute is evaluated; if true, the body's contents are evaluated. In the action below, it shows the optional `var` attribute that stores the test results for later use in the page (or elsewhere, if the other optional scope attribute is specified):

```
        <c:if test="${status.totalVisits == 1000000}" var="visits">
          You are the millionth visitor to our site!  Congratulations!
        </c:if>
```

JSTL also supports for switching logic with `c:choose`, `c:when`, and `c:otherwise`. A set of `c:when` actions may be included within a choose tag; if any of the expressions in the `c:when` blocks evaluate to true, the following tests in the `c:choose` action are not evaluated. If none of the tests in the `c:when` blocks evaluate to true, `c:otherwise` action's contents, if present, are evaluated.

5.5.1.2.2.2 SQL Tag Library

SQL Tag Library allows easy database integration. In this snippet from figure 24, it sets up a connection to a database, applies a select statement and iterates through the results building a table.

```
<%-- open a database connection --%>
<sql:connection id="conn1">
  <sql:url>jdbc:mysql://localhost/test</sql:url>
  <sql:driver>org.gjt.mm.mysql.Driver</sql:driver>
</sql:connection>

<%-- open a database query --%>
<table>
<sql:statement id="stmt1" conn="conn1">
  <sql:query>
    select id, name, description from test_books order by 1
  </sql:query>
  <%-- loop through the rows of your query --%>
  <sql:resultSet id="rset2">
    <tr>
      <td><sql:getColumn position="1"/></td>
      <td><sql:getColumn position="2"/></td>
      <td><sql:getColumn position="3"/>
          <sql:wasNull>[no description]</sql:wasNull></td>
    </tr>
  </sql:resultSet>
</sql:statement>
</table>

<%-- close a database connection --%>
<sql:closeConnection conn="conn1"/>
```

Figure 24: Sample JSP code using JSTL to connect to a database.

5.5.1.2.3 Apache Tag Libraries

These are Supported Taglibs from Apache such as IO, Regexp, Session etc. This allows a JSP developer to have repeated java code hidden from them completely, sometimes trivial and sometimes very difficult. These Supported Tag libraries are officially supported at Jakarta Taglibs. It is important to note that the functionality covered by some of these tag libraries may coincide with standardization efforts in the Java Community Process (JCP), both presently and in the future. Here are just a few examples.

5.5.1.2.3.1 Dates

Normally, to out printing a date use: `<% = new Date()%>` giving the output 'Thu Dec 15 17:12:53 GMT 2003'. This is an undesirable way to show a date. Using taglibs this would be:

```
      <dt:format pattern="MM/dd/yyyy HH:mm">
            <dt:currentTime/>
```

```
        </dt:format>
```

giving the output 12/15/03 13:05.
A more complex example, this outputs the same information as above but in Chicago's time zone.

```
        <dt:timeZone id="tz">America/Chicago</dt:timeZone>
```

The current time in America/Chicago is:

```
        <dt:format timeZone="tz" pattern="MM/dd/yyyy hh:mm">
                <dt:currentTime/>
        </dt:format>
```

5.5.1.2.3.2 XML/ XSL

Xml can be very tricky to style within a JSP. Taglibs allow a stylesheet transformation to be as simple as.

```
        <xtags:style xml="foo.xml" xsl="bar.xsl" />
```

A more complicated styelsheet transformation would be to transform an XML document but only show the 2nd to 5th elements.

```
        <xtags:style xml="foo.xml" xsl="bar.xsl>
                <xtags:param name="min" value="2"/>
                <xtags:param name="max" value="5"/>
        </xtags:style>
```

5.5.1.3 Java Server Faces

Java Server Faces [14] is another technology that is coming out of Sun's JSR process. JavaServer Faces technology simplifies building user interfaces for JavaServer applications. Developers of various skill levels can quickly build web applications by: assembling reusable UI components in a page; connecting these components to an application data source; and wiring client-generated events to server-side event handlers. With the power of JavaServer Faces technology, these web applications handle all of the complexity of managing the user interface on the server, allowing the application developer to focus on application code. The complexity of Java Script and validation and workflow is removed from the client to the server.

JavaServer Faces technology includes:

- A set of APIs for: representing UI components and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility.
- A JavaServer Pages (JSP) custom tag library for expressing a JavaServer Faces interface within a JSP page.

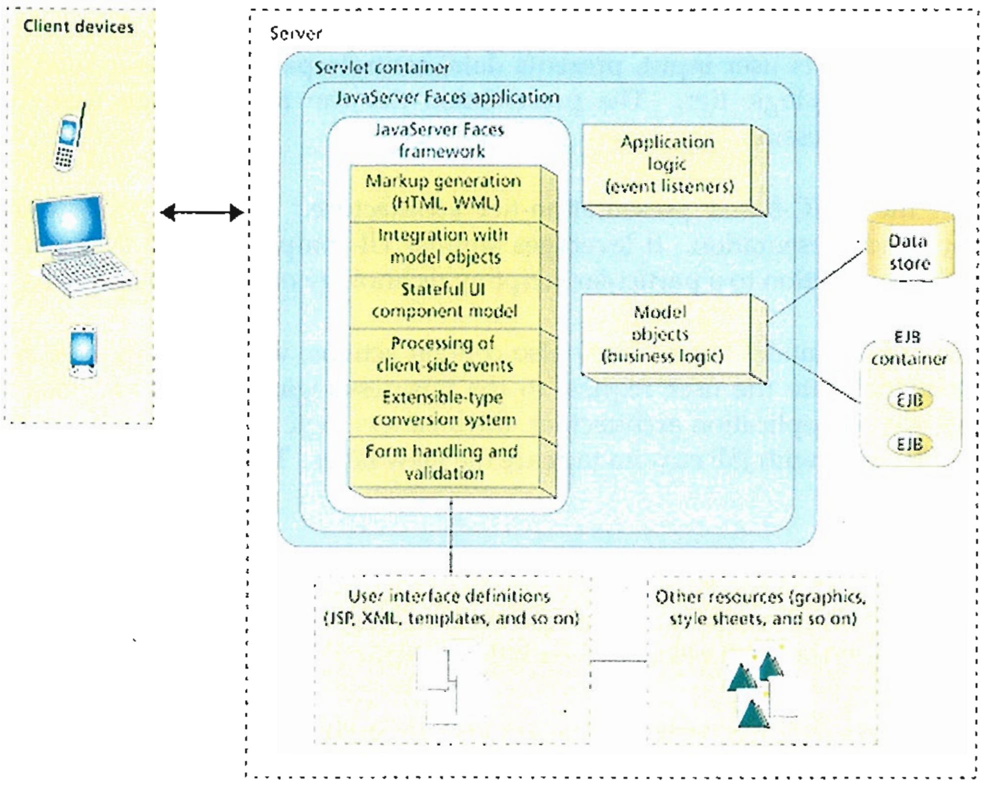

Client devices
Server
Servlet container
JavaServer Faces application
JavaServer Faces framework
Markup generation (HTML, WML)
Integration with model objects
Stateful UI component model
Processing of client-side events
Extensible-type conversion system
Form handling and validation
Application logic (event listeners)
Model objects (business logic)
Data store
EJB container
EJB
EJB
User interface definitions (JSP, XML, templates, and so on)
Other resources (graphics, style sheets, and so on)

Figure 25: Overview of the JSF architecture

5.5.1.3.1 Key benefits

It is a framework for building Web-based user interfaces in Java. Like Swing, it provides a set of standard widgets (buttons, hyperlinks, checkboxes, and so on); a model for creating custom widgets; a way to process client-generated events (such as changing the value of a text box or clicking on a button) on the server; and excellent tool support.

Since Web-based applications, unlike their Swing cousins, must often appease multiple clients (desktop browsers, phones, and PDAs), JSF has a powerful architecture for displaying components in different ways. It also has extensible facilities for validating input (the length of a field to the format of a field, for example) and converting objects to and from strings for display. In addition, Faces can automatically keep your user interface components in sync with your business (or *model*) objects. The figure 25 shows the architecture of the system.

5.5.1.3.2 Future directions

The Data Portal must keep abreast of future technologies and applications, one such technology is the new Portlet specification by Sun [15] and the implementation of a Portlet application Sakai [16]. Sakai is a framework that builds on the recently ratified JSR 168 portlet standard and the open service interface (OKI) definitions to create a services-based, enterprise portal for tool delivery.

Making the Data Portal pluggable into a Portlet framework application like Sakai would be benificial for the Data Portal and other services / portals that integrate with each other. The Web Interface is the only section of the Data Portal that needs to be remodelled to allow portlet API and applications to be able to contact and use the Data Portal.

5.5.1.4 Presentation tier and JavaServer Faces

The presentation tier collects user input, presents data, controls page navigation, and delegates user input to the business-logic tier. The presentation tier can also validate user input and maintain the application's session state.

JSF fits well with the MVC-based presentation-tier architecture. It offers a clean separation between behaviour and presentation. It leverages familiar UI-component and Web-tier concepts without limiting the application to a particular scripting technology or mark-up language.

JSF backing beans are the model layer. They also contain actions, which are an extension of the controller layer and delegate the user request to the business-logic tier. Please note, from the perspective of the overall application architecture, the business-logic tier can also be referred to as the model layer. JSP pages with JSF custom tags are the view layer. The Faces Servlet provides the controller's functionality.

5.5.2 Business-logic and the Spring framework

Business objects and business services exist in the business-logic tier. A business object contains not only the data, but also the logic associated with that specific object.

Business services interact with business objects and provide higher-level business logic. A formal business interface layer should be defined, which contains the service interfaces that the client uses directly. Spring Framework [17], implements the business-logic tier.

Spring is based on the concept of inversion of control (IOC) or Dependency Injection. This concept means that you do not create your objects, you describe how they should be created. You do not directly connect your components and services together in code, you describe which services are needed by which components, and the container is responsible for hooking it all together. Because Spring links objects together instead of the objects linking themselves together, it is categorized as a 'dependency injection' or 'inversion of control' framework.

Spring's object linking is defined in XML files, thus during runtime different components can be plugged-in, or for different application configurations. This is particularly useful for applications that do unit testing or applications that deploy different configurations for different customers.

A Spring/Hibernate combination is a nice alternative to EJBs. With a large complex, distributed, clustered applications that have massive through put then EJBs is the solution, but with applications that do not need this then the Spring/Hibernate combination gives you most of the features of EJBs but without the complexity, for example declarative transaction management, pooling, security, resource lookups etc. This combination can be described as a lightweight version of EJBs.

5.5.3 Integration tier and Hibernate

Proposed to use an ORM (Object Relational mapping) when using java with relational databases with complex schemas or when need the application to be multiplatform with different databases. The Session Manager and Shopping Cart are examples of modules that could benefit from using this. With a good ORM, it is possible to define the way to map classes to tables once - which property maps to which column, which class to which table, etc. After this, it becomes much easier to communicate with databases. This is similar to an EJB entity bean, where a class maps on to a row in a table, and the class variables maps to a column.

With a good ORM, plain java objects can be used in the application to tell the ORM to persist them:

```
    orm.save(myObject);
```

This will automatically generate all the SQL needed to store the object. An ORM allows loading of objects just as easily:

```
    myObject = orm.load(MyObject.class, objectId);
```

A good ORM will feature a query language too:

```
  List myObjects = orm.find(
        "FROM MyObject object WHERE object.property = 5");
```

This will probably translate to an SQL query using multiple joins, which would be much more complicated to write. An ORM will also automatically populate the returned objects with their data, and even their associations (if necessary).

One such implementation for ORM is Hibernate [18], it is an open source ORM framework that relieves the need to use the JDBC API. Hibernate supports all major SQL database management systems and is one of the most mature and most complete open source object-relational mapper out there. The Hibernate Query Language, designed as a minimal object-oriented extension to SQL, provides an elegant bridge between the object and relational worlds. Hibernate offers facilities for data retrieval and update, transaction management, database connection pooling, programmatic and declarative queries, and declarative entity relationship management.

Hibernate is also less invasive than other ORM frameworks. Reflection and runtime bytecode generation are used, and SQL generation occurs at system startup. It allows us to develop persistent objects following common Java idiom—including association, inheritance, polymorphism, composition, and the Java Collections Framework.

Vendor-transparency is just one advantage of using Hibernate. Another big advantage is that Hibernate abstracts JDBC code from the business objects, Hibernate does it in the background. Instead leaving the developer to concentrate on building business objects. In addition, business objects are mapped to the db schema using XML configuration files so changing the table structure may require nothing more than modifying the XML file. Depending on the changes you make to the table, changes may have to be made to the business object, but that cannot be helped that no matter what persistence mechanism is used.

For example, a new column is added into your table:

A) JDBC

- Add the new field into the business object
- Modify the JDBC method that performs the "select" in order to include the new column
- Modify the JDBC method that performs the "insert" in order to add a new value into the new column.

- Modify the JDBC method that performs the "update" in order to update an existing value in your new column

B) Hibernate

- Add the new field into the business object
- Modify the Hibernate XML mapping file to include the new column

5.6 Security

5.6.1 Session ID

The session id is passed as plain text within the Data Portal (unless it is communicating via SSL). Even though each web service resides on a single machine and therefore eliminating any snooping of the web service packets and the session id and theoretically, spoofing a user's session, it would be recommended that the session id is passed encrypted in the future when the Data Portal becomes more heterogeneous.

Java Web Service Developers Pack 1.4 [19] has now a full implementation of XML Encryption and XML Signature specification by OASIS. Released in July 2004. The sensitive information that the Data Portal web services pass around must all be sent encrypted either using Java Web Service Developers Pack 1.4 or another implementation of XML Encryption and XML Signature.

5.6.1.1 Certificate delegation

The delegation of the user's certificate is best done using GSI delegation. The Data Portal team have developed code that achieved this via web services. This should be added to each web service module that requires a user proxy certificate.

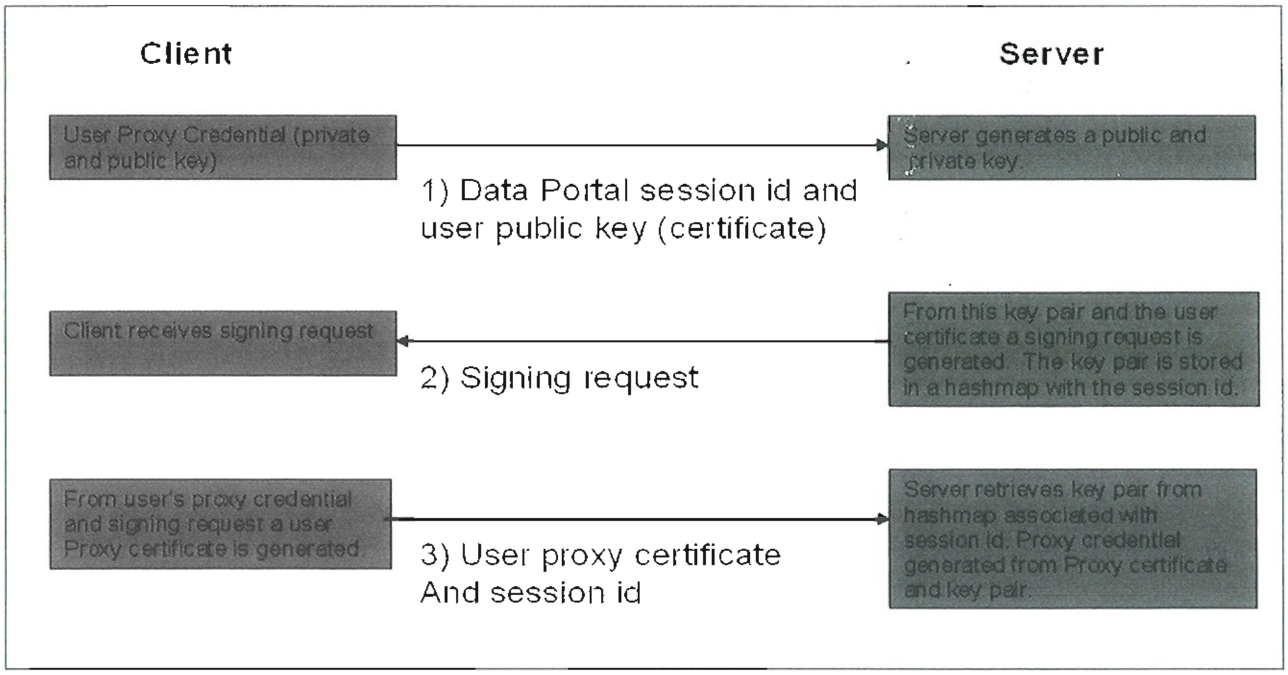
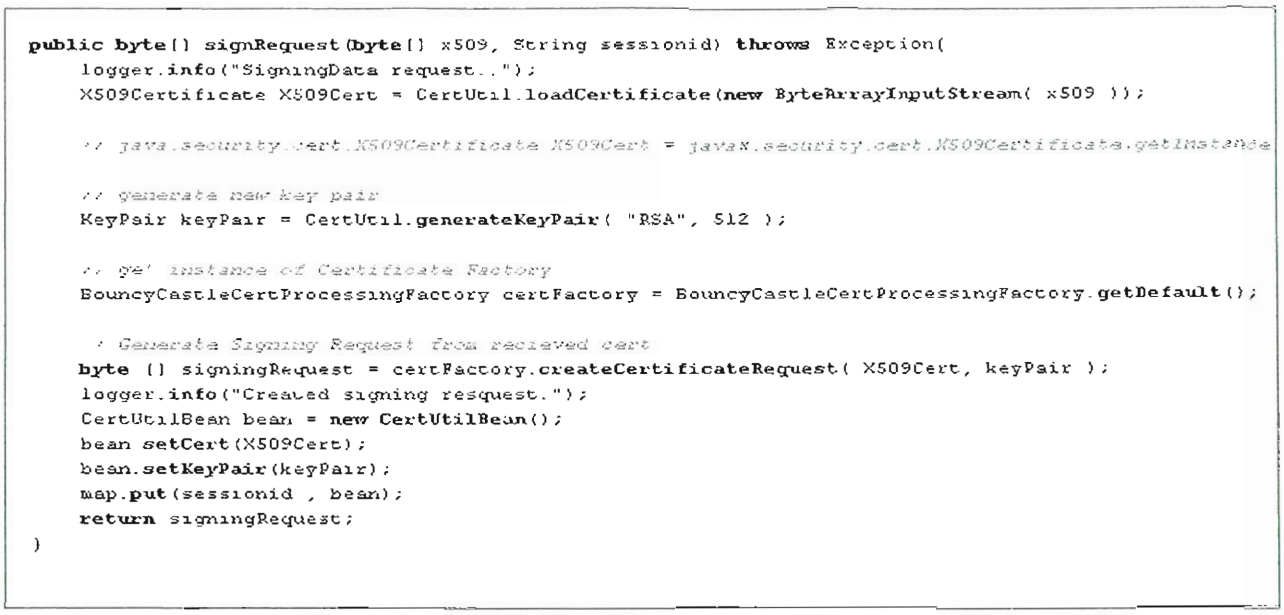

Client
Server
User Proxy Credential (private and public key)
Server generates a public and private key.
1) Data Portal session id and user public key (certificate)
Client receives signing request
From this key pair and the user certificate a signing request is generated. The key pair is stored in a hashmap with the session id.
2) Signing request
From user's proxy credential and signing request a user Proxy certificate is generated.
Server retrieves key pair from hashmap associated with session id. Proxy credential generated from Proxy certificate and key pair.
3) User proxy certificate And session id

Figure 26: Two web service invocations to show the delagation of a proxy credential.



```
public byte[] signRequest(byte[] x509, String sessionid) throws Exception{
    logger.info("SigningData request..");
    X509Certificate X509Cert = CertUtil.loadCertificate(new ByteArrayInputStream( x509 ));

    // java.security.cert.X509Certificate X509Cert = javax.security.cert.X509Certificate.getInstance

    // generate new key pair
    KeyPair keyPair = CertUtil.generateKeyPair( "RSA", 512 );

    // get instance of Certificate Factory
    BouncyCastleCertProcessingFactory certFactory = BouncyCastleCertProcessingFactory.getDefault();

     / Generate Signing Request from recieved cert
    byte [] signingRequest = certFactory.createCertificateRequest( X509Cert, keyPair );
    logger.info("Created signing resquest.");
    CertUtilBean bean = new CertUtilBean();
    bean setCert(X509Cert);
    bean.setKeyPair(keyPair);
    map.put(sessionid , bean);
    return signingRequest;
}
```

Figure 27: Sample code from the Web Service for credential delegation

The code in figure 27 requires two web service invocations. The first requesting a signing request for the delegation by sending a session id and the delegated certificate of the user which generates and sends back the signing request. The second invocation creates a delegated certificate from the signing request and sends this back to the web service. This web service then creates a delegated credential from the delegated certificate therefore has a delegated credential for the user without having to move a private key. This process should either be invoked over SSL or the information sent encrypted using XML Encryption.

5.6.1.1.1 Globus httpg

This protocol is used with Globus Tool Kit 3 [20]. The protocol allows secure transfer of a user's certificate over the network. The client code is below is similar to the code used in normal axis invocations but notice the URL. It is using httpg protocol over 8443, which is encrypted, and it sets certain properties, e.g. the user's credentials. The problem with this protocol is that the web service itself needs a grid map file to map a user to a local user on the server machine, if the user's credential's DN is not present in the grid map file the web service invocation will fail.

```
SimpleProvider provider = new SimpleProvider();

SimpleTargetedChain c = null;

c = new SimpleTargetedChain(new GSIHTTPSender());
provider.deployTransport("httpg", c);

c = new SimpleTargetedChain(new HTTPSender());
provider.deployTransport("http", c);

    Service  service = new Service(provider);
    Call     call    = (Call) service.createCall();

// set users globus credentials
call.setProperty(GSIHTTPTransport.GSI_CREDENTIALS,  cred);

// sets authorization type
call.setProperty(GSIHTTPTransport.GSI_AUTHORIZATION,
SelfAuthorization.getInstance());

// sets gsi mode
call.setProperty(GSIHTTPTransport.GSI_MODE,
GSIHTTPTransport.GSI_MODE_LIMITED_DELEG);

call.setTargetEndpointAddress( new URL("httpg://localhost:8443/axis/services/Test") );
call.setOperationName(new QName("method",  "serviceMethod"));
call.addParameter( "arg1", XMLType.XSD_STRING, ParameterMode.IN);
call.setReturnType( XMLType.XSD_STRING );

String ret = (String) call.invoke( new Object[] { textToSend } );

System.out.println("Service response : " + ret);


} catch (Exception e) {
e.printStackTrace();
}
```

Figure 28: Client code for sending a delegated proxy to a web services.

```
MessageContext ctx MessageContext.getCurrentContext();
setUpEnv(ctx);
GSSCredential cred  =  (GSSCredential)ctx.getProperty(GSIConstants.GSI_CREDENTIALS);
```

Figure 29: Server code for extracting the proxy from a web service invocation.

Where setUpEnv(ctx) is a simple method set up the properties so it is read to extract the credential..

Since the Data Portal's access is limited only by the configuration of the MyProxy server that the Data Portal trusts, it is not feasible to have a large and constantly changing and updating grid map file. The core Data Portal modules should only use the certificate delegation that the Data Portal team has developed. Any certificate delegation to outside services, i.e. XMLWrapper should use httpg protocol

The code for the client and server sections for the web service delegating credentials should be created and maintained as an API to allow the functionality of this concept to be applied without the knowledge of the delegation process, i.e. the signing request and credential delegation steps.

A more detailed research into globus httpg and certificate delegation needs to be undertaken to fully understand the security implications and how to delegate certificates via web services.

5.7 Optimising Java code

Once the unit testing, profiling and stress testing has identified areas for improvement, the code needs to be speeded up and enhanced.

There are normally four categories for optimising techniques:

- J2SE Optimisations
- J2EE Optimisations
- Resource Pooling
- Servlet Container Configuration

It is worth noting that certain code maintainability techniques can slow down J2EE applications, i.e. Tag Libraries can often slow down JSP pages. Therefore, any concessions in terms of code maintainability must be necessary by measuring the performance gains against the maintainability.

5.7.1 J2SE Optimisations

Techniques for optimizing Java 2 Standard Edition features are covered all over the internet. Search on "java performance" on www.google.com. Simple java mistakes can easily hinder the performance of an application. Large iterations that create new `String` or other objects inside the iteration or large amounts of `String` concatenations can take up or/and increase the heap memory for the application at a surprisingly fast rate. Any concatenations of `Strings` should be implemented with `StringBuffer.append()` methods.

An easy way for simple optimisation is to replace the JVM or to stress test your application with different JVMs. IBM JVM [21] is a popular one.

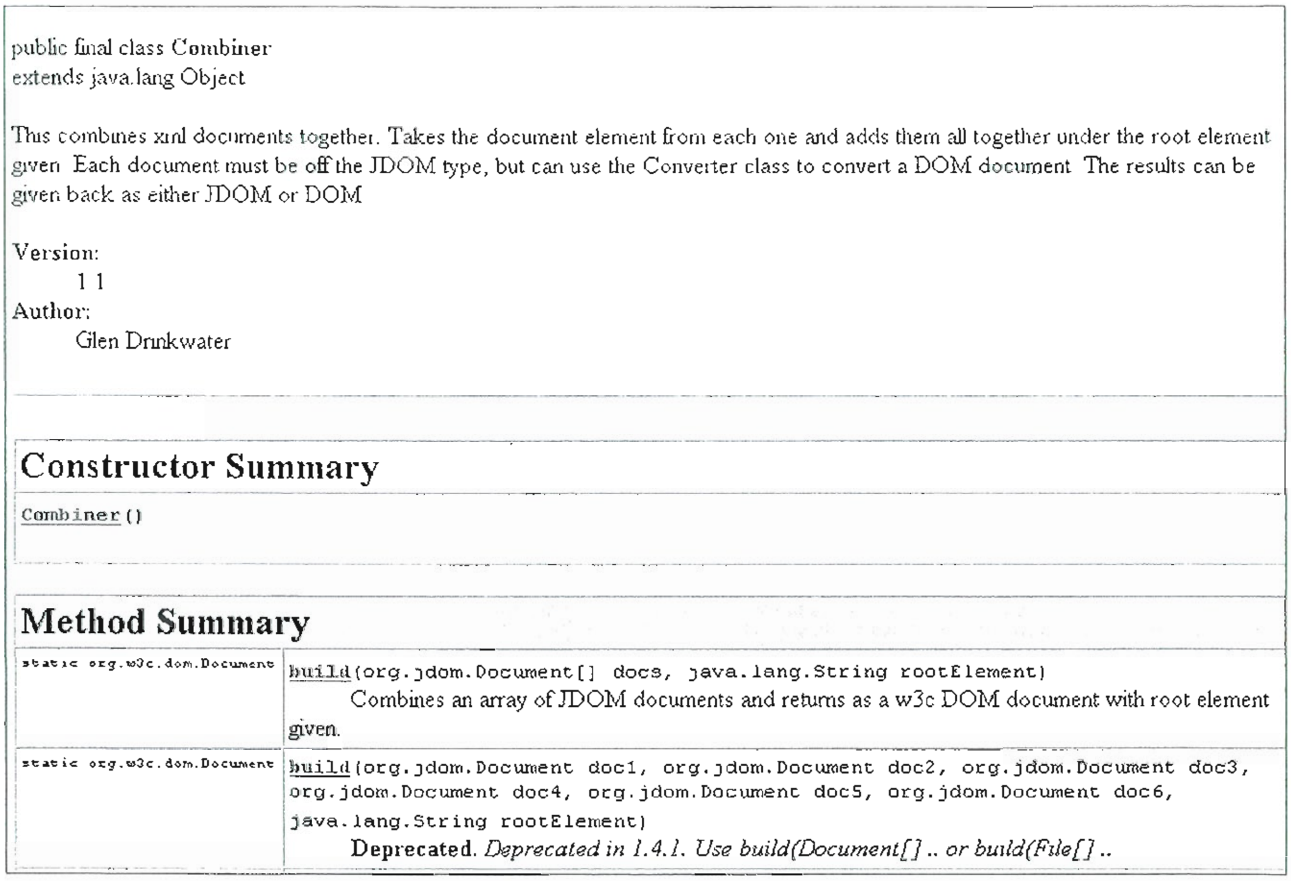
Use HotSpot profiling optimiser use JAVA_OPTS="-server", this is customized for long running server applications. Sun's Java 1.4 improves performance for servlet containers by approximately 35%. It is also possible to tune the JVM Garbage collection. Tomcat will freeze processing of all requests while the JVM is performing GC, on a poorly tuned JVM this can last 10's of seconds. Most GC's should take < 1 second and never exceed 10 seconds.

Make sure the java process always keeps the memory it uses resident in physical memory and not swapped out to virtual memory. JVM garbage collection performance [22] [23] can degrade significantly if the JVM stack gets swapped out to disk as virtual memory.

5.7.2 J2EE Optimisations

Here are a few examples for simple but effective ways to increase performance for J2EE applications.

- Factor out constant computations from loops. In case of Servlets, push one time computations into the `init()` method. Also use the servlet `init()` method to cache static data, and release them in the `destroy()` method.
- Flush the data in sections so that the user can see partial pages more quickly.



public final class Combiner
extends java.lang Object

This combines xml documents together. Takes the document element from each one and adds them all together under the root element given Each document must be off the JDOM type, but can use the Converter class to convert a DOM document The results can be given back as either JDOM or DOM

Version:
1 1
Author:
Glen Drinkwater

Constructor Summary

`Combiner()`

Method Summary

`static org.w3c.dom.Document`	`build(org.jdom.Document[] docs, java.lang.String rootElement)` Combines an array of JDOM documents and returns as a w3c DOM document with root element given.
`static org.w3c.dom.Document`	`build(org.jdom.Document doc1, org.jdom.Document doc2, org.jdom.Document doc3, org.jdom.Document doc4, org.jdom.Document doc5, org.jdom.Document doc6, java.lang.String rootElement)` **Deprecated.** *Deprecated in 1.4.1. Use build(Document[] .. or build(File[] ..*

Figure 32: HTML page showing the results of java code run though javadoc

6.2 Configuration

6.2.1 Databases

If the module requires and database for it functionality, scripts should be provided to build the tables in the database. These configuration scripts should be available for most of the common databases e.g., MySQL, HypersonicSQL PostgreSQL.

6.2.2 Configuration Files

These files contain the information that the module needs to function. E.g. URL for the Lookup Module, Public Private keys etc. The file should explain what each property is and what it is needed for with and example configuration file.

6.2.3 Configuration Help

In addition to the configuration file, there must be a help file, either pdf or HTML. This explains the process of installing the module, where the files are and what they do and other important information, i.e. the WSDL file for the web service.

6.3 Description

Maybe the same as 6.2.3, but a file needs to be present with a description of the web services that are available from this module. How are the web services called, example code to invoke the, what functionality does it provide and other services it interacts with etc.

7. WHATS NEEDED FOR A DATA PORTAL

7.1 Requirements

7.1.1 Operating System

The Data Portal can run under any operating system that fully supports java (JSDK 1.3) which at the moment is Linux (Red Hat 7+ and SuSE 8+), Solaris SPARC (7+), Solaris x86 (7+), Mac OS (10.2.6+) and Windows (98+).

7.1.1.1 Future Data Portal Versions

The next version of the Data Portal will need to support JSDK 1.4. This is the same list of operating systems as section 7.1.1.

7.1.2 Servlet container.

The Data Portal requires a servlet container that supports JSP 1.2 and Servlet 2.3 specification from Sun. Many containers support this including Jetty, Tomcat, WebSphere, WebLogic, Oracle and Sun One Application server.

7.1.2.1 Future Data Portal Versions

The next releases of the Data Portal will require JSP 2 and Servlet 2.4 specification. For example Tomcat 5 and Sun One Application server.

7.1.3 Databases

PostgreSQL [25] is the choice of the Data Portal at the moment. The requirements of the Data Portal are that it supports BLOBs and has a JDBC driver available.

Hypersonic SQL has been successfully tested with the Data Portal. The only problem with Hypersonic is that the JDBC API for the database does not support moving backwards within the result set. This is simple to rectify. Hypersonic also has been successfully tested with the Data Portal as well as the UDDI registry.

7.1.4 UDDI

Any UDDI that conforms to UDDI4J [26]. Currently the Data Portal uses WASP UDDI 4.5 from Systinet, which conforms to UDDI v2. Others include Novels UDDI and JUDDI [27] by Apache.

Apache jUDDI has been released early this year and is built and tested on Tomcat, jUDDI can be used with most databases and works with UDDI4J.

7.1.5 Web Services

Apache Axis version 1.0+.

7.1.6 Java Cogkit

Java Cog Kit version 1.1+ with Bouncy Castle JCE provider jce-jdk-117.jar installed and configured.

7.1.7 Apache Ant

The Data Portal does not necessary need ant to compile the source code but it is highly recommended that Apache Ant [28] 1.5.x be used.

ISSN 1362-0207