

Application logging in WebSphere Application Server Community Edition

Using java.util.logging, Log4j and SLF4j

Skill Level: Intermediate

[Phani Madgula \(mabalaji@in.ibm.com\)](mailto:mabalaji@in.ibm.com)
Software Developer
IBM

25 Mar 2009

WebSphere Application Server Community Edition provides several ways to configure application logging, using `java.util.logging`, Log4j and SLF4j APIs. Though the steps to configure these logging services are to large extent independent of any application server, WebSphere Application Server requires certain tweaks to get your desired logging behavior. This tutorial walks you through these tweaks coupled with sample applications. WebSphere® Application Server Community Edition is freely available for download, so you can get started in just a few minutes.

Section 1. Introduction

Application logging provides ways to capture various events happening in the execution of an application. It gathers information in detail about what the application is doing when the application performs various tasks. This information is useful for debugging, troubleshooting and even auditing. WebSphere® Application Server Community Edition (hereafter called Community Edition) ships with various libraries that help application developers configure logging services. These libraries are:

- Log4j
- SLF4j

- [java.util.logging](#)

The `java.util.logging` package is a Java API for logging that is available in all standard Java development kits. This tutorial explains with samples how to use these APIs in applications deployed on Community Edition.

You configure `java.util.logging` per JVM instance. Once configured, it is available to all the applications running on that server. This tutorial explains how to use `java.util.logging` in the applications running on Community Edition.

The most commonly used API for logging is Log4j from the [Apache Software Foundation](#). Community Edition ships with [Log4j](#) libraries that are used by the server modules at runtime. Applications can also use these libraries; they can either log messages to the same destination as the server logs at the runtime, or they can configure their own logging destinations and formats as desired. This tutorial demonstrates different ways to configure logging using Log4j in Community Edition.

The Simple Logging Facade for Java (or [SLF4j](#)) is yet another logging API that applications can use as a logging service. SLF4j does not invent another logging framework but allows applications use a standard API and plug in the actual logging implementation at deployment time, such as, `NOP`, `Simple`, `log4j version 1.2`, `JDK 1.4 logging`, `JCL` and `logback`. Community Edition also ships with SLF4j libraries. This tutorial demonstrates how to use SLF4j over log4j in the applications targeted to run on Community Edition.

Community Edition v2.1 is a lightweight application server that is based on Apache Geronimo v2.1. Community Edition also includes many other defect fixes, and receives world-class support from IBM. You can [download](#) the binary images of the server for no charge. It is a fully certified server for Java EE 5.

In this tutorial

This tutorial shows how to use `java.util.logging`, Log4j and SLF4j APIs in applications targeted to run on Community Edition. It contains the following sections:

- [Setting up the environment](#)
- [Configuring `java.util.logging` in Community Edition](#)
- [Community Edition nuances and getting the most from Log4j](#)
- [Using SLF4j in the Community Edition](#)

In each section, we briefly describe the corresponding logging API and explain various ways to configure and use the API to obtain desired logging behavior. We use the `EMPDemo` sample to demonstrate how to use the three logging APIs. The

EMPDemo sample can be downloaded from the [developerWorks](#) site.

Prerequisites

You should be fairly skilled in Java programming. Understanding Java EE 5 concepts and database concepts will help you with the context of the tutorial. If you have experience running a HelloWorld sample on Community Edition and writing Community Edition deployment plans, you are the perfect candidate to get most out of the tutorial.

System Requirements

To develop, deploy and run the application, the following environment is required:

- IBM Java SDK v1.5.0 SR8 or above
- Community Edition v2.1.0.1 or above

We use the Apache Derby database shipped with Community Edition to deploy and run the EMPDemo application. The EMPDemo application connects to EMPLOYEE_DB in the embedded Derby database and retrieves information from the EMPLOYEE table. We then show the retrieved information to the user on the browser. We demonstrate using logging APIs to log messages when the application performs various database operations.

Duration

2 hours

Section 2. Setting up the environment

In this section, you perform the following tasks:

- Install Community Edition v2.1.0.1
- Create the EMPLOYEE_DB database in the embedded derby database.
- Deploy the EMPLOYEE_DS data source on the EMPLOYEE_DB database.

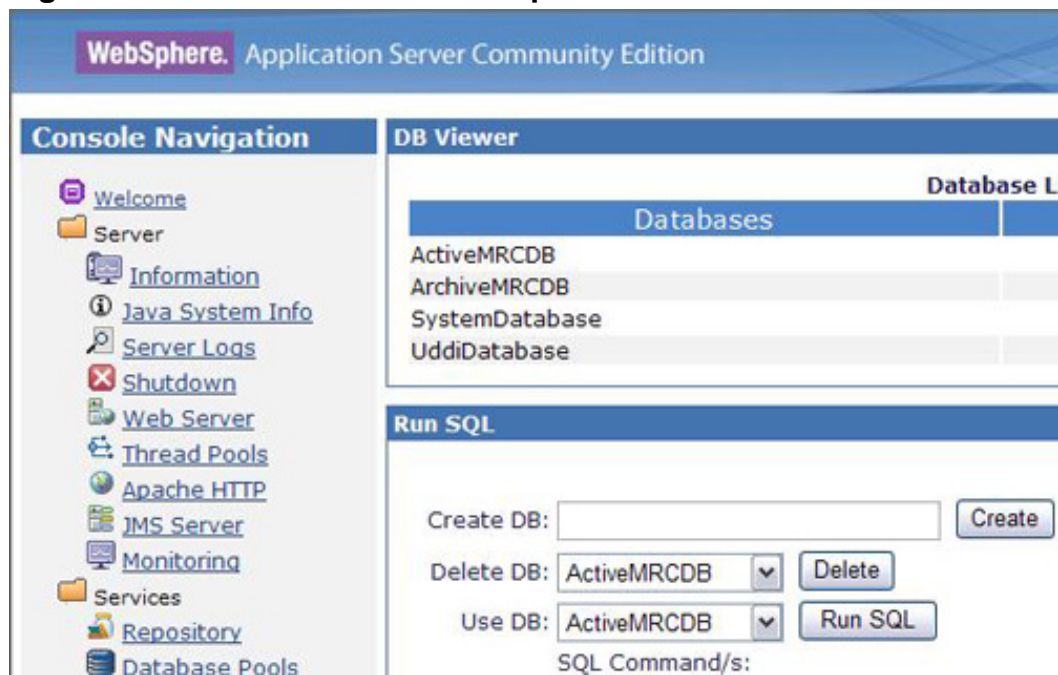
Install Community Edition v2.1.0.1

The Community Edition installer is available for download from [developerWorks](#). Download the server to your machine and follow the instructions in the Community Edition v2.1 [documentation](#). In this tutorial, we refer to the Community Edition installation directory as `<wasce_home>`. The Community Edition installer ships the IBM Java SDK1.5.0. This JDK is used by the installer as well as the server run time. For more information about the recommended and compatible platforms, see the [support site](#).

Create EMPLOYEE_DB database in the embedded derby database

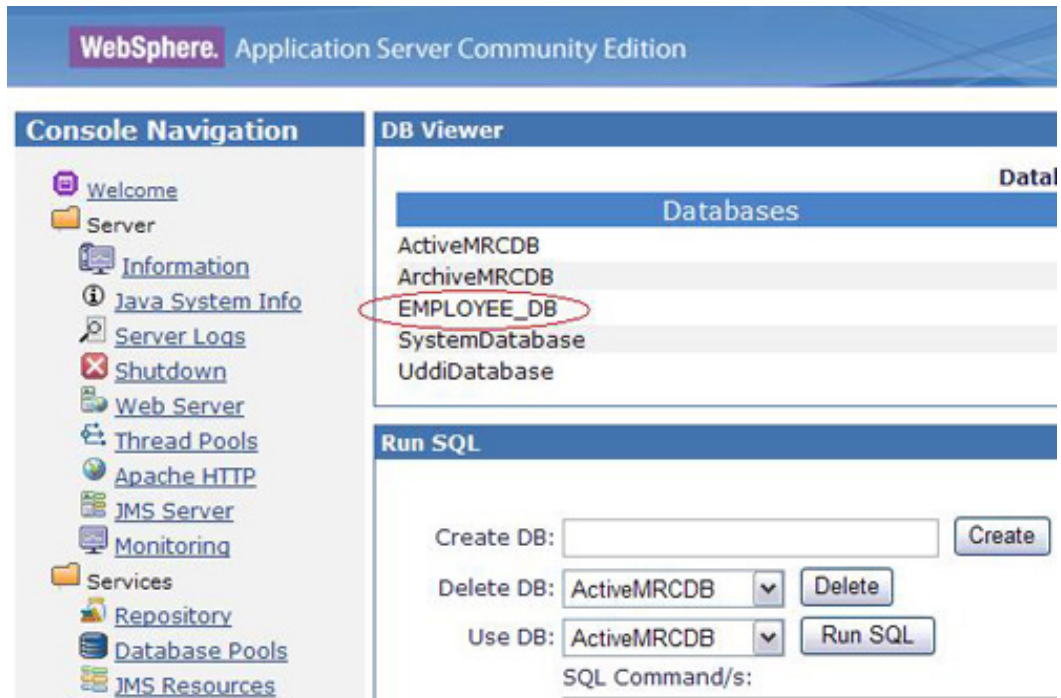
1. Start the Community Edition server and open the Web administration console by pointing the browser window to <http://localhost:8080/console>
2. Log into the administration console by providing `system` as the user name and `manager` as the password.
3. On the left hand side of the administration console, in the **Console Navigation** portlet, click on the **DB Manager** link to open up **DB Viewer** and **Run SQL** portlets on the right side, as Figure 1 shows:

Figure 1. DB Viewer and Run SQL portlets communication



- On the **Run SQL** portlet, enter `EMPLOYEE_DB` in the **Create DB** textbox and click on the **Create** button, which will create an **EMPLOYEE_DB** database in the embedded Derby database. After it's created, the **Database List** portlet lists the **EMPLOYEE_DB** database, as Figure 2 shows.

Figure 2. EMPLOYEE_DB in DB Viewer portlet



- Click on **Application** link for the **EMPLOYEE_DB** database to show a list of application tables created in the database. Currently, no application tables exist in the database. In the **SQL Command/s** text area, enter the SQL statement in Listing 1 to create the **EMPLOYEE** table in the database:

Listing 1. SQL statement to create a table

```
create table EMPLOYEE (EMPNO int, ENAME varchar(50), JOB varchar(10),
MGR varchar(10), SAL decimal(15,2), COMM decimal(15,2), DEPTNO int);
```

- Similarly, insert some sample rows in the database using the below SQL statements.

Listing 2. SQL statement to insert rows

```
insert into EMPLOYEE values (1, 'PHANI', 'SSE', 'NIKHIL', 10000, 15, 100);
insert into EMPLOYEE values (2, 'JOE', 'SSE', 'NIKHIL', 12000, 15, 100);
insert into EMPLOYEE values (3, 'JOHN', 'SSE', 'BOB', 13000, 15, 200);
```

7. Finally, the table should look like Figure 3:
Figure 3. Final EMPLOYEE_DB table

DB: EMPLOYEE_DB Table: APP.EMPLOYEE						
EMPNO	ENAME	JOB	MGR	SAL	COMM	DEPTNO
1	PHANI	SSE	NIKHIL	10000.00	15.00	100
2	JOE	SSE	NIKHIL	12000.00	15.00	100
3	JOHN	SSE	BOB	13000.00	15.00	200

[View Tables](#) | [View Databases](#)

Deploy the EMPLOYEE_DS data source on the EMPLOYEE_DB database

1. Click **Database Pools** in the **Console Navigation** portlet on the administration console. This step opens the **Database Pools** portlet, which lists the database pools currently deployed on the server.
2. Click **Using the Geronimo database pool wizard** to create a new database pool. Enter **EMPLOYEE_DS** in the **Name of the Database Pool** field and select **Derby embedded** in the **Database Type** combo box as shown in Figure 4. Click **Next**.

Figure 4. Creating the Database Pool

Database Pools

Create Database Pool -- Step 1: Select Name and Database

Name of Database Pool:
A name that is different than the name for any other name please).

Database Type:

The type of database the pool will connect to.

[Cancel](#)

3. On the next screen, enter **EMPLOYEE_DB** in the **Database Name** field

(Figure 5). Select the single entry in the **Driver JAR** box and click **Deploy** at the bottom of the page. This step deploys the EMPLOYEE_DS database pool. The **Database pools** portlet now shows the newly created database pool.

Figure 5. Configuring the Database Pool

Database Pools

This page edits a new or existing database pool.

Pool Name:

A name that is different than the name for any other database name please).

Pool Type: *TranQL Embedded Local Resource Adapter for Apache Derby*
A resource adaptor that provides access to an embedded Apache transaction support.

Basic Connection Properties

Driver JAR:

The JAR(s) required to make a connection to the database. Use multiple jars.
The JAR(s) should already be installed under Geronimo/rep

Create Database:

Flag indicating that the database should be created if it does

Database Name:

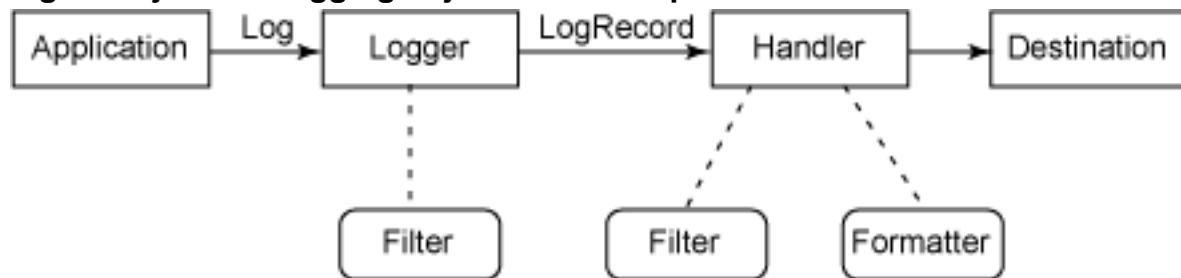
Section 3. Configuring java.util.logging in Community Edition

In this section, we briefly describe the java.util.logging API and explain how it works in Community Edition. We do not elaborate on the various features of this API.

However, we list the objects involved and show the different ways you can use this logging service in an applications.

Applications make logging calls on *Logger objects*. Logger objects are organized in a hierarchical namespace fashion, and child Loggers can inherit some logging properties from their parents in the hierarchy. The Logger objects allocate LogRecord objects when logging calls are made. They pass the LogRecord objects to Handler objects for publication to destinations. Both Loggers and Handlers may use logging levels, and optionally filters, to evaluate if they are interested in the LogRecord. If the LogRecord must be published, a Handler can optionally use a Formatter to localize and format the message before sending it to the destination. Figure 6 explains how all these objects relate:

Figure 6. java.util.logging object relationships



The java.util.logging API provides the following handlers:

- **StreamHandler**: A simple handler for writing formatted records to an OutputStream.
- **ConsoleHandler**: A simple handler for writing formatted records to System.err
- **FileHandler**: A handler that writes formatted log records either to a single file, or to a rotating set of log files.
- **SocketHandler**: A handler that writes formatted log records to remote TCP ports.
- **MemoryHandler**: A handler that buffers log records in memory.

In addition to the above handlers, the API also provides the following formatters:

- **SimpleFormatter** : Writes brief human-readable summaries of log records.
- **XMLFormatter** : Writes detailed XML-structured information

java.util.logging defines the following log levels:

- **SEVERE (highest)**
- **WARNING**
- **INFO**
- **CONFIG**
- **FINE**
- **FINER**
- **FINEST (lowest)**

In this section, we show the following aspects of the `java.util.logging`, using the EMPDemo application to illustrate how `java.util.logging` works:

- Using the default `java.util.logging` configuration
- Customizing `java.util.logging` using a gbean

Using the default `java.util.logging` configuration

By default, `java.util.logging` uses the `<JAVA_HOME>/jre/lib/logging.properties` file to configure Loggers, Handlers and Formatters. However, you can programmatically add new Handlers and Formatters at runtime. The default configuration supplied is a very simple one that configures only a `ConsoleHandler` with the `SimpleFormatter`. The log level set for `ConsoleHandler` is `INFO`; that is, messages whose log level is `INFO` or above are logged by `ConsoleHandler` by default.

We can modify the `logging.properties` file to add new Handlers or Formatters. EMPDemo contains the `com.ibm.sample.EMPDemo` servlet that connects to the `EMPLOYEE_DB` database and retrieves the rows from the `EMPLOYEE` table. It also logs messages when it performs various database operations. Listing 3 shows the corresponding code, with the logging statements marked in **bold**:

Listing 3. Default `java.util.logging`

```
Logger logger = Logger.getLogger(EMPDemo.class.getName());
logger.setLevel(Level.FINEST);
Connection con = null;
Statement stmt = null;

PrintWriter out = response.getWriter();

logger.info("Created the PrintWriter on the Response object");

try {
    Context initContext = new InitialContext();
```

```

Context envContext = Context)initContext.lookup("java:comp/env");
logger.info("Got Initial context: " +envContext);
DataSource ds = (DataSource)envContext.lookup("jdbc/DataSource");
logger.info("Got DataSource: " +ds.toString());
con = ds.getConnection();
logger.info("Got Connection: " +con.toString() +"\n");
stmt = con.createStatement();
logger.info("Got Statement : " +stmt);
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
logger.info("Table EMP after SELECT:");

out.println("Your EMP table contains the following entries:<BR>");

out.println("<table>");
out.println("<tr>");
out.println("<th>Empno</th>");
out.println("<th>Name</th>");
out.println("<th>Job</th>");
out.println("<th>Manager</th>");
out.println("<th>Salary</th>");
out.println("<th>Commission</th>");
out.println("<th>Deptno</th>");
out.println("</tr>");

    while (rs.next()) {
        String emp = rs.getString("EMPNO");
        String name = rs.getString("ENAME");
        String job = rs.getString("JOB");
        String mgr = rs.getString("MGR");
        String sal = rs.getString("SAL");
        String comm = rs.getString("COMM");
        String dept = rs.getString("DEPTNO");

        out.println("<tr>");
        out.println("<td>"+emp+"</td>");
        out.println("<td>"+name+"</td>");
        out.println("<td>"+job+"</td>");
        out.println("<td>"+mgr+"</td>");
        out.println("<td>"+sal+"</td>");
        out.println("<td>"+comm+"</td>");
        out.println("<td>"+dept+"</td>");
        out.println("</tr>");

        logger.info(emp + " " + name + " " + job);
        logger.info(" " + mgr + " " + dept);
    }
out.println("</table>");

rs.close();
stmt.close();
con.close();

logger.severe("my severe message");
logger.warning("my warning message");
logger.info("my info message");
logger.config("my config message");
logger.fine("my fine message");
logger.finer("my finer message");
logger.finest("my finest message");

    }
    catch(java.lang.Exception e) {

        e.printStackTrace();
        logger.severe(e.getClass().getName());
        logger.severe(e.getMessage());
    }
}

```

The servlet gets the Logger and overrides the default log level (which is `INFO` in the `<JAVA_HOME>/jre/lib/logging.properties` file) to `FINEST`. It starts logging the messages at the `INFO` level using the `logger.info()` method at several places during the execution. To illustrate logging at different levels, it also logs sample messages at all levels after displaying the `EMPLOYEE` table rows. Finally, in the catch block, it logs the exception at `SEVERE` level. Follow these steps to deploy and run the `EMPDemo` application:

1. Download the `EMPDemo` application. The WAR file is `EMPDemo-UtilLogging.war`.
2. Deploy the war file using this deploy command:
`<wasce_home>/bin>deploy --user system --password manager deploy EMPdemo-UtilLogging.war`
3. Access the `EMPDemo` servlet in a browser at this URL:
<http://localhost:8080/EMPDemo-UtilLogging/EMPDemo>.

You can see on the server console all `INFO`, `WARNING` and `SEVERE` messages logged by the application. The console does not show the messages logged with the other log levels even though the log level is overridden in the servlet to be `FINEST`. This is because; by default, the `ConsoleHandler` logs the messages with an `INFO` or above log level only as configured in the `<JAVA_HOME>/jre/lib/logging.properties` file. Listing 4 shows the output:

Listing 4. Messages logged on the console

```
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO:
Created the PrintWriter on the Response object
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO:
Got Initial context:
org.apache.xbean.naming.context.ImmutableContext$NestedImmutableContext@16801680
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO:
Got DataSource: org.tranql.connector.jdbc.DataSource@55425542
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO:
Got Connection: org.tranql.connector.jdbc.ConnectionHandle@30363036
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO:
Got Statement : org.tranql.connector.jdbc.StatementHandle@43aa43aa
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: Table EMP after SELECT:
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: 1 PHANI SSE
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: NIKHIL 100
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: 2 JOE SSE
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: NIKHIL 100
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: 3 JOHN SSE
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: BOB 200
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost SEVERE: my severe message
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost WARNING: my warning message
Jan 2, 2009 10:49:14 AM com.ibm.sample.EMPDemo doPost INFO: my info message
```

You can modify the `logging.properties` file to add a `FileHandler` to log the message to a file. You can also use `XMLFormatter` instead of `SimpleFormatter`. For example, set the `ConsoleHandler`'s log level to `FINEST` to observe that all the log

messages logged at all levels in the servlet are logged on the console. The line to modify is as follows.

```
java.util.logging.ConsoleHandler.level = FINEST.
```

If you have a different configuration file from the default `<JAVA_HOME>/jre/lib/logging.properties` file, you can supply the file as a JVM argument during server start-up as in Listing 5 (which is for Windows.)

Listing 5. Starting a Windows server with a properties file parameter

```
C:\>set -Djava.util.logging.config.file=<new_configuration.properties>
C:\>startup.bat
```

Customizing java.util.logging using a gbean

Sometimes you may want to tweak the logging configuration programmatically using the `java.util.logging` API. That is, add a new handler dynamically and adjust logging levels etc. Since, `java.util.logging` is configured per JVM instance; programmatic configuration is best done in a separate module rather than in any applications. That way, you can easily make any changes required to the logging configuration in this separate module. The Community Edition provides the gbean mechanism for developing and deploying custom services. This section explains the gbean service.

Listing 6 show `UtilLogPropGBean.java`, which implements a gbean service, with the relevant portion marked in **bold**:

Listing 6. UtilLogPropGBean.java

```
package com.ibm.sample;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.logging.LogManager;
import org.apache.geronimo.gbean.GBeanInfo;
import org.apache.geronimo.gbean.GBeanInfoBuilder;
import org.apache.geronimo.gbean.GBeanLifecycle;

public class UtilLogPropGBean implements GBeanLifecycle{

    private static final GBeanInfo GBEAN_INFO;
    private final String objectName;

    private String utilPropFile;

    static {
        GBeanInfoBuilder infoFactory =
            new GBeanInfoBuilder(UtilLogPropGBean.class.getName(),
                UtilLogPropGBean.class);

        infoFactory.addAttribute("objectName", String.class, false);
        infoFactory.addAttribute("utilPropFile", String.class, true);
    }
}
```

```

        infoFactory.setConstructor(
            new String[]{"objectName","utilPropFile"});
        GBEAN_INFO = infoFactory.getBeanInfo();
    }

    public UtilLogPropGBean(String objectName, String utilPropFile) {
        this.objectName = objectName;
        this.utilPropFile = utilPropFile;
    }

    public UtilLogPropGBean() {
        objectName = null;
        utilPropFile = null;
    }

    public void doFail() {
        System.out.println("UtilLogPropGBean has failed");
    }

    public void doStart(){
        LogManager logManager;

        try{
            System.out.println("[UtilLogPropGBean] GBean " + objectName + " Started");
            InputStream in = new FileInputStream(utilPropFile);
            logManager = LogManager.getLogManager();
            logManager.reset();
            logManager.readConfiguration(in);
            System.out.println("Properties file successfully read!!");

        }catch(IOException exp){
            exp.printStackTrace();
            logManager = LogManager.getLogManager();
        }catch(Exception exp){
            exp.printStackTrace();
        }
    }

    public void doStop(){
        System.out.println("GBean " + objectName + " Stopped");
    }

    public static GBeanInfo getGBeanInfo() {
        return GBEAN_INFO;
    }
}

```

When you deploy the gbean, the GBean kernel in Community Edition calls the `doStart()` method on the gbean. This method opens the `FileInputStream` on `utilPropFile`, injected to the gbean specified in the gbean deployment plan as an attribute. Then the `LogManager` reads the configuration from the input stream and configures `java.util.logging`.

Listing 7 shows the deployment plan of the gbean.

Listing 7. The gbean deployment plan

```

<module xmlns="http://geronimo.apache.org/xml/ns/deployment-1.2">
    <environment>

```

```
<moduleId>
  <groupId>UtilLogPropGBean</groupId>
  <artifactId>UtilLogPropGBean-app</artifactId>
  <version>1.0</version>
  <type>car</type>
</moduleId>

<dependencies>
  <dependency>
    <groupId>GBeans</groupId>
    <artifactId>UtilLoggingCustom</artifactId>
    <version>1.0</version>
    <type>jar</type>
  </dependency>
</dependencies>
</environment>

<gbean name="UtilLogPropGBean"
  class="com.ibm.sample.UtilLogPropGBean" xsi:type="dep:gbeanType"
  xmlns:dep="http://geronimo.apache.org/xml/ns/deployment-1.2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <attribute name="utilPropFile">
    C:/temp/applevellogging/UtilLogging/UtilLogging.properties
  </attribute>
</gbean>
</module>
```

In Listing 6, the `java.util.logging` configuration properties file, `C:/temp/applevellogging/UtilLogging/UtilLogging.properties`, is injected as a string to the `utilPropFile` attribute of `UtilLogPropGBean`.

In this section, we will deploy our gbean in Community Edition. When this gbean is deployed, all the applications that use `java.util.logging` API will use this configuration. Follow these steps to deploy the gbean:

1. [Download](#) the zip file and unzip the gbean (`UtilLogPropGBean.jar`), the deployment plan (`UtilLogPropGBean.xml`) and the sample `java.util.logging` configuration properties file (`UtilLogging.properties`).
1. This jar file contains `UtilLogPropGBean.java`. The `UtilLogging.properties` file defines a `ConsoleHandler` and a `FileHandler`. The `FileHandler` logs the messages to the `C:/temp/applevellogging/UtilLogging/java.log` file. The log levels of `FileHandler` and `ConsoleHandler` are `FINER` and `CONFIG` respectively. The log level for `com.ibm.sample.EMPDemo` is set to `SEVERE` at the bottom of the file. Of course, this value is overridden in the `EMPDemo` servlet. You can modify these values including the location of the log file according to your needs.
1. Upload the `UtilLogPropGBean.jar` to the Community Edition server

repository with the moduleid configuration as GBeans/UtilLoggingCustom/1.0/jar. This [link](#) explains how to upload the Java libraries into the repository.

1. Open a command prompt and change the directory to `<wasce_home>/bin`. Deploy the GBean using the deploy command. `<wasce_home>/bin>deploy -user system -password manager deploy UtilLogPropGBean.xml`
1. The gbean reads the `UtilLogging.properties` file and configures the `java.util.logging` system. Access the EMPDemo servlet by pointing your browser to this URL <http://localhost:8080/EMPDemo-UtilLogging/EMPDemo> (deployed in the previous section).

Listing 8 shows the messages from the server console:

Listing 8. Server console messages

```
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO:
  Created the PrintWriter on the Response object
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO:
  Got Initial context:
  org.apache.xbean.naming.context.ImmutableContext$NestedImmutableContext@16801680
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO:
  Got DataSource: org.tranql.connector.jdbc.DataSource@a000a00
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO:
  Got Connection: org.tranql.connector.jdbc.ConnectionHandle@1b3a1b3a
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO:
  Got Statement : org.tranql.connector.jdbc.StatementHandle@3b6e3b6e

Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: Table EMP after SELECT:
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: 1 PHANI SSE
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: NIKHIL 100
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: 2 JOE SSE
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: NIKHIL 100
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: 3 JOHN SSE
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: BOB 200
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost SEVERE: my severe message
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost WARNING: my warning message
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost INFO: my info message
Jan 2, 2009 11:11:46 AM com.ibm.sample.EMPDemo doPost CONFIG: my config message
```

The ConsoleHandler's log level has been set to CONFIG in the `UtilLogging.properties` file. Hence, we see all the messages logged at log level CONFIG or above (CONFIG, INFO, WARNING and SEVERE).

In addition to the output in Listing 5, the log file, Community Edition also creates `C:/temp/applevellogging/UtilLogging/java.log`. Because we have configured XMLFormatter for FileHandler, this file has the messages logged in XML format. Also, the log level has been set to FINER for FileHandler. Hence, in the file we see all the messages logged at FINER or above, except for FINEST.

Section 4. Community Edition nuances and getting the most from Log4j

[Log4j](#) is an open source library for logging services from the [Apache Software Foundation](#). Log4j is widely used in the open source community, including some big names like Apache Geronimo, JBoss, etc. Log4j's architecture revolves around three main concepts: *Loggers*, *Appenders*, and *Layouts*.

Applications first call Logger objects to initiate the logging of a message. When given a message to log, loggers generate LogEvent objects to wrap the given message. The loggers then pass on the LogEvents objects to their associated Appenders. The Appenders send the information contained by the LogEvents to specified output destinations. For example, a ConsoleAppender writes the information to System.out, and a FileAppender writes it to a log file. Before sending the information to the destination, the Appenders can use Layouts to create a text representation of the information in a desired format. For example, Appenders can use XMLLayout that formats the log messages as XML strings.

The LoggingEvents are assigned a level that indicates its priority. The default levels are (from highest to lowest):

- OFF
- FATAL
- ERROR,
- WARN
- INFO
- DEBUG
- ALL

Loggers and Appenders are assigned a level. If a LogEvent has the log level as `WARN` and the Appender's log level is `ERROR`, the Appender will not write the LogEvent. In this way, you can control the amount of log output.

All Loggers in Log4j have a name. Log4j organizes Logger instances in a hierarchical, tree structure, according to the names, as in the packaging namespace in the Java language. The Log4j documentation says

"A Logger is said to be an ancestor of another Logger if its name followed by a dot is a prefix of the descendant Logger name. A Logger is said to be a parent of a child Logger if there are no ancestors between itself and the descendant Logger".

For example, a Logger named `com.ibm` is said to be the child of the `com` Logger. The `com.ibm.wasce` Logger is the child of the `com.ibm` Logger and the grandchild of the `com` Logger. If a Logger is not explicitly assigned a level, it uses the level of its closest ancestor that has been assigned a level. Loggers inherit Appenders from their ancestors, although they can also be configured to use only Appenders that are directly assigned to them.

You configure all the Loggers, Appenders and Layouts in the `log4j.properties` or `log4j.xml` file. Log4j libraries first look for `log4j.xml` file and then `log4j.properties` file in the classpath to configure the logging service.

From the above discussion, you might think that Log4j is similar to `java.util.logging`. Of course, they are conceptually the same, but Log4j can do more than `java.util`. Handlers in `java.util.logging` perform the same tasks as Appenders in Log4j. The Formatters in `java.util.logging` perform the same task as Layouts in Log4j. However, `java.util.logging` has only four Handlers, whereas Log4j has a dozen Appenders. In addition, Log4j offers a rich set of Layouts where `java.util.logging` offers only SimpleFormatter and XMLFormatter. Some of the Appenders available in Log4j are:

- **FileAppender:** Appends log events to a file.
- **RollingFileAppender:** Extends FileAppender to backup the log files when they reach a certain size.
- **ConsoleAppender:** Appends log events to System.out or System.err using a layout specified by the user. The default target is System.out
- **SocketAppender:** Sends LoggingEvent objects to a remote a log server, usually a SocketNode.
- **JMSAppender:** A simple Appender that publishes events to a JMS Topic. The events are serialized and transmitted as the JMS message type ObjectMessage.
- **NTEventLogAppender:** Append to the NT event log system.

Log4j includes XMLLayout, SimpleLayout, TTCCLayout, HTMLLayout. For descriptions of various Appenders and Layouts, see the [Log4j](#) documentation.

In this section, we show how to use Log4j in applications deployed on Community Edition. Community Edition, by default, uses the following log4j configuration files:

- `<wasce_home>/var/log/server-log4j.properties`: This file

configures Appenders and Layouts for the server components as well as applications deployed on the server. This file configures a ConsoleAppender and a RollingFileAppender. The ConsoleAppender logs to `System.out` and the RollingFileAppender logs to the `<wasce_home>/var/log/server.log` file. The default logging is WARN. It also overrides log levels for various Loggers in the properties file.

- `<wasce_home>/var/log/deployer-log4j.properties`: This file configures logging service for the deployer. The deployer component is invoked when any applications are deployed using the command line deployer.
- `<wasce_home>/var/log/client-log4j.properties`: This file configures logging service for a Java EE application client.

In this section, we will configure Log4j in the following ways

1. Using the default server Log4j configuration
2. Setting up Log4j at the application level

Using the default server Log4j configuration

As described earlier, server components as well as applications deployed on Community Edition use the configuration in `<wasce_home>/var/log/server-log4j.properties`. We will use the modified EMPDemo sample that has Log4j for logging. It also uses the default configuration available in Community Edition. The sample contains the same EMPDemo servlet modified for logging using Log4j API. Listing 9 shows the modified EMPDemo, with the logging statements marked in **bold**. The servlet obtains the Logger and sets the log level to ALL.

Listing 9. Modified EMPDemo using the Log4j API

```
Logger logger = Logger.getLogger(EMPDemo.class.getName());
logger.setLevel(Level.ALL);
Connection con = null;
Statement stmt = null;

PrintWriter out = response.getWriter();

logger.info("Created the PrintWriter on the Response object");

try {
    Context initContext = new InitialContext();
    Context envContext = (Context)initContext.lookup("java:comp/env");
logger.info("Got environment context: " +envContext);
    DataSource ds = (DataSource)envContext.lookup("jdbc/DataSource");
logger.info("Got DataSource: " +ds.toString());
    con = ds.getConnection();
```

```

logger.info("Got Connection: " +con.toString() +"\n");

stmt = con.createStatement();
logger.info("Created the statement: " +stmt);
ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
logger.info("Gto the result set: " +rs);
logger.info("Table EMP after SELECT:");

out.println("Your EMP table contains the following entries:<BR>");

out.println("<table>");
out.println("<tr>");
out.println("<th>Empno</th>");
out.println("<th>Name</th>");
out.println("<th>Job</th>");
out.println("<th>Manager</th>");
out.println("<th>Salary</th>");
out.println("<th>Commission</th>");
out.println("<th>Deptno</th>");
out.println("</tr>");

while (rs.next()) {
    String emp = rs.getString("EMPNO");
    String name = rs.getString("ENAME");
    String job = rs.getString("JOB");
    String mgr = rs.getString("MGR");
    String sal = rs.getString("SAL");
    String comm = rs.getString("COMM");
    String dept = rs.getString("DEPTNO");

    out.println("<tr>");
    out.println("<td>"+emp+"</td>");
    out.println("<td>"+name+"</td>");
    out.println("<td>"+job+"</td>");
    out.println("<td>"+mgr+"</td>");
    out.println("<td>"+sal+"</td>");
    out.println("<td>"+comm+"</td>");
    out.println("<td>"+dept+"</td>");
    out.println("</tr>");

    logger.info(emp + "    " + name + "    " + job);
    logger.info("    " + mgr + "    " + dept);

out.println("</table>");

rs.close();
stmt.close();
con.close();

logger.debug("Debug");
logger.info("Info");
logger.warn("Warn");
logger.error("Error");
logger.fatal("Fatal");
}
catch(java.lang.Exception e) {

    e.printStackTrace();
    logger.fatal(e.getClass().getName());
    logger.fatal(e.getMessage());
}
}

```

Follow these steps to deploy and run the application:

1. From the sample [download](#), unzip the EMPDemo sample into a directory. The war file name is EMPdemo-Log4jLogging1.war.
2. Deploy the application.

```
<wasce_home>/bin>deploy --user system --password
manager deploy Log4jLogging1.war.
```
3. Access the EMPDemo servlet by pointing your browser to <http://localhost:8080/EMPdemo-log4j1/EMPDemo>. The servlet shows the rows from the employee table on the browser window.

Observe the console window to note that only the log messages shown in Listing 10 are logged. This is because the ConsoleAppender's log level is `WARN` by default (in the `server-log4j.properties`). So, the log messages whose log level is `WARN` or above are logged in the console.

Listing 10. Log messages in the console

```
14:13:31,687 WARN [EMPDemo] Warn
14:13:31,687 ERROR [EMPDemo] Error
14:13:31,687 FATAL [EMPDemo] Fatal
```

Open the `<wasce_home>/var/log/server.log` file to observe that all the log messages have been logged. This is because the log level of the FileAppender is `TRACE` by default. Hence it has logged all the messages. Listing 11 shows the logged messages:

Listing 11. server.log file

```
14:13:31,687 INFO [EMPDemo]
Created the PrintWriter on the Response object
14:13:31,687 INFO [EMPDemo]
Got environment context:
org.apache.xbean.naming.context.ImmutableContext$NestedImmutableContext@5b9e5b9e
14:13:31,687 INFO [EMPDemo]
Got DataSource: org.tranql.connector.jdbc.DataSource@8660866
14:13:31,687 INFO [EMPDemo]
Got Connection: org.tranql.connector.jdbc.ConnectionHandle@a1e0a1e
14:13:31,687 INFO [EMPDemo]
Created the statement: org.tranql.connector.jdbc.StatementHandle@c980c98
14:13:31,687 INFO [EMPDemo]
Got the result set: org.tranql.connector.jdbc.ResultSetHandle@11221122

14:13:31,687 INFO [EMPDemo] Table EMP after SELECT:
14:13:31,687 INFO [EMPDemo] 1 PHANI SSE
14:13:31,687 INFO [EMPDemo] NIKHIL 100
14:13:31,687 INFO [EMPDemo] 2 JOE SSE
14:13:31,687 INFO [EMPDemo] NIKHIL 100
14:13:31,687 INFO [EMPDemo] 3 JOHN SSE
14:13:31,687 INFO [EMPDemo] BOB 200
14:13:31,687 DEBUG [EMPDemo] Debug
14:13:31,687 INFO [EMPDemo] Info
14:13:31,687 WARN [EMPDemo] Warn
14:13:31,687 ERROR [EMPDemo] Error
14:13:31,687 FATAL [EMPDemo] Fatal
```

Setting up Log4j at the application level

Sometimes you might want to configure Log4j at the application level, ignoring the server-level configuration. You can do this by packaging the Log4j libraries and `log4j.properties` configuration with the application itself, and deploying the application. However, there is a hitch here -- Community Edition uses a “parent first” classloader policy. This means, if the classes are available in parent classloader, they will be loaded from the parent classloader. Since the Log4j is used by server components and is loaded by a classloader (which is higher in the classloader hierarchy) Log4j configured at the server level is always used even if Log4j libraries and `log4j.properties` file are packaged with the application.

To resolve this issue, we need to hide Log4j from the parent classloader and make it always load from the application classloader. To hide Log4j, you specify `<hidden-classes>` for the Log4j classes in the Community Edition application deployment plan. For information about other various classloader policies available in Community Edition, see this [technote](#).

This section illustrates how to configure Log4j at the application level. We use the same sample as in the previous section. However, we modify the Community Edition application deployment plan (`geronimo-web.xml`) to hide the Log4j classes. Listing 12 shows the modified Geronimo deployment plan, with the relevant portion of the plan marked in **bold**.

Listing 12. Modified `geronimo-web.xml` deployment plan

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<web:web-app xmlns:dep="http://geronimo.apache.org/xml/ns/deployment-1.2"
  xmlns:conn="http://geronimo.apache.org/xml/ns/j2ee/connector-1.2"
  xmlns:name="http://geronimo.apache.org/xml/ns/naming-1.2"
  xmlns:ejb="http://openejb.apache.org/xml/ns/openejb-jar-2.2"
  xmlns:pkgen="http://openejb.apache.org/xml/ns/pkgen-2.1"
  xmlns:app="http://geronimo.apache.org/xml/ns/j2ee/application-2.0"
  xmlns:sec="http://geronimo.apache.org/xml/ns/security-2.0"
  xmlns:web="http://geronimo.apache.org/xml/ns/j2ee/web-2.0.1"
  xmlns:pers="http://java.sun.com/xml/ns/persistence"
  xmlns:client="http://geronimo.apache.org/xml/ns/j2ee/application-client-2.0">
  <dep:environment>
    <dep:moduleId>
      <dep:groupId>com.ibm.wasce.samples</dep:groupId>
      <dep:artifactId>EMPdemo-log4j-2</dep:artifactId>
      <dep:version>2.1.0.0</dep:version>
      <dep:type>war</dep:type>
    </dep:moduleId>
    <dependencies>
      <dependency>
        <groupId>console.dbpool</groupId>
        <artifactId>EMPLOYEE_DS</artifactId>
      </dependency>
    </dependencies>
    <hidden-classes>
      <filter>org.apache.log4j</filter>
    </hidden-classes>
  </dep:environment>
  <web:context-root>/EMPdemo-log4j2</web:context-root>
  <name:resource-ref>
```

```

        <name:ref-name>jdbc/DataSource</name:ref-name>
        <name:resource-link>EMPLOYEE_DS</name:resource-link>
    </name:resource-ref>
</web:web-app>

```

Also, we copy the `log4j.properties` file into the `WEB-INF/classes` directory and copy `log4j-1.2.14.jar` in to the `WEB-INF/lib` directory of the `EMPDemo` application. Listing 13 shows the `log4j.properties` file:

Listing 13. The `log4j.properties` file

```

log4j.logger.com.ibm.sample=debug,applog

log4j.appender.applog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.applog.File=C:/temp/applevellogging/Log4J/applog1.log
log4j.appender.applog.layout=org.apache.log4j.PatternLayout

log4j.appender.applog.layout.ConversionPattern=%d{ABSOLUTE} %-5p [%c{1}] %m%n

```

This properties file defines a Logger for the servlet, and sets the Log level to `DEBUG`. But the servlet overrides this value to make it `ALL`. It also defines a `DailyRollingFileAppender` that logs messages to the `C:/temp/applevellogging/Log4J/applog1.log` file. The Layout used by the Appender is the standard Community Edition layout used in the `server-log4j.properties` file. Follow these steps to deploy the application.

1. From the sample [download](#), unzip the modified Web application. The WAR file name is `EMPDemo-Log4jLogging2.war`.
2. Deploy the application.

```
<wasce_home>/bin>deploy --user system --password
manager deploy EMPdemo-Log4jLogging2.war
```
3. Access the `EMPDemo` servlet by pointing a browser to <http://localhost:8080/EMPDemo-log4j2/EMPDemo>. The servlet shows the rows from the employee table in the browser.

Note that no messages have been logged in server console or in the `server.log` file, because we have overridden the server level Log4j configuration and configured it at the application level. Moreover, the application creates the `C:/temp/applevellogging/Log4J/applog1.log` file and logs the messages in this file. Listing 14 shows the messages logged in the file:

Listing 14. Messages in the `applog1.log` file

```

15:18:36,078 INFO [EMPDemo]
Created the PrintWriter on the Response object
15:18:36,093 INFO [EMPDemo]
Got environment context:
org.apache.xbean.naming.context.ImmutableContext$NestedImmutableContext@d760d76

```

```
15:18:36,093 INFO [EMPDemo]
  Got DataSource: org.tranql.connector.jdbc.DataSource@32a232a2
15:18:36,109 INFO [EMPDemo]
  Got Connection: org.tranql.connector.jdbc.ConnectionHandle@61e461e4
15:18:36,109 INFO [EMPDemo]
  Created the statement: org.tranql.connector.jdbc.StatementHandle@67206720
15:18:36,109 INFO [EMPDemo]
  Got the result set: org.tranql.connector.jdbc.ResultSetHandle@6f9a6f9a

15:18:36,109 INFO [EMPDemo] Table EMP after SELECT:
15:18:36,109 INFO [EMPDemo] 1 PHANI SSE
15:18:36,109 INFO [EMPDemo] NIKHIL 100
15:18:36,109 INFO [EMPDemo] 2 JOE SSE
15:18:36,109 INFO [EMPDemo] NIKHIL 100
15:18:36,109 INFO [EMPDemo] 3 JOHN SSE
15:18:36,109 INFO [EMPDemo] BOB 200
15:18:36,109 DEBUG [EMPDemo] Debug
15:18:36,109 INFO [EMPDemo] Info
15:18:36,109 WARN [EMPDemo] Warn
15:18:36,109 ERROR [EMPDemo] Error
15:18:36,109 FATAL [EMPDemo] Fatal
```

Section 5. Using SLF4J in Community Edition

The Simple Logging Facade for Java ([SLF4J](#)) serves as a simple facade for various logging APIs, which lets you plug in the desired implementation at deployment time. SLF4j does not invent another logging framework, but lets application use a standard API and plug in the actual logging implementation at the deployment time.

SLF4j supports multiple logging systems, such as, NOP, Simple, Log4j version 1.2, java.util.logging, JCL and logback. The SLF4j distribution ships with several JAR files:

- `slf4j-nop.jar`
- `slf4j-simple.jar`
- `slf4j-log4j12.jar`
- `slf4j-log4j13.jar`
- `slf4j-jdk14.jar`
- `slf4j-jcl.jar`.

Each of these jar files is hardwired at compile-time to use just one implementation. All of the bindings shipped with SLF4j depend on `slf4j-api.jar`, which must be present on the class path for the binding to function properly. It also provides a migration path for applications using concrete logging implementations using [bridge](#)

[modules](#). Community Edition ships with these libraries that you can use in applications:

- slf4j-api-1.4.3.jar
- slf4j-log4j12-1.4.3.jar
- jcl104-over-slf4j-1.4.3.jar

In this section, we show the use of SLF4j in the following ways.

1. Configure SLF4j to use the server Log4j configuration
2. Configure SLF4j using Log4j at the application level

Configure SLF4j to use the server Log4j configuration

In this section, we use the SLF4j API in the EMPDemo servlet instead of the Log4j API. However, it uses Log4j implementation configured at the server level for concrete implementation. Listing 15 shows the servlet code:

Listing 15. EMPDemo servlet using the SLF4j API

```
Logger logger = LoggerFactory.getLogger(EMPDemo.class.getName());
Connection con = null;
Statement stmt = null;

PrintWriter out = response.getWriter();

logger.info("Created the PrintWriter on the Response object");

try {
    Context initContext = new InitialContext();
    Context envContext = (Context)initContext.lookup("java:comp/env");
    logger.info("Got environment context: "+envContext);
    DataSource ds = (DataSource)envContext.lookup("jdbc/DataSource");
    logger.info("Got DataSource: "+ds.toString());
    con = ds.getConnection();
    logger.info("Got Connection: "+con.toString() +"\n");
    stmt = con.createStatement();
    logger.info("Created the statement: " +stmt);
    ResultSet rs = stmt.executeQuery("SELECT * FROM EMPLOYEE");
    logger.info("Gto the result set: " +rs);
    logger.info("Table EMP after SELECT:");

    out.println("Your EMP table contains the following entries:<BR>");

    out.println("<table>");
    out.println("<tr>");
    out.println("<th>Empno</th>");
    out.println("<th>Name</th>");
    out.println("<th>Job</th>");
    out.println("<th>Manager</th>");
    out.println("<th>Salary</th>");
    out.println("<th>Commission</th>");
```

```

out.println("<th>Deptno</th>");
out.println("</tr>");

while (rs.next()) {
    String emp = rs.getString("EMPNO");
    String name = rs.getString("ENAME");
    String job = rs.getString("JOB");
    String mgr = rs.getString("MGR");
    String sal = rs.getString("SAL");
    String comm = rs.getString("COMM");
    String dept = rs.getString("DEPTNO");

    out.println("<tr>");
    out.println("<td>"+emp+"</td>");
    out.println("<td>"+name+"</td>");
    out.println("<td>"+job+"</td>");
    out.println("<td>"+mgr+"</td>");
    out.println("<td>"+sal+"</td>");
    out.println("<td>"+comm+"</td>");
    out.println("<td>"+dept+"</td>");
    out.println("</tr>");

    logger.info(emp + " " + name + " " + job);
    logger.info(" " + mgr + " " + dept);
}
out.println("</table>");

rs.close();
stmt.close();
con.close();

logger.trace("Trace");
logger.debug("Debug");
logger.info("Info");
logger.warn("Warn");
logger.error("Error");
}
catch(java.lang.Exception e) {
    e.printStackTrace();
    logger.error(e.getClass().getName());
    logger.error(e.getMessage());
}
}

```

Listing 16 shows the dependencies declared for SLF4j in the Community Edition application deployment plan (`geronimo-web.xml`).

Listing 16. The `geronimo-web.xml` file with dependencies for SLF4j

```

<dependencies>
    .....
    .....

<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.4.3</version>
  <type>jar</type>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version>1.4.3</version>

```

```
<type>jar</type>
</dependency>
</dependencies>
```

Follow these steps to deploy and run the application:

1. From the sample [download](#), unzip the application file. The war file is `Log4j-SLF4j-WEB.war`.
2. Deploy the application.

```
<wasce_home>/bin>deploy --user system --password
manager deploy Log4j-SLF4j-WEB.war
```
3. Access the EMPDemo servlet at <http://localhost:8080/Log4j-SLF4j-WEB/EMPDemo>.

Listing 17 shows the log messages that display on the server console:

Listing 17. Server console messages

```
10:25:18,593 WARN [EMPDemo] Warn
10:25:18,593 ERROR [EMPDemo] Error
```

These messages display because; the ConsoleAppender's log level is set to `WARN` in the `server-log4j.properties` file. Hence, only log messages with log level `WARN` or above are logged on the console. However, you can see that all the log messages are logged in the `server.log` file.

Configure SLF4j using Log4j at the application level

In this section, we configure SLF4j to use the Log4j configured at the application level for the EMPDemo servlet. The servlet code in Listing 13 does not change. However, the application ships with a `log4j.properties` file that replaces Log4j configuration at the server level. The EMPDemo servlet uses SLF4j API to log the messages. Listing 18 shows the `log4j.properties` file:

Listing 18. The log4j.properties file

```
log4j.logger.com.ibm.sample=debug,applog

log4j.appender.applog=org.apache.log4j.DailyRollingFileAppender
log4j.appender.applog.File=C:/temp/applevellogging/SLF4j/java.log
log4j.appender.applog.layout=org.apache.log4j.PatternLayout

log4j.appender.applog.layout.ConversionPattern=%d{ABSOLUTE} %-5p [%c{1}] %m%n
```

This `log4j.properties` file configures the Log4j service to use

DailyRollingFileAppender to log messages to C:/temp/applevellogging/SLF4j/java.log file. It uses the Community Edition format used to log messages.

Listing 19 shows the relevant portion of the Community Edition application deployment plan (geronimo-web.xml) for the application:

Listing 19. Community Edition application deployment plan

```
<dep:environment>
  .....
  .....
  <dependencies>
    .....
    .....
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.4.3</version>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.4.3</version>
      <type>jar</type>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
      <type>jar</type>
    </dependency>
  </dependencies>
  <hidden-classes>
    <filter>org.apache.log4j</filter>
  </hidden-classes>
</dep:environment>
```

The deployment plan declares a dependency on the Log4j and SLF4j libraries, and hides the Log4j package from the parent classloader. Follow these steps to deploy the application.

1. Download the application and save it into a directory. The WAR file name is Log4j-SLF4j-AppLevel-WEB.war.
2. Deploy the application.

```
<wasce_home>/bin>deploy -user system -password manager
deploy Log4j-SLF4j-AppLevel-WEB.war.
```
3. Access the EMPDemo servlet at <http://localhost:8080/Log4j-SLF4j-WEB/EMPDemo>.

You can see that no messages are logged to either the server console or the

server.log file. However, the application creates the C:/temp/applevellogging/SLF4j/java.log file and logs the messages as configured.

Section 6. Conclusion

We have shown how to use java.util.logging, Log4j and SLF4j in Community Edition in various ways. You have learned how to use gbeans to configure services at the server scope. You have also learned how classloader issues can become a problem when you need application-specific configurations for logging services. Finally, your applications might benefit from using SLF4j as a standard API for logging, to let you plug in the desired logging implementation at deployment time.

Section 7. Resources

- Learn more about Community Edition by exploring the [documentation](#).
- Visit the [Community Edition support page](#) for support offerings available from IBM.
- The developerWorks WebSphere Application Server [Community Edition zone](#) has many articles related to server administration and developing applications.
- Visit [Apache OpenJPA](#) to learn more about this open source implementation of JPA.
- Learn how to design applications using JPA from the article [Design enterprise applications with the EJB 3.0 Java Persistence API](#)
- Learn how to migrate a Hibernate application to JPA from the article [Migrating legacy Hibernate applications to OpenJPA and EJB 3.0](#)

Downloads

Description	Name	Size	Download method
Sample code	wasce_logging.zip	358KB	HTTP

[Information about download methods](#)

About the author

Phani Madgula

Phani is currently working for WebSphere Application Server Community Edition support at India Software Labs (ISL). He has been actively involved in many projects related to migrating applications from other application servers to WebSphere Application Server Community Edition. He has 5 years experience at IBM. He worked in various product teams including WebSphere Application Server Community Edition, WebSphere Business Integration Adapters and DB2. He has experience in developing JEE applications, product support, and database administration. He is an Oracle9i certified professional.