

Database Tutorials

Java Batch Job Framework

Author: Adym Lincoln, Java Batch Job Framework
Copyright © 2006-2013, Java Batch Job Framework Software, All Rights Reserved

PREAMBLE	3
GLOSSARY	3
REFERENCES.....	3
<u>DATABASE TUTORIALS.....</u>	<u>4</u>
PREREQUISITES	4
JBJF PROJECT SETUP.....	4
DATABASE SETUP	8
DATABASE MODEL	9
JAVA PACKAGES.....	10
CREATING BATCH JOB CLASS.....	11
<u>CREATING TASK CLASSES</u>	<u>14</u>
READTEXTFILE.....	15
RUNTASK()	16
LOADDELIMITEDDOCUMENT.....	20
INITTASK()	23
GETCONNECTION ()	25
RUNTASK ().....	27
SQLSELECTTASK	31
LOADRESULTSET	32
RUNTASK ().....	35
<u>CREATING THE JBJF BATCH DEFINITION FILE</u>	<u>38</u>
JBJF-PARAMETERS	38
JBJF-EMAIL	38
JBJF-DIRECTORIES	39
JBJF-TASKS.....	39
JBJF-CONNECTIONS	40
JBJF-PLUGINS.....	41
JBJF-LOGS	41
JBJF-SQL.....	42
LOG4J PROPERTIES.....	44
<u>RUNNING THE BATCH JOB.....</u>	<u>47</u>
LAUNCH FACILITY	47
CONSOLE CONFIGURATION.....	48

Preamble

This tutorial assumes that you've read through the JBJF User Guide and the JBJF Basics Tutorials. Because of this assumption, we won't spend a great deal of time discussing Project setup and JBJF architecture. In this tutorial we will focus on the following concepts:

- ✓ Database Processing
- ✓ Task Resource Tags
- ✓ SQL Definitions
- ✓ Plugin Definitions and Management
- ✓ Job Stack
- ✓ Encryption

Glossary

Name	Description/Comments
JBJF	A document acronym for Java Batch Job Framework.
XML	Industry standard for Extensible Markup Language. A simple language for adding structure to data and documents.
XML Definition	A coding paradigm that combines Java's programming language with XML configuration files.
JBJF Batch Definition File	A specialized XML file that contains data and elements specific to a JBJF batch process.

References

- Title: [jbjf-user-guide.pdf](#)
Location: <http://jbjf.sourceforge.net/pdfs/jbjf-user-guide.pdf>
Author: Adym S. Lincoln
Referenced Revision: 1.0.0
- Title: [jbjf-basics-tutorial.pdf](#)
Location: <http://jbjf.sourceforge.net/pdfs/jbjf-basics-tutorial.pdf>
Author: Adym S. Lincoln
Referenced Revision: 1.1.0, 8/1/2007

Database Tutorials

In this tutorial we'll be working beyond JBJF basics and begin to touch on JBJF Database techniques and how to use JBJF to run against various database engines.

Prerequisites

The Database tutorials will focus on using JBJF to read, write and process database information. We will build on the basics tutorials such as directories; email and logging, but there will be a focus on new aspects in database connections, sql, encryption and the job stack.

For this tutorial, I'll be using the following toolset:

- ✓ Windows 7
- ✓ Eclipse Galileo
- ✓ JDK 1.6.0-xx, Sun Microsystems mixed mode
- ✓ A source JBJF package from the SourceForge JBJF File Release repository. We'll need the source distribution in order to build and customize the JBJF to a specific database engine.
- ✓ A database engine, Oracle Express, MySQL, etc....

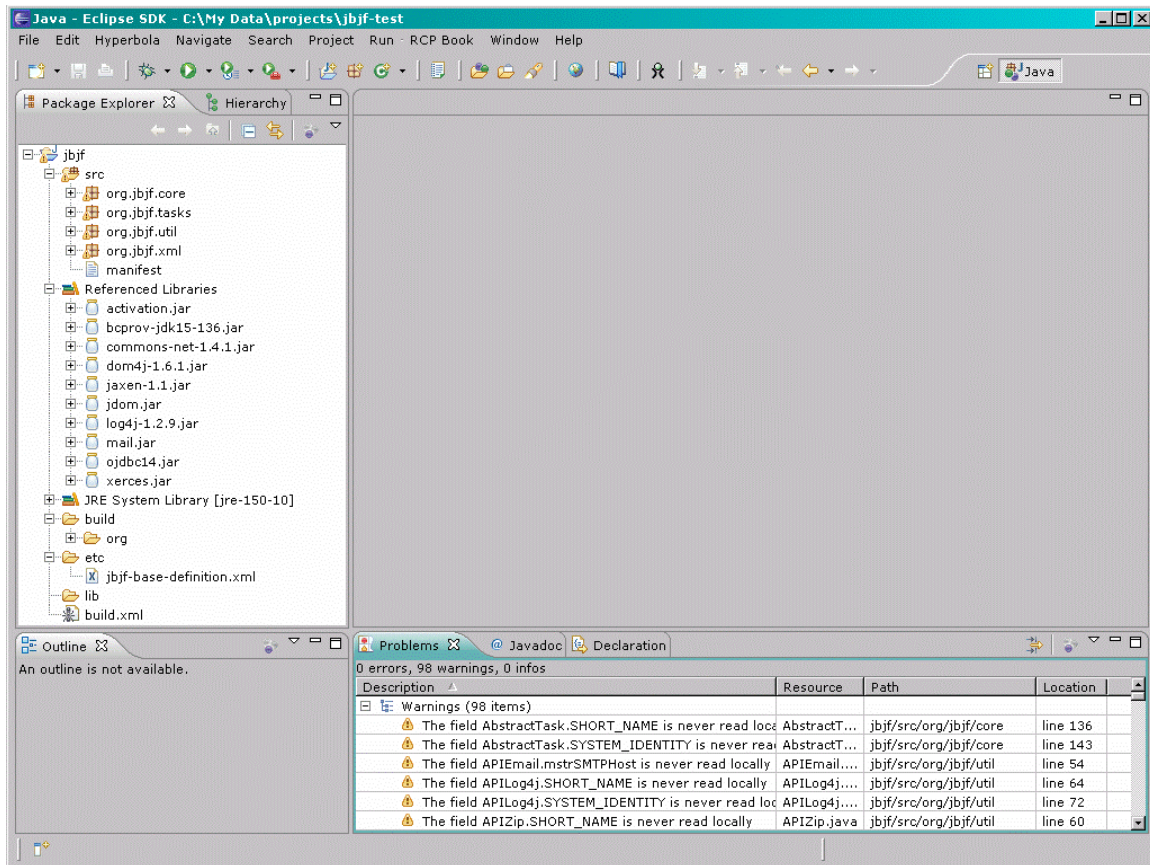
JBJF Project Setup

Open up Eclipse and/or create a new Eclipse workspace. Checkout the JBJF project or download a source package from the SourceForge repository. Unzip the source package into your Eclipse workspace. For the purposes of this tutorial I'll refer to this location as `${bjf-dir}`.

[JBJF Downloads Page - http://sourceforge.net/projects/bjbf/](http://sourceforge.net/projects/bjbf/)

Open up your Eclipse and point it to the workspace that contains the `${bjf-dir}`. For my examples, I'm using one Eclipse project to handle the JBJF source code and a separate project for the Tutorials.

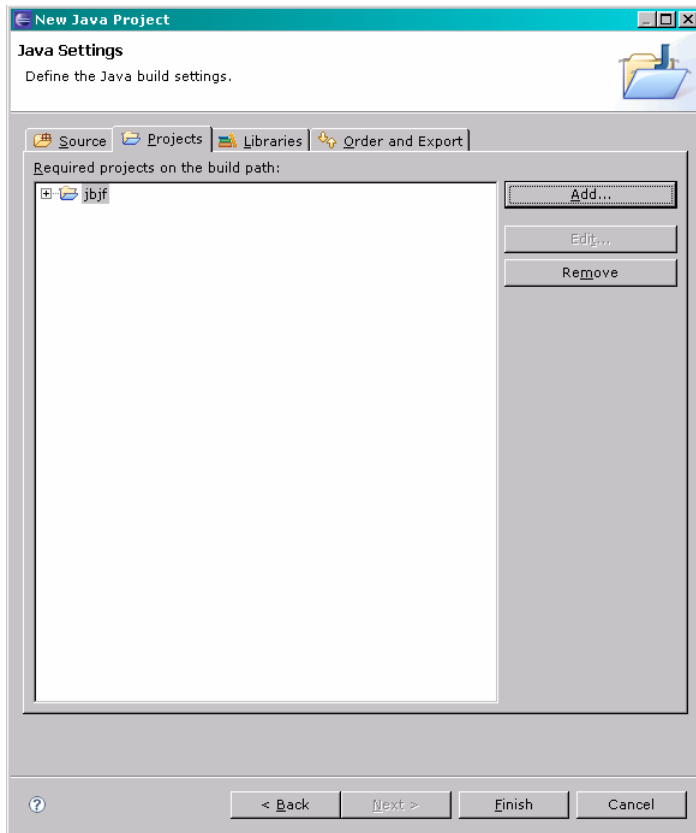
Anyway, the general goal for these Database tutorials will be to reveal how to use the JBJF to read a recordset onto the job stack. Pull that data off the JBJF job stack in another task and use the recordset to perform further tasks and work. Close the recordset and release any database and memory resources and finally gain an overall approach to JBJF database processing. While simple in nature, the tutorial will reveal the basic strategy that you can apply to your more complex batch processes.



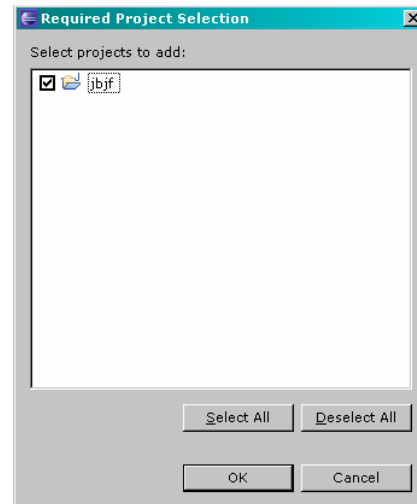
(Typical Eclipse workspace with JBJF source project)

Next, we need to create a tutorial project that we can build our batch job and tasks in. Project creation was covered in the basics tutorial, so refer to that document for details. For this tutorial, I'll name my tutorial project jbjf-database-tutorials.

Unlike the basics tutorial, this tutorial project will require a few different things. First, we need to add the JBJF project to the tutorials project. During the database tutorial Project creation, click the Projects Tab. Click the Add button and then checkmark the jbjf project. See figures 1.1 and 1.2.

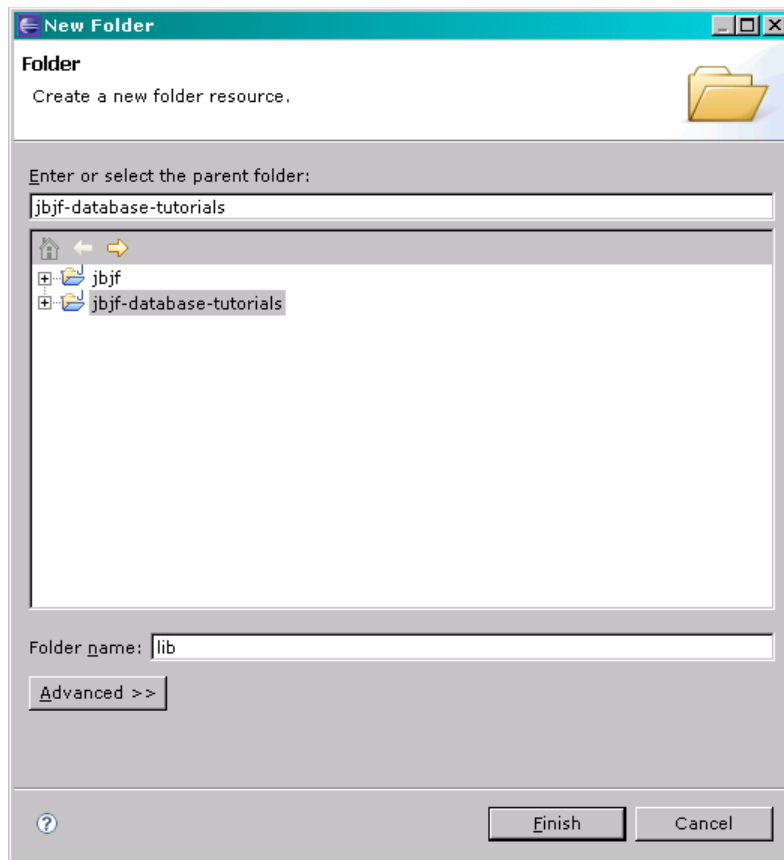


(Figure 1.1)



(Figure 1.2)

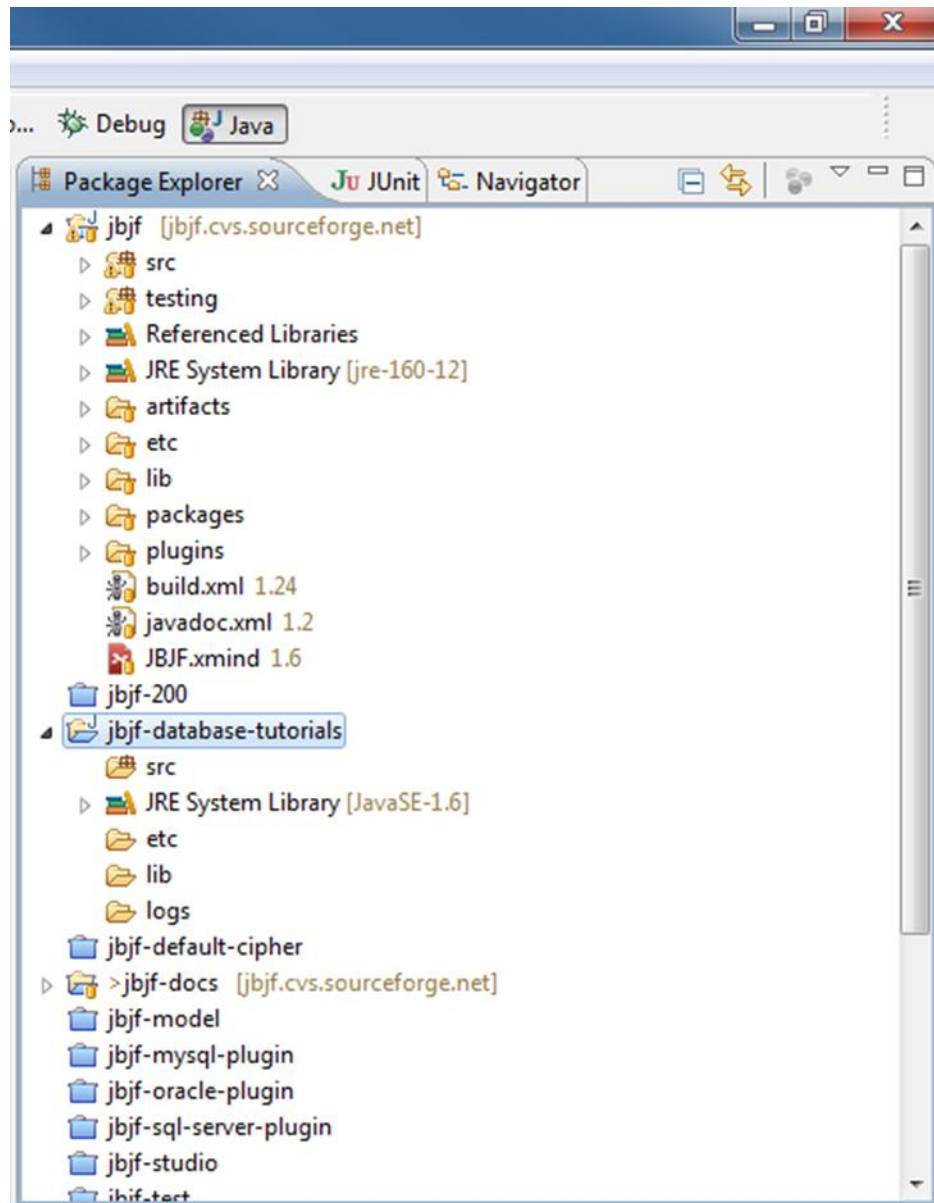
Click the Finish button to create the tutorial project. Once the project has been created, we also need to add the following sub-folders within the tutorial project. Again, details are covered in the basics tutorial document. For each sub-folder to be added, right click on the tutorial project and select New -> Folder. Then type in the name and click the Finish button.



Create the following sub-folders in the tutorial project:

- lib
- etc
- logs

This completes the project setup. Your new tutorial project should be similar to the following:



Database Setup

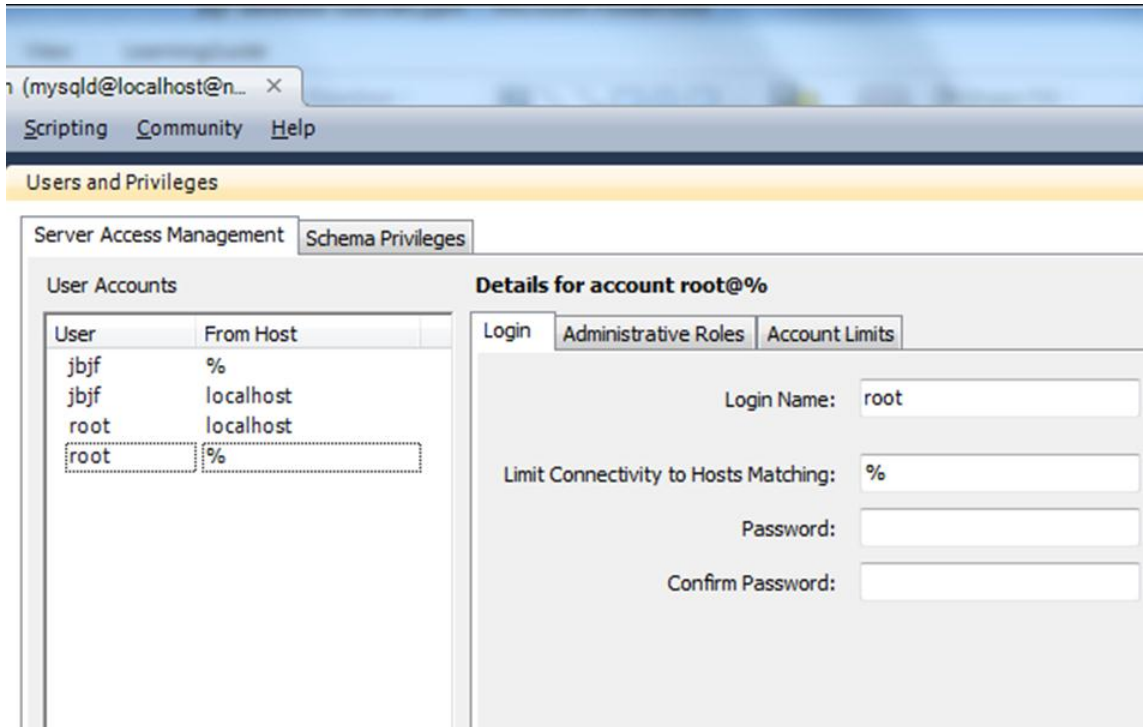
The next step is to get the database installed and working. Then take the example code from the tutorials and build a database. For this tutorial I'll be using MySQL along with the MySQL Workbench, both are free.

You can download MySQL from the Oracle website for free. There are ample instructions on how to install and get the database running and I won't reiterate those instructions here. You are also free to use a different database engine, Oracle Express and Microsoft's SQL Server Express are excellent freebies, but make sure you also download the JDBC drivers for that engine. You should be familiar with these other database engines as you'll need to alter the build scripts and be attentive to differences in the tutorial.

Once the database is installed and running, we need to setup a schema and user for the tutorial:

- 🔗 Open up the MySQL Workbench and connect to the localhost instance using root. Use the Open Connection to start Querying, not the Server Administration section.
- 🔗 In the Object Browser, create a new schema called jbjf...case sensitive.

- Now return to the Home/Start Page in the Workbench and create a new Server Instance under the Server Administration section.
- Create two new Users called jbjf, one for jbjf@% and one for jbjf@localhost:



- In the Workbench > Server Administration, Click on the Schema Privileges (Tab).
- In the Schema Privileges (Tab), Click on the jbjf user.
- In the Schema Privileges (Tab), Click on the Add Entry... button.
- In the New Schema Privilege Definition, select/enable the following options:
 - Any Host (%)
 - Selected schema: Click the jbjf schema.
 - Click the OK button.
- Back in the Schema Privileges Tab, checkbox the following options:
 - Object Rights, All of them.
 - DDL Rights, All of them.
 - Other Rights, CREATE TEMPORARY TABLES.
- Save Changes

Database Model

With the database up and running and our new jbjf userid we are now ready to build the database model. Included in the tutorial code is a directory called database that contains a number of sub-directories and *.sql files. You are free to look thru this code, but our focus at this point is the build.sql file in the ./database directory.

- Open up a command prompt and change the directory to the location of the \${jbjf-database-tutorials}/database directory:


```
... \projects \jbjf \jbjf-database-tutorials > dir
Volume in drive C
Volume Serial Number is
```

Directory of ...\\projects\\jbjf\\jbjf-database-tutorials

```
10/12/2007  06:31p      <DIR>      .
10/12/2007  06:31p      <DIR>      ..
10/12/2007  06:22p                309 .classpath
10/12/2007  06:22p                399 .project
10/12/2007  06:24p      <DIR>      build
10/16/2007  08:11p      <DIR>      database
10/12/2007  06:23p      <DIR>      etc
10/12/2007  06:23p      <DIR>      lib
10/12/2007  06:23p      <DIR>      logs
10/12/2007  06:24p      <DIR>      src
                2 File(s)                708 bytes
                8 Dir(s)   3,464,982,528 bytes free
```

...\\projects\\jbjf\\jbjf-database-tutorials>cd database


...\\projects\\jbjf\\jbjf-database-tutorials\\database>

 Run the build.bat command script, it will run the mysql command and the build.sql script.

...\\projects\\jbjf\\jbjf-database-tutorials\\database>build.bat



```
Database.....[jbjf]
User.....[jbjf]
Host.....[localhost]
C:\\My Data\\projects\\jbjf\\jbjf-database-tutorials\\database
jbjf.orders: Records: 2 Deleted: 0 Skipped: 0 Warnings: 0
jbjf.items: Records: 4 Deleted: 0 Skipped: 0 Warnings: 0
```

...\\projects\\jbjf\\jbjf-database-tutorials\\database>

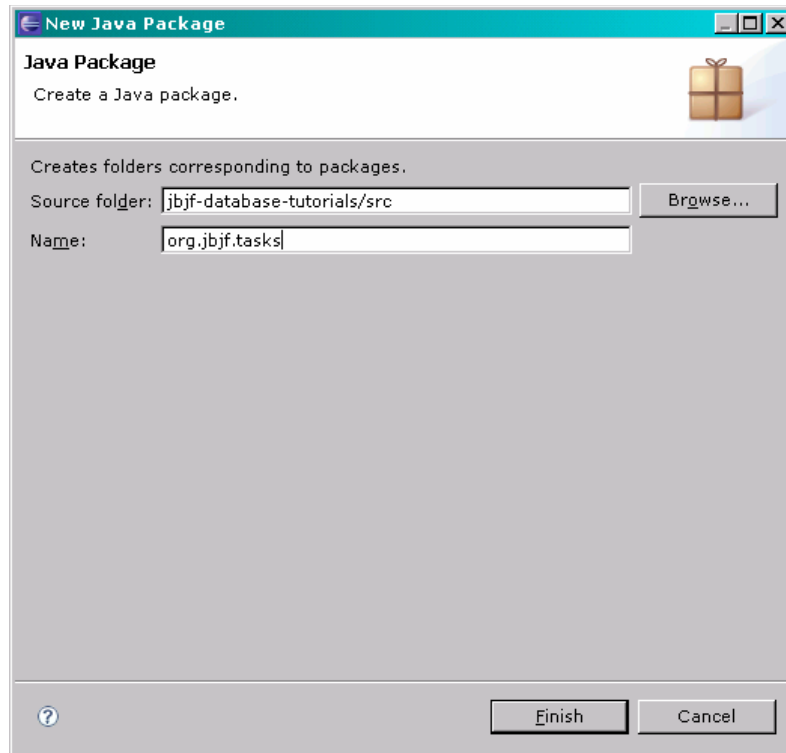
 Open up the MySQL Workbench and connect to the localhost instance. Validate that the jbjf schema now has tables and rows in each table.

Java Packages

Before we begin creating classes, we first need to setup a couple of Java packages to hold our classes. Essentially, we'll be creating Batch jobs (sub-classes of AbstractBatch) and Tasks (sub-classes of the AbstractTask class) within the tutorial project. For this, we'll create the following Java packages:

-  org.jbjf.tasks
-  org.jbjf.jobs

Within the tutorial project is a folder called src. You can expand this source folder to review the packages and their names. Right-click on the src folder and select New -> Package. Name the new package org.jbjf.tasks and click the Finish button to add the package.



Repeat the above step for each package:

🔗 `org.jbjf.jobs`

Creating Batch Job Class

Once the packages are created our first chore will be to create a batch job class. Start by right-clicking on the `org.jbjf.jobs` package and select **New > Class** from the pop-up menu. In the New Java Class dialog, set the following values:

Name : `FirstDatabaseJob`

Modifiers : `public`

Source folder: `${Tutorial Project}/src` (should be pre-filled)

Package : `org.jbjf.jobs` (should be pre-filled)

Superclass : `org.jbjf.core.AbstractBatch`

public static void main() : ☒

Constructors from superclass : ☒

Inherited abstract methods : ☒

Generate comments : `<your option>`

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?
☒ public static void main(String[] args)
☒ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments as configured in the [properties](#) of the current project?
☒ Generate comments

For the Superclass, you can use the Browse button and search for AbstractBatch or simply type in org.jbjf.core.AbstractBatch. Click the Finish button when you've set all the values. Eclipse generates a shell for the batch job, it should look similar to the following (comments removed for brevity):

```
package org.jbjf.jobs;

import org.jbjf.core.AbstractBatch;

public class FirstDatabaseJob extends AbstractBatch {

    /**
     * @param pstrName
     */
    public FirstDatabaseJob(String pstrName) {
        super(pstrName);
    }

    /**
```

```
    * @param args
    */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
    }
}
```

Essentially, there are two methods, the default constructor and the main() method. Given the simple nature of this tutorial, there is no need to touch the default constructor. The main() method however needs some code. The simplest and shortest implementation is to create an instance of the batch job subclass, passing the batch job name into the constructor. Then use that class instance to call into the _main() method, the clandestine entry point for a command line execution.

```
/**
 * @param args
 */
public static void main(String[] args) {
    // BATCH JOB INSTANCE : Create a class instance of the
    // batch job...
    //
    FirstDatabaseJob ltheJob = new FirstDatabaseJob (
        "jbjf-database-tutorial-001"
    );

    // ENTRY POINT : Call the clandestine entry point for the
    // batch job...
    //
    ltheJob._main ( args );
}
```

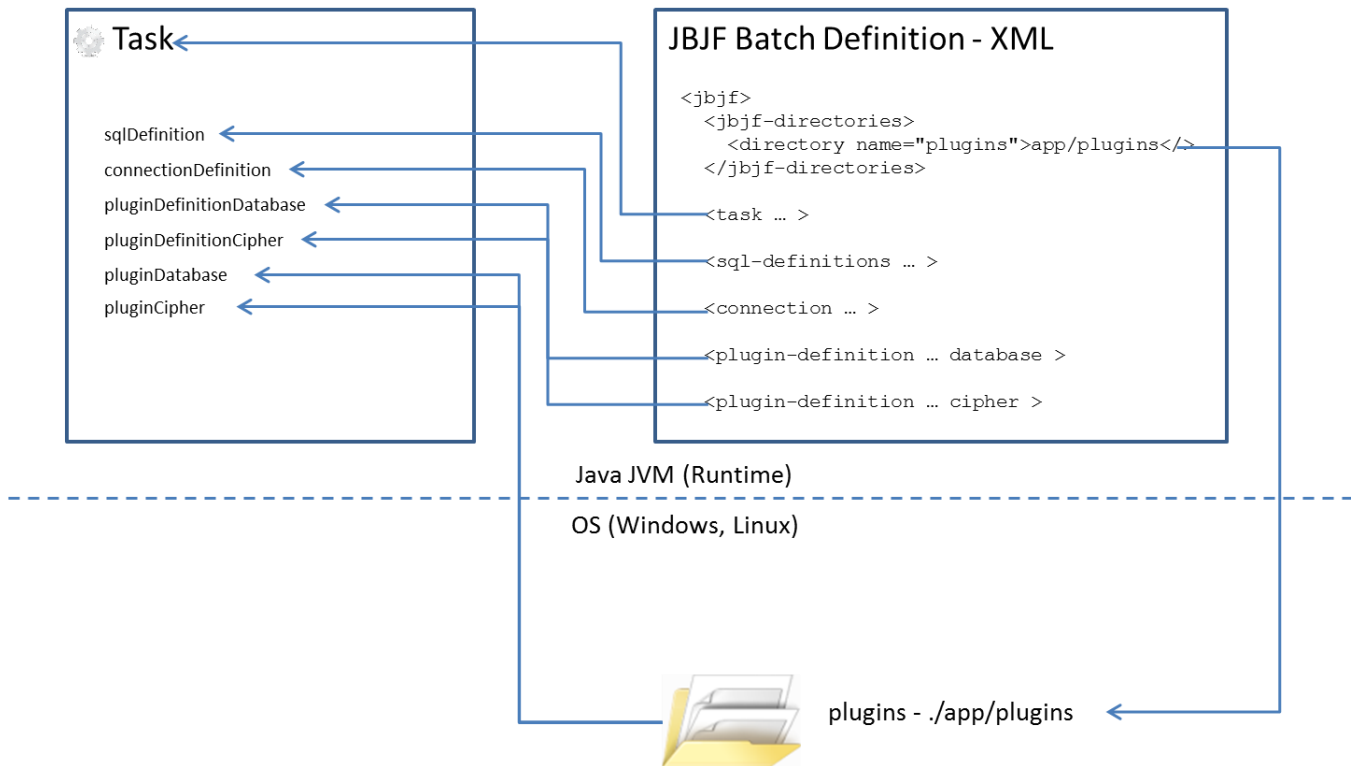
This completes the batch job class, you may review the entire class in the example source code. On to the tasks.

Creating Task Classes

When creating JBJF Database Task classes, a general rule of thumb is you really need four things:

- 1.) A Task class (Java)
- 2.) A connection definition to the database ... connection from the JBJF Batch Definition file.
- 3.) A database plugin for the specific database platform ... plugin-definition from the JBJF Batch Definition file.
- 4.) An SQL statement to run ... sql-definition from the JBJF Batch Definition file.

The following diagram illustrates where the Task gets the definitions and the objects it needs to carry out an SQL database task. One thing to note is the plugin-definitions and the Plugins require essentially two objects per plugin, one is the plugin-definition, and the other is the actual runtime plugin object.



Also, a second rule of thumb, JBJF Database Tasks should be tailored to run either a read transaction (SELECT) or an action transaction (INSERT, UPDATE, DELETE, and CALL).

The simplest strategy when writing JBJF Database Tasks is the following:

- 🔗 Extend the AbstractTask.
- 🔗 Implement a getConnection() method.
- 🔗 Use the getConnection() method to interface with and manage the JBJF Database Plugin for the task.
- 🔗 Call the getConnection() method from your runTask() method to get a JDBC connection.

For this first tutorial we'll be setting up a simple job that does the following steps:

- 🔗 Reads in the master data for an Order Entry application as a flat text file.
- 🔗 Loads that data into a database table.
- 🔗 Reads out certain columns of that master data into an SQL Resultset and stores that Resultset onto the Job Stack.
- 🔗 Pull the Resultset off the Job Stack and load the data into the Customers table.

While this may seem like a rather simple set of tasks, it will exercise the crucial goals of:

- 🔗 Loading a simple text file.

- 🔗 Loading that text file data into a database table.
- 🔗 Reading data from a database table and storing it on the Job Stack.
- 🔗 Pulling that Job Stack data and loading it into another table.

ReadTextFile

The first Task class is designed to read in our flat text file (master-data.txt) and store that object onto the Job Stack to be loaded later. Why don't we just read the file in and load it all in one task? The answer is you can. However, a foundational principle of JBJF is reusability. A task that simply reads in a text file is far more reusable than a Task that reads in a text file and loads it into an SQL database. Not every batch process reads in a text file and loads it into an SQL database, some may only need to read in a text file and do something else. Ironically, the less a single Task does, the more reusable it is.

That said, let's add the first task, ReadTextFile, into the org.jbjf.tasks package. Create a new class and name it ReadTextFile. Use the following diagram to match up the correct properties of the New Class dialog.

New Java Class

Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)
☒ Constructors from superclass
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))
☒ Generate comments

Click the Finish button when all the properties have been set. Eclipse will generate the following skeleton class, I've removed the comments for brevity:

```
package org.jbjf.tasks;

import java.util.HashMap;
import org.jbjf.core.AbstractTask;

/**
 *
 */

public class ReadTextFile extends AbstractTask {

    /**
     * Default constructor.
     */
    public ReadTextFile() {
        // TODO Auto-generated constructor stub
    }



    /* (non-Javadoc)
     * @see org.jbjf.core.AbstractTask#runTask(java.util.HashMap)
     */
    @Override
    public void runTask(HashMap pjobParameters) throws Exception {
        // TODO Auto-generated method stub
    }

}
```

The skeleton includes the basics from the AbstractTask, a default constructor and the runTask() method. The default constructor is "required", as it will be used when the JBJF framework creates the Task via Dynamic allocation. The runTask() method will contain our code that reads in the Text file from the project directory.





runTask()

For this task we will utilize the following Text File utility classes in the org.jbjf.core.file package:

-  DelimitedDocumentAdapter
-  DelimitedDocument

For details on these classes see the Javadocs. But basically we use them in the following manner, the DelimitedDocumentAdapter is designed to store an entire text document as a series of FlatRow objects. The Adapter allows us to traverse the document in different ways, including row by row and field by field.

So, the basic goal of this Task class will be to read in a text file as a DelimitedDocumentAdapter. Then store that object onto the Job Stack. Later, the object can be pulled off the Job Stack by another task. So what do we need to provide to that DelimiterDocumentAdapter to let it carry out its task?

-  A directory path to the file, either partial or full.
-  A filename.
-  A delimiter for the columns in the file.
-  Number of columns in the file.

These properties are passed into the Task class using the <resource> XML element in the JBJF Batch Definition file, in the <task> element. There is also one other optional property that is part of the DelimitedDocumentAdapter, a boolean header indicator that indicates whether the first line in the text file contains header (column) names. This resource (property) we'll label as header:

🔗 Does the text file contain header (column) names.

Once created, the DelimitedDocumentAdapter can then be packaged and placed onto the Job Stack for use in other Task classes. For this piece, we need another resource (property) like the following:

🔗 A Job Stack key, any String value will do, it just needs to be unique for the Batch process.

Again, these properties can be passed into the Task class via the <resource> XML element in the JBJF Batch Definition file.

At this point we've got enough details to begin coding the <task> element for the JBJF Batch Definition file. Normally, we wouldn't code the <task> element until the end, but I think it important that you understand how to pass properties from the JBJF Batch Definition file into a Task class.

That said, we've outlined the properties we need above, and below is an example <task> element that will provide all the needed elements for the Task:

```
<jbjf-tasks>
  <task name="t001" order="1" active="true">
    <class>[java.package.name].ReadTextFile</class>
    <resource type="path">./database/data</resource>
    <resource type="filename">master-data.txt</resource>
    <resource type="delimiter">master-data.txt</resource>
    <resource type="job-stack-key">master.data </resource>
  </task>
```

A couple of things to note on the <task> elements:

- 🔗 The <class> element has a stub () that we won't know until we've actually coded the Task class.
- 🔗 For the job-stack-key, don't get confused on what gets used to store the results/outcome of the Task class, in this case the DelimitedDocumentAdapter. The results will be stored on the Job Stack using master.data.

For each of the <resource> items we create a class property for it in the Java Task class. I typically set any optional properties to their default values. You can also initialize everything to null, and then set optional properties in the default constructor. Required properties are typically set to null:

```
/**
 * Class property that stores the delimiter.
 */
protected String delimiter = ",";

/**
 * Class property that stores the directory path.
 */
protected String path = ".";

/**
 * Class property that stores the filename.
 */
protected String filename = null;

/**
 * Class property that stores the Job Stack key that will be
 * used to store the return object of the task.
 */
```

```
protected String jobStackKey = "read.text.file";
```

For each of these properties I used Eclipse to Generate Getter and Setter methods in the class. I won't cover these for abatement reasons, you can view them in the source code. I've also implemented the Required Resources in this example class. Introduced in 1.3.x of JBJF, Required Resources provide a simple control in all JBJF Task classes to ensure that any given class has all the Required Resources to operate correctly. For our ReadTextFile class only the filename resource is required.

At this point we have all the supporting pieces of the Task class. So we now begin constructing the actual runTask() method. There are probably 10 different ways to write this piece of code, I'll present what I think is a good way using many Java and JBJF coding principles.

We start by creating a check to ensure that all required resources are available. This is a typical (and good) JBJF coding standard. It's also good practice to put log4j INFO output at the start and end of the runTask() method:

```
getLog().info( SHORT_NAME + "...Start..." );

/* CHECK : Make sure the task has all the required resources.
 */
if ( hasRequiredResources() ) {

}

getLog().info( SHORT_NAME + "...Complete..." );
```

All the remaining code will be placed within the if (hasRequiredResource()). This ensures that nothing runs unless the Required Resources are available. You can even put an else in there and throw a custom exception if you wish. Our first step is to get all the resources resolved into the task. We basically grab the String values from each resource and call the setter method to capture the value in the class property.

```
getLog().info( SHORT_NAME + "...Start..." );

/* CHECK : Make sure the task has all the required resources.
 */
if ( hasRequiredResources() ) {
    String lkeyHeaders = null;
    boolean lblnHeader = false;
    if ( getResources().containsKey(REZ_KEY_HEADER) ) {
        lkeyHeaders = (String)getResources().get( REZ_KEY_HEADER );
        lblnHeader = lkeyHeaders.contains("true");
    }

    if ( getResources().containsKey(REZ_KEY_PATH) ) {
        setPath ( (String)getResources ().get ( REZ_KEY_PATH ) );
    }

    if ( getResources().containsKey(REZ_KEY_FILE) ) {
        setFilename ( (String)getResources ().get ( REZ_KEY_FILE ) );
    }

    if ( getResources().containsKey(REZ_KEY_DELIMITER) ) {
        setDelimiter ( (String)getResources ().get ( REZ_KEY_DELIMITER
) );
    }

    if ( getResources().containsKey(REZ_KEY_JOBSTACK) ) {
```

```

        setJobStackKey ( (String)getResources ().get ( REZ_KEY_JOBSTACK
    ) );
}

```

Next, we initialize our results/outcome object for the Task, a DelimitedDocumentAdapter.

```

/* RESULTS : Initialize the results object.
*/
DelimitedDocumentAdapter    ldlmDocument = null;

```

I then dump all the class properties using log4j debug output, just for good measure. By default every JBJF Task comes with a log4j logger enabled...simply use the getLog() method to grab it. I also convert the delimiter String into a char in preparation for the DelimitedDocumentAdapter.

```

getLog().debug( "Text file name [" + getFilename() + "]" );
getLog().debug( "Text file source directory [" + getPath() + "]" );

/* CHAR : The DelimitedDocumentAdapter uses a single
 * character (char) as the delimiter, we must convert
 * the String property to a char.
 */
char    lchrDelimiter = getDelimiter().charAt( 0 );
getLog().debug( "Text file delimiter [" + lchrDelimiter + "]" );

getLog().debug( "Job Stack storage key [" + getJobStackKey() + "]"
);

```

Next, I process the optional resource header, if any. I purposely made this a local variable in the method to illustrate that not every resource element in the JBJF Batch Definition file has to be a Class property in the Task class. It's also a good example of how to check for an optional resource element.

```

String    lkeyHeaders = null;
boolean    lblnHeader = false;
if ( getResources().containsKey("header") ) {
    lkeyHeaders = (String)getResources().get( "header" );
    lblnHeader = lkeyHeaders.contains("true");
}
getLog().debug( "Job Stack headers [" + lkeyHeaders + "]" );

```

Next, we create the DelimitedDocumentAdapter for the text file, we make use of a mixture of Class property getter methods and the local variable for headers. The DelimitedDocumentAdapter will automatically link up to the text file and load the file.

```

ldlmDocument = new DelimitedDocumentAdapter (
    getPath() + File.separator + getFilename()
    ,lchrDelimiter
    ,lblnHeader
);

```

Finally, we push that object onto the Job Stack using the job stack key provided in the <resource> element.

```

getParameters().put(getJobStackKey(), ldlmDocument);

```

NOTE : In the final version of the code, you'll see the code within the Required Resources check is also wrapped in a try {} catch {} block that allows any errors/exceptions to be captured and a custom message constructed before throwing it up to the parent batch process.

With the code written for the Task, we now code the <task> element in the JBJF Batch Definition file. For purposes of brevity, I won't explain the entire construction of the JBJF Batch Definition file. This tutorial assumes you've worked through the basic tutorials and that you are familiar with XML. That said, I will focus on the <task> elements and supporting elements only.

The <task> element must provide the required <resource> elements at the very least. For this example I will specify all the <resource> elements even though only the filename is required.

```
<jbjf-tasks>
  <task name="t001" order="1" active="true">
    <class>org.jbjf.tasks.ReadTextFile</class>
    <resource type="path">./database/data</resource>
    <resource type="filename">master-data.txt</resource>
    <resource type="delimiter">TAB</resource>
    <resource type="job.stack.key">master.data</resource>
  </task>
```

So, to explain the above <task> element we'll take them line-by-line:

- 🔗 The name attribute (t001) must be unique for the JBJF Batch Definition file.
- 🔗 The order attribute (1) is critical and indicates when in the batch process this task should run.
- 🔗 The active attribute (true) indicates whether the task will run in the batch process. This is typically set to true, but is handy for debugging purposes.
- 🔗 The <class> element is a fully qualified class name to the Task class. In this case I've placed my class in a package called org.jbjf.tasks.
- 🔗 The <resource type="path"> is the value for the directory path.
- 🔗 The <resource type="filename"> is the value for the filename to read in.
- 🔗 The <resource type="delimiter"> is the value for text file delimiter, TAB is converted to \t.
- 🔗 The <resource type="job.stack.key"> is the value that will be used for the Job Stack Key to store our results/outcome from the task.

This completes the first Task for this batch process.

LoadDelimitedDocument

The next task we need is one that takes the DelimitedDocumentAdapter and loads the data into a database table. The basic steps that we want to do in this Task are the following:

- 🔗 Validate and initialize the plugin definitions.
- 🔗 Use the valid plugin/connection definitions to establish a JDBC (java.sql.Connection) connection to the database.
- 🔗 Pulls the DelimitedDocumentAdapter off the Job Stack.
- 🔗 For each record in the DelimitedDocument
 - Build the SQL statement for an Insert into the database.
 - Run the SQL statement to the database.
 - Record the success/failure of the transaction.
- 🔗 Disconnect the JDBC connection.
- 🔗 Store the results onto the Job Stack.

Because it's a database task, it also has to do a number of housekeeping steps as well:

- 🔗 Instantiate the Database object, the resource element will contain the plugin definition.
- 🔗 Locate the correct SQL definition to use.

- Instantiate the Cipher object if the resource was provided.
- Establish a JDBC database connection to the SQL database.
- Close out the JDBC connection when the work is complete.

The good news is you can wrap most of the housekeeping steps into an Abstract Task class and then just extend that Abstract Task class with all your SQL database tasks. I'll leave the Abstract class for you to do. For simplicity, we'll create a Task class with all the steps for illustration purposes.

Start by sketching out how the JBJF Batch Definition <task> element might look for your task, specifically the <resource> elements that will be needed. We basically need a <resource> element for each of the different objects we need to carry out an SQL database task ... we listed these earlier. We also need a <resource> that points the task to where the Delimited Document text file is. That puts us up to 5 <resource> elements. And finally, we need a Job Stack key to store our results, if any:

```
<!--
SQL : Load a table...
-->
<task name="tTBD" order="TBD" active="true">
  <class>org.jbjf.tasks.LoadDelimitedDocument</class>
  <resource type="sql-definition">insert-to-a-table</resource>
  <resource type="connection">jbjf</resource>
  <resource type="plugin-database">mysql-database</resource>
  <resource type="plugin-cipher">default-cipher</resource>
  <resource type="delimited-document">master.data</resource>
  <resource type="job-stack-key">load.delimited.document</resource>
</task>
```

Note:

The plugin-cipher resource is optional. Cipher encryption is typically used for encryption/decryption of users and passwords for database, ftp, etc... connections.

With our <task> element sketched out, we can begin the Java Task class. Start by extending the AbstractTask class. Click the Finish button when all the properties have been set. Eclipse will generate the following skeleton class, I've removed the comments for brevity:

```
package org.jbjf.tasks;

import java.util.HashMap;
import org.jbjf.core.AbstractTask;

public class LoadDelimitedDocument extends AbstractTask {
    /**
     * Default constructor. Required for JBJF integration.
     */
    public LoadDelimitedDocument() {
        // TODO Auto-generated constructor stub
    }

    /* (non-Javadoc)
     * @see org.jbjf.core.AbstractTask#runTask(java.util.HashMap)
     */
    @Override
    public void runTask(HashMap pjobParameters) throws Exception {
        // TODO Auto-generated method stub
    }
}
```

```
}
```

The skeleton includes the basics from the AbstractTask, a default constructor and the runTask() method. The default constructor is "required", as it will be used when the JBJF framework creates the Task during runtime. Using the above list of resource elements, we define the following Class properties in our Task class. I've also included some counters to track success/failure/total counts from the SQL processing. The last property is the JDBC connection object. One thing to note on the JDBC connection object, there's probably 10 different ways to manage this object, I'm using a Class property level object for easier illustration:

```
/**
 * Comments removed for brevity...
 */
public class LoadDelimitedDocument extends AbstractTask {

    public static final String ID = AbstractSQLTask.class.getName();

    private String SHORT_NAME = "AbstractSQLTask()";

    private String SYSTEM_IDENTITY = String.valueOf(System.identityHashCode(this));

    protected JBJFSQLDefinition sqlDefinition;

    protected JBJFDatabaseConnection connectionDefinition;

    protected String pluginDefinitionDatabase;

    protected String pluginDefinitionCipher;

    protected IJBJPPluginDatabase pluginRuntimeDatabase = null;

    protected IJBJPPluginCipher pluginRuntimeCipher = null;

    protected String jobStackKey;

    protected DelimitedDocumentAdapter delimitedDocument;

    protected long totalRowCount = 0;




    protected long failureRowCount = 0;

    protected long successRowCount = 0;

    protected Connection connection;
```

To support the Plugins we need four properties 2 (Plugins) * 2. Basically for any one Plugin we need two properties, a String to store the id of the plugin-definition and then an IJBJPPlugin[Database|Cipher] object for the actual runtime plugin. Also, please note we define all these Class Properties as protected and not private. This allows any sub-classes of this Task full access to the property. Finally, to follow the Java standard it's recommended that you generate Getters and Setters for all these Class properties. Use Eclipse > Source > Generate Getters/Setters. Later we will revise the getConnection() method to process and instantiate the plugins.

With the properties in place we can begin construction of the Task class. Each Task either extends AbstractTask and/or implements ITask, and it's the ITask interface that JBJF will use to cast your Task class during runtime. By default, JBJF will process each Task in the same manner:

-  Loads all IJBJPPlugin classes that are present in the "plugins" directory.
-  initTask()
-  preTask()

- 🔗 runTask()
- 🔗 postTask()

initTask()

Early versions of JBJF would automatically pickup and initialize resource elements like connections and sql-definitions. However, later versions removed this dependency and allowed special SQL elements like connections and sql-definitions to be processed by specialty Abstract Base Task classes. To make a long explanation short, we need to override the initTask() and pre-process the specialty resource elements before the runTask() is called. By default, JBJF will call initTask() prior to runTask(). Thus, it's in initTask() that we wish to perform many of these preliminary things.

With the Task class in the Eclipse editor:

- 🔗 Navigate to Source > Override/Implement methods.
- 🔗 Select the initTask() to override.
- 🔗 Eclipse generates an initTask() method and it SHOULD have a super.initTask() in place...If not, then put one in.

```
@Override
public void initTask(HashMap jobStack) throws Exception {
    // TODO Auto-generated method stub
    super.initTask ( jobStack );
}
```

We let the super.initTask() do the built-in initialization. We just have to add code in here to take care of the specialty <resource> elements for the connections and sql-definition that we know our current Task needs. Please note the use of getter/setter methods when using Class properties.

```
@Override
public void initTask(HashMap jobStack) throws Exception {
    super.initTask ( jobStack );
    getLog().debug( "Initialize SQL Work Unit...Start..." );

    if ( getResources() != null ) {
        getLog().debug( "Gathering SQL resources..." );
        Iterator lstepThru = getResources().entrySet().iterator();
        while ( lstepThru.hasNext() ) {
            Entry ltheItem = (Entry)lstepThru.next();
            String lstrTypeResource = (String)ltheItem.getKey();
            if ( lstrTypeResource.equalsIgnoreCase("sql-definition") ) {
                getLog().debug( "Gathering " + lstrTypeResource + " [ " +
                    (String)ltheItem.getValue() + " ]" );
                setSqlDefinition (
                    (JBJFSQLDefinition)getDefinition().getSQLDefinitions().get
                    (
                        (String)ltheItem.getValue()
                    )
                );
            }
            else if (lstrTypeResource.equalsIgnoreCase("connection")) {
                setConnectionDefinition (
                    (JBJFDatabaseConnection)
                    getDefinition().getConnections().get (
                        (String) ltheItem.getValue()
                    )
                );
            }
        }
    }
}
```

```

        lstepThru = null;
    }
    getLog().debug( "Initialize SQL Work Unit...Complete..." );
}

```

JBJF will process these <resources> in the `initTask()` and place them in the appropriate class properties. We defined these properties earlier in the tutorial and then generated the getter/setter methods. We then use those getter/setter methods in `initTask()` to take the values from the <resource> elements.

In the next phase of `initTask()`, we need to locate the Plugin Definitions (Database and possibly Cipher) and place them into local class properties. We then use the Plugin Definitions to locate the Runtime Plugins and initialize them. Step one is to locate and set the plugin definitions listed in the <resource> elements of the <task>. The value of the <resource> element is the plugin-definition id. By default, JBJF will process all your <resource> elements and their values for your task. These are then available using the `getResources()` as a `HashMap` collection. Earlier in the tutorial we defined the class properties and then generated getter/setter methods. So here we use those getter/setter methods:

```

// PLUGIN-DEFINITION : Database
// Grab the database plugin definition, this is the
// plugin-definition (XML) object, not the actual runtime
// object
//
setPluginDatabaseDefinition (
    (JBJFPluginDefinition)getResources().get ( "plugin-database" )
);

```

The above is repeated for the Cipher plugin as well...note here we check to make sure they provided a Cipher.

Coding Tip:

Remember, some database connection may choose to provide credentials in clear text. In this situation, you would not provide a plugin-definition for a Cipher plugin. Just remove the <resource> element for the Cipher.

By default, JBJF will detect when a Cipher <resource> is provided and “assume” that your user and password are encrypted. You can control this by the inclusion or exclusion of the <resource> in the <task> element.

```

// PLUGIN-DEFINITION : Cipher
// Grab the OPTIONAL cipher plugin definition, this is the plugin-
definition
// (XML) object, not the actual runtime object
//
if ( getResources().containsKey( "plugin-cipher" ) ) {
    setPluginCipherDefinition (
        (JBJFPluginDefinition)getResources().get ( "plugin-cipher" )
    );
}

```

We then use the Plugin Definitions to locate and initialize the Plugin Runtime instances. The Runtime Plugins are gathered during the Batch initialization and are configured with the current `log4j` ONLY using the `IJBJFPlugin` interface. At the Task phase we use the plugin-definition to complete the Runtime Plugin initialization/configuration. Every Plugin is designed to accept a corresponding Plugin Definition. Database Plugins implement the interface `IJBJFPluginDatabase` that standardizes the entire Database Plugin Runtime lifecycle. Cipher Plugins implement the interface `IJBJFPluginCipher` that standardizes the Cipher Plugin Runtime lifecycle. We use the id (attribute) of the plugin-definition to perform the lookups. Again, the Cipher plugin is an optional resource, so we do a quick check to see whether a cipher is needed.

```

// RUNTIME PLUGINS : Database
// Grab the runtime plugins from the task cache...Using

```



```

// the id from the plugin-definition, we can lookup the actual
// runtime instances of the plugins.
//
setPluginDatabaseRuntime (
    getTaskPlugins().getDatabasePlugin(
        getPluginDatabaseDefinition().getPluginId()
    )
);

// RUNTIME PLUGINS :
// Populate the database plugin with the connection
// definition property values. The connection definition "defines"
// the database connection and the database plugin
// "implements" the JDBC connection. The JDBC connection
// is NOT established yet, only the properties are supplied...
//
if ( getConnectionDefinition() != null ) {
    getPluginDatabaseRuntime().setProperties( getConnectionDefinition() );
}
else {
    throw new Exception (
        SHORT_NAME
        + " Plugin Database Definition [" +
getPluginDatabaseDefinition().getPluginId ()
        + " has no Connection Definition."
        + " Please check the resources for the task [" + this.toString ()
+ "]"
    );
}

// RUNTIME PLUGINS : Cipher
// Grab the runtime plugins from the task cache...Using
// the id from the plugin-definition, we can lookup the actual
// runtime instances of the plugins.
//
if ( getPluginCipherDefinition() != null ) {
    setPluginCipherRuntime (
        getTaskPlugins().getCipherPlugin(
            getPluginCipherDefinition().getPluginId()
        )
    );
}
}

```

At this point we have the Runtime Database plugin and the optional Runtime Cipher plugin. This completes the initialization for the Task and we have all the objects we need to create the JDBC connection when we need it. Since we're only in the `initTask()`, there's no need to create the JDBC connection, we can wait until the `runTask()`. However, you can certainly move forward and carry out the JDBC connection, the choice is really yours. For this tutorial, I'm going to wait until the `runTask()` though.

This completes the `initTask()`.

getConnection ()

Earlier in the tutorial we defined a class property (`java.sql.Connection`) and generated getter/setter methods for that property. Also, in the `initTask()` we initialized both the Plugin Definitions and the Runtime instances for the Database and Cipher plugins.

We need to replace the default getter method, getConnection(), and integrate the Database and Cipher plugins to create the JDBC connection object. We start by putting in a check for the existence of the Database Runtime Plugin, we only want to try and create the connection if the Plugin is not null, otherwise what's the point.

```
if ( getPluginDatabaseRuntime() != null ) {
}
```

Since we can't predict when the getConnection() method gets called, or how many times, start by putting in a simple check that tests to see whether the connection has already been created. There's a good chance that the Database Plugin may have already created the connection, especially if the task is running in multiple threads or the plugin gets used in multiple tasks. We also check to see whether the connection is closed? If closed, then we want to clean up any objects inside the Runtime Plugin by calling the closeConnection()...this will close out the connection and set it equal to null.

```
if ( this.connection != null ) {
    if ( this.connection.isClosed() ) {
        this.connection = null;
        getPluginDatabaseRuntime().closeConnection ();
    }
}
else {
    this.connection = null;
}
```

In the next step we need to check the Class property (this.connection) to see whether it was set to null during the checks.

```
if ( this.connection == null ) {
}
```

FINALLY, we can actually make the call into the Database Plugin to create the connection. Again, we check for a Cipher plugin and pass the Cipher Plugin into the Database Plugin if it is present. What gets returned is the Connection in an Open state, ready to run SQL statements:

```
// CONNECT : Connect to the database...pass in the cipher
// plugin to decrypt the userid and password.
//
if ( getPluginRuntimeCipher() != null ) {
    this.connection =
getPluginRuntimeDatabase().getConnection(getPluginRuntimeCipher());
}
else {
    this.connection = getPluginRuntimeDatabase().getConnection();
}
```

This completes the getConnection() method. You can view the finished method in the example code. As I mentioned earlier, this method is reusable in any SQL Task class, so keep this code handy.

Coding Tip:

Keep this getConnection() method handy as we will be re-using this in other SQL Tasks in this tutorial. I won't be going through this getConnection() again, I'll simply refer to this section of the document.

runTask ()

In our final piece of coding we'll assemble the runTask() method. This is the actual method that runs the code that the task was designed for.

Coding Tip:

In the example code, you may see code that is not covered in the tutorial. This is code and/or concepts presented in other JBJF tutorials. I won't spend time and space presenting them here. These include but are not limited to:

- Required Resources

I'm going to setup some local constants that mirror all the different resources that the class can accept...this includes required and optional resources:

```
private String REZ_KEY_SQLDEFINITION      = "sql-definition";
private String REZ_KEY_CONNECTION        = "connection";
private String REZ_KEY_PLUGINDATABASE    = "plugin-database";
private String REZ_KEY_PLUGINCIPHER      = "plugin-cipher";
private String REZ_KEY_DELIMITEDDOC      = "delimited-document";
private String REZ_KEY_JOBSTACK          = "job-stack-key";
```

Next, we will setup a try {} catch () {} block to protect and record any errors/exceptions that the task may encounter. This is standard Java practice and it's a good idea in JBJF to log the exception before re-throwing the exception up to the parent class.

```
try {
}
catch ( Exception ltheXcp ) {
    getLog().fatal(ltheXcp.getMessage());
    throw ltheXcp;
}
```

From here, all task code should be enclosed in the try {} section. We start by getting the DelimitedDocumentAdapter off the Job Stack. For this we need the Job Stack key that the previous task (ReadTextFile) used to store the object onto the Job Stack. We then use that key to lookup the DelimitedDocumentAdapter object on the job stack and initialize the class property.

```
/* RESOLVE RESOURCES : Locate and store the individual
 * objects from the Job Stack into the class properties...
 */
setDelimitedDocument (
    (DelimitedDocumentAdapter) getParameters ().get (
        (String)getResources().get ( REZ_KEY_DELIMITEDDOC )
    )
);
setJobStackKey ( (String)getResources().get ( REZ_KEY_JOBSTACK ) );
```

Next, we need to establish the database connection using our newly revised getConnection() method. To guard against any kind of environmental database settings and/or timeout disconnects we need to test the connection returned...remember, in our current development environment we have complete control. Once we release our Task into the runtime Production world, it will get extended, it will be threaded out, and it will run in differing database environments with differing configuration settings. That said, we have the following tests (if-else if-statements)

- Null – Test and make sure the connection even got created.

```
if ( getConnection() == null ) {
```

```

        throw new Exception (
            "A database connection could not be established. Please"
            + " check all properties in the <connection> element"
            + " named [" + getConnectionDefinition().getName () + "]"
        );
    }
    else {
    }
}

```

The else {} (default) section is where we want to put our final code. The final code really just iterates through all the rows in the Delimited Document and builds an INSERT SQL statement. Then runs that SQL using our database connection. This continues until all the rows are exhausted. Most of this code is standard Java JDBC Statement code. There is some code that touches on JBJF concepts and objects, and that is what I will cover here. I also put comments in the code to assist in your understanding.

Coding Tip:

The following code is written in a pattern like paradigm. In fact, you can use this code for ANY JBJF SQL task that deals with an insert statement, or a JBJFSQLDefinition object with many JBJFSQLParameters:

```

else {
    Statement    ldtbStatement = null;
    // STATEMENT : Create the database statement...
    //
    ldtbStatement = getConnection().createStatement ();

    // LEADING ZERO : Provides a simple decimal format
    // with leading zeros that allows us to iterate
    // thru all the JBJFSQLParamaters, p001, p002, p003, etc...
    //
    DecimalFormat    leadingZeros = new DecimalFormat ( "000" );

    int             rowCount = 0;
    int             successCount = 0;

    // ROWS : Iterate thru all the rows in the document...
    //
    for ( int lngIdx=0; lngIdx<docSource.getRecordCount(); lngIdx++ ) {
        FlatRow vmRow = (FlatRow)docSource.getRecordContainer(lngIdx);
        // COLUMNS : Iterate thru all the columns in the row...
        //
        for ( int colIdx=0; colIdx<vmRow.getColumnCount(); colIdx++ ) {
            // sql-definition params are "1" based...
            // build a lookup key to the current SQL
            // parameter...p00n...
            //
            String lparmKey = "p" + leadingZeros.format(colIdx+1);
            // Use the lookup key to find the parameter object...
            //
            JBJFSQLParameter ltheParm =
            (JBJFSQLParameter)getSqlDefinition().getParameters().get(lparmKey);
            // Grab the value from the document row column...
            String ltheValue = vmRow.getColumnValue(colIdx);
            if ( ltheParm != null ) {
                if ( ltheValue != null ) { ltheValue = scrubData ( ltheValue ); }
                // Set the value of the parameter using the
            }
        }
    }
}

```

```

        // column value...
        ltheParm.setValue (
            ltheValue
        );
    }
}

// SQL : Resolve parameters and fetch the SQL...
// This will take all the SQL Parameter values
// and substitute them into the SQL Statement...
//
String lstrSQL = getSqlDefinition().getStatement();

// NULL : Force all 'null' to true NULL...
lstrSQL = lstrSQL.replaceAll( "'null'", "null" );

// RUN : Run the SQL against the database...
//
getLog().debug( SHORT_NAME + " SQL " );
getLog().debug( "-----" );
getLog().debug( lstrSQL );
getLog().debug( "-----" );
try {
    ldtbStatement.executeUpdate ( lstrSQL );
    successCount++;
}
catch ( SQLException lsqLXcp ) {
    getLog().warn( SHORT_NAME + " SQLException on Record [" + rowCount +
"]" );
    getLog().warn( lstrSQL );
}
catch ( Exception ltheXcp ) {
    throw ltheXcp;
}
rowCount++;
}

if ( rowCount != successCount ) {
    getLog().warn( SHORT_NAME + " Not all rows were successfully loaded. " +
successCount + " of " + rowCount + " were loaded!" );
}

// GC : Check for the DelimitedDocument on the job
// stack and remove it for GC...
String lkeyDoc = (String)getResources().get( "job-stack-key" );
if ( getParameters().containsKey(lkeyDoc) ) {
    getParameters().remove(lkeyDoc);
}
}

```

At the bottom of the task we do some simple housekeeping:

- Check the rowCount versus the successCount, any difference indicates some rows didn't load.
- Check the Job Stack for the DelimitedDocument object, and remove it from the Job Stack if it's there. This should allow the object to be Garbage Collected.

Coding Tip:

Garbage Collection is a really an important concept. Many JBJF tasks will need to do housekeeping of some kind, otherwise lengthy processes with many tasks can lead to a heap dump. I find the best way is to include another <resource> element as a control mechanism that allows you to choose when to perform the housekeeping.

```
// CHECK : If the rowcount doesn't match the
// success count, then some records didn't load...
//
if ( rowCount != successCount ) {
    getLog().warn (
        SHORT_NAME
        + " Not all rows were successfully loaded.  "
        + successCount + " of "
        + rowCount + " were loaded!"
    );
}
else {
    getLog().info (
        SHORT_NAME
        + " Rows were successfully loaded ["
        + successCount + "]"
    );
}

// GC : Check for the DelimitedDocument on the job
// stack and remove it for GC...
//
String lkeyDoc = (String)getResources().get( REZ_KEY_JOBSTACK );
if ( getParameters().containsKey(lkeyDoc) ) {
    getParameters().remove(lkeyDoc);
}
```

Finally, we add a finally {} class to the try {} catch {} block that calls and closes the database connection. Please note, when we close the connection we use the Plugin method closeConnection. This method traps any Exceptions and logs a message only. This prevents any exception that may occur during the close from terminating the entire process. Also, the connection object is set to null, thus allowing sub-sequent tasks to call the getConnection() and get a fresh connection:

```
catch ( Exception ltheXcp ) {
    getLog().fatal(ltheXcp.getMessage());
    throw ltheXcp;
}
finally {
    getPluginRuntimeDatabase().closeConnection ();
}
```

Coding Tip:

If you see the following exception during a process, chances are you've called the close() method of your connection and not the plugin closeConnection() method. And then a sub-sequent task tries to use the connection and/or plugin. You should ALWAYS call the closeConnection() method of the Database Plugins, this ensures that the connection is closed and then set to null. Thus, any sub-sequent calls will re-create the connection.

Not great, I know, but until I can get a connection pool built into JBJF, this works pretty well.

2013-05-29 22:29:32,289 [main] FATAL jbjf.standard - Processing Exception

`com.mysql.jdbc.exceptions.jdbc4.MySQLNonTransientConnectionException`: No operations allowed after connection closed.

```
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance(Unknown Source)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unknown Source)
```

This completes the `runTask()` method and the task. It's ready to run at this point.

SQLSelectTask

For this next task we are using a pre-built JBJF task. The `SQLSelectTask` works almost in the same manner as our `LoadDelimitedDocument`. The only difference is the Load was putting records into a database table, the Select reads data from the database table. It's really the same set of `<resource>` elements except the delimited document, we don't need one for the Select.

The details of the task are documented in the Javadocs so I won't spend a great deal of time discussing it here. What I do want though is the list of resources needed by the Task class.

Location	Id/Name	Type	Required	Description/Comments
<code><task></code>	sql-definition	String	True	The <code><sql-definition></code> in the JBJF Batch Definition file that contains the SQL statement to run. The SQL statement is assumed to have no parameters, thus no substitution is performed prior to statement execution.
<code><task></code>	connection	String	True	Contains the id of the <code><connection></code> element in the JBJF Batch Definition file that contains the database to use for the statement execution.
<code><task></code>	sql-results	String	True	A Job Stack key that will be used to store the <code>java.sql.ResultSet</code> on the Job stack.
<code><task></code>	plugin-database	JBJFPluginDefinition	True	The id for the <code><plugin-definition></code> that will get used in this task. This will be the Database Plugin (<code>IJBJFPluginDatabase</code>) for the JDBC connection management.
<code><task></code>	plugin-cipher	JBJFPluginDefinition	False	The id for the <code><plugin-definition></code> that will get used in this task. This will be the Cipher Plugin (<code>IJBJFPluginCipher</code>) for the encryption/decryption of database <code><connection></code> properties.

Using the above table we can quickly code the `<task>` element in the JBJF Batch Definition file. OR you can just copy the `LoadDelimitedDocument` `<task>` element we created in the previous task and remove the delimited-document `<resource>`.

```
<task name="t003" order="3" active="true">
  <class>org.jbjf.tasks.SQLSelectTask</class>
  <resource type="sql-definition">select-from-master-data</resource>
  <resource type="connection">jbjf</resource>
  <resource type="plugin-database">mysql-not-persistent</resource>
  <resource type="plugin-cipher">default-cipher</resource>
  <resource type="job-stack-key">select.master.data</resource>
</task>
```

The Select Task places the `ResultSet` (rows of data) onto the Job Stack using the job-stack-key `select.master.data`. It is this value that you use in later tasks to pull the object off the Job Stack.

LoadResultSet

The next task we need is to pull the `ResultSet` off the Job Stack from a previous Select SQL and load that data into the customers database table. We've already done one load using a text file to Delimited Document to Database table. In this one we're taking a typical `java.sql.ResultSet` and loading it into a database table.

Unlike the previous Load task, I'm going to extend the JBJF `AbstractSQLTask` instead of the base `AbstractTask`. The `AbstractSQLTask` implements much of the mundane initialization, plugin management and `getConnection()`, so we won't be burdened with reproducing all that code over and over again. In fact, the only method we really need to implement is the `runTask()`.

You are free to copy the `LoadDelimitedDocument` and remove unneeded code, but I will start from scratch.

In Eclipse, start by right clicking on the Java package `org.jbjf.tasks` and selecting `New Class...Superclass is AbstractSQLTask` and the name is `LoadResultSet`. By extending the `AbstractSQLTask` instead of the typical `AbstractTask`, we gain a great many built-in methods, properties and plugin management that we implemented in the `LoadDelimitedDocument`.

New Java Class

Create a new Java class.

Source folder:

Package:

☐ Enclosing type:

Name:

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass:

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☒ Constructors from superclass

☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☒ Generate comments

This will give you the default constructor and the runTask(), just like before:

```
public class LoadResultSet extends AbstractSQLTask {

    /**
     *
     */
    public LoadResultSet() {
        // TODO Auto-generated constructor stub
    }

    /* (non-Javadoc)
     * @see org.jbjf.tasks.AbstractSQLTask#runTask(java.util.HashMap)
     */
    @Override
```

```

public void runTask(HashMap jobStack) throws Exception {
    // TODO Auto-generated method stub

}

}

```

Again, much of the Required Resources won't be covered here, but you'll see it in the example code. Like before, we need to define a couple of class properties to hold the ResultSet and the Job Stack Key. And we use the Source > Generate Getters and Setters...

```

/**
 * Class property that stores the Job Stack key where the
 * <code>DelimitedDocumentAdapter</code> is stored.
 */
protected String jobStackKey;

/**
 * Class property that stores the java.sql.ResultSet from the
 * Job Stack.
 */
protected ResultSet resultSet = null;

```

Because we'll be building an INSERT SQL statement to match the database table name, we need to add another class property to store the field names from the ResultSet that we'll be iterating thru. Since we need to order the fields in the same way as the columns in the database table, we use a simple List (ArrayList) to store the names. And we initialize the names in the default constructor in the sequence that the for loop in the code will encounter them:

```

/**
 * Class property that stores a list of <code>String</code>
 * field names that will be processed.
 */
private ArrayList fields = null;

/**
 * Default constructor. Needed by JBJF for dynamic allocation.
 */
public LoadResultSet() {
    // CHECK : Abstract classes need to always check to make sure
    // a parent class has NOT already initialized the required
    // resources.
    if ( getRequiredResources() == null ) {
        mtaskRequired = new ArrayList();
    }
    getRequiredResources().add("sql-definition"); /* sql statement to run */
    getRequiredResources().add("connection");      /* database connection */
    getRequiredResources().add("plugin-database"); /* JDBC management */
    fields = new ArrayList();
    fields.add("id_customer");
    fields.add("first_name");
    fields.add("middle_name");
    fields.add("last_name");
    fields.add("job_title");
}

```

I'll explain the use of these fields in more detail when we get to the runTask() code. For now, I just need you to be aware that the fields are needed in the class.

runTask ()

The runTask() will contain our code that iterates thru the ResultSet, builds the INSERT statement contained in our sql-definition and then executes that SQL against our database. Earlier I defined the fields list, and promised the details on how this works...so here goes:

For the INSERT to work correctly, we really need to coordinate the sql-definition for the task, along with the ResultSet on the Job Stack and finally the columns on the database table:

sql-definition:

```
<sql-definition name="insert-into-customers" order="1">
  <text>
    <![CDATA[
      INSERT INTO jbjf.customers (
        id_customer
        ,first_name
        ,middle_name
        ,last_name
        ,job_title
        ,company_name_1
        ,company_name_2
        ,entered_on
        ,entered_by
        ,modified_on
        ,modified_by
      )
      VALUES (
        ?
        ,?
        ,?
        ,?
        ,?
        ,null
        ,null
        ,CURDATE() /* 8-entered_on */
        ,USER() /* 9-entered_by */
        ,null /* 10-modified_on */
        ,null /* 11-modified_by */
      )
    ]]>
  </text>
  <sql-parameters>
    <param name="p001" order="1" type="int">0</param>
    <param name="p002" order="2" type="string">0</param>
    <param name="p003" order="3" type="string">0</param>
    <param name="p004" order="4" type="string">0</param>
    <param name="p005" order="5" type="string">0</param>
  </sql-parameters>
</sql-definition>
```

ResultSet, SELECT SQL was run in the previous task:

```

<sql-definition name="select-from-master-data" order="1">
  <text>
    <![CDATA[
      SELECT
        id_customer
      ,first_name
      ,middle_name
      ,last_name
      ,job_title
      ,entered_on
      ,entered_by
      ,modified_on
      ,modified_by
      FROM jbjf.master_data
      where zip_code = ?
    ]]>
  </text>
  <sql-parameters>
    <param name="p001" order="1" type="string">03103</param>
  </sql-parameters>
</sql-definition>

```

Database table:

```

CREATE TABLE `customers` (
  `id_customer` bigint(20) NOT NULL AUTO_INCREMENT,
  `first_name` varchar(64) NOT NULL,
  `middle_name` varchar(64) DEFAULT NULL,
  `last_name` varchar(64) NOT NULL,
  `job_title` varchar(64) DEFAULT NULL,
  `company_name_1` varchar(64) DEFAULT NULL,
  `company_name_2` varchar(64) DEFAULT NULL,
  `entered_on` timestamp NULL DEFAULT NULL,
  `entered_by` varchar(64) DEFAULT NULL,
  `modified_on` timestamp NULL DEFAULT NULL,
  `modified_by` varchar(64) DEFAULT NULL,
  PRIMARY KEY (`id_customer`)
) ENGINE=InnoDB AUTO_INCREMENT=2850 DEFAULT CHARSET=latin1$$

```

JBJF will do some of the work for us, but we essentially need to pre-process all the `<sql-parameters>` in the insert-into-customers sql-definition and assign the current value to the parameter using the value in the corresponding column from the ResultSet. To get the `<sql-parameters>` we need to use the parameter id (p001, p002, etc...) and look it up:

So let's build the code. We start by setting up a traditional while loop that iterates thru the ResultSet:

```

// DATA : Iterate thru the rows of data...
while ( getResultSet().next() ){

} //end while dataSource.next()*

```

Inside the while loop we then grab an iterator for all the fields we need to loop thru as well as a column index counter:

```

Iterator    lfieldIter = getFields().iterator();
int         colIdx = 0;

```

We setup a second while loop for the fields iterator, and we increment the column index within that loop. Notice we initialize the column index as zero, but increment at the beginning of the loop. The column index will actually be used to build the sql-parameter key id that we use to lookup the JBJFSQLParameter object from JBJF:

```
while ( lfieldIter.hasNext() ) {
    colIdx++;

} /* END while ( fields ) */
```

Using the column index counter and a simple leading zero format string, we build the parameter id:

```
String lparmKey = "p" + leadingZeros.format(colIdx);
```

We then use that id to lookup the sql-parameter (JBJFSQLParamater) object from JBJF:

```
JBJFSQLParameter ltheParm = (JBJFSQLParameter)
getSqlDefinition().getParameters().get(lparmKey);
```

We then grab the value of the current ResultSet row using the field name:

```
String ltheValue =
getResultSet().getString((String)lfieldIter.next());
```

Then we do a simple check and make sure the JBJFSQLParameter object is non-null...avoids a NullPointerException:

```
if (ltheParm != null) {

}
```

We scrub the data of any problematic characters...this code is simple and doesn't require review:

```
ltheValue = scrubData(ltheValue);
```

And then we inject the value into the JBJFSQLParamater object:

```
ltheParm.setValue(ltheValue);
```

You can see the finished code in the example source. I know it seems like a lot of work, but the good news is, this code is very reusable and can be controlled via the fields class property. Also, setting the ids of the sql-parameters using a counter of some kind allows us to use the while loop for easier coding.

Once we drop out of the inner loop (fields iterator) we can make a call into the SQL Definition object and, getStatement(). JBJF uses the getStatement() method to take all the Parameter values and replace them ? placeholders with the values. A string will automatically substitute single quotes around the value, an int will simply substitute the value...no quotes.

```
// SQL : Resolve parameters and fetch the SQL...
//
String lstrSQL = getSqlDefinition().getStatement();
```

The remaining code is self-explanatory and was discussed in the LoadDelimitedDocument task.

This completes the runTask() method and the task. It's ready to run at this point.

Creating the JBJF Batch Definition file

Now that we've written the tasks, let focus on the JBJF Batch Definition file. This is the XML file that provides all the data, parameters and task list to JBJF. We've covered some of the XML while writing the tasks, mainly to determine what Java class properties we need.

Unlike the Basics tutorial which had no database connection or SQL, we'll be adding additional elements in this batch definition file. The <jbjf-connections> element will contain the database connection definitions, the <jbjf-plugins> will contain the Plugin Definitions and the <jbjf-sql> will contain the SQL Definitions. For brevity, I'll focus construction of the Batch Definition file to only these segments, the JBJF Basics Tutorial covers many of the traditional elements.

For this tutorial we need to implement:

- ✓ <jbjf-parameters>
- ✓ <jbjf-email>
- ✓ <jbjf-directories>
- ✓ <jbjf-tasks>
- ✓ <jbjf-logs>
- ✓ <jbjf-sql>
- ✓ <jbjf-plugins>

The best option is to copy the Batch Definition XML file from the JBJF Basics Tutorial and then just adding the elements you need. There is also a JBJF Database Tutorials Batch Definition file included in the source for you to review.

jbjf-parameters

For this element we change the name to something like jbjf-database-tutorial, or something that indicates our current tutorial. We'll disable the archivist and enable the email. If you don't have an available email server or SMTP service, then just set this to N to disable it.

```
<jbjf-parameters>
  <name>jbjf-database-tutorial</name>
  <enable-archivist>N</enable-archivist>
  <enable-email>Y</enable-email>
</jbjf-parameters>
```

jbjf-email

The email parameters will differ, depending on the name of your SMTP host. Make sure to setup at least one email recipient and that the attachments attribute is missing or set to N. Also, the email recipients and the SMTP host below are not valid, you'll need to supply your own values in here.

```
<jbjf-email>
  <notifications>
    <email attachments="N">lincolnb@hotmail.com</email>
    <email>lincolnc@hotmail.com</email>
  </notifications>
  <email-host>smtp.host.org</email-host>
  <email-sender>jbjf-database-tutorial@hotmail.com</email-sender>
</jbjf-email>
```

jbjf-directories

The `directories` element is typically utilized for semi-customized tasks that need to do file reads/writes/etc. However, for database tasks or tasks that utilize any plugin service, you **MUST** provide a `directory` element with the name="plugins" that contains a full or partial directory path to where the plugin jarfiles are dropped.

```
<jbjf-directories>
  <directory name="base" addressing="relative">.</directory>
  <directory name="log4j" addressing="relative">etc</directory>
  <directory name="plugins" addressing="relative">plugins</directory>
</jbjf-directories>
```

Coding Tip:

The `<directory>` element named `plugins` is required for batch processes that need at least one plugin. Whether Database or Cipher or Custom plugins, you need this `<directory>` element. JBJF will search for this `directory` element during startup and then load all the plugin classes that have the `IJBjFPlugin` interface in the hierarchy.

```
<directory name="plugins" addressing="relative">[full|partial path]</directory>
```

jbjf-tasks

Using the fully qualified names from the task classes we developed, let's setup the task-list for this batch job. A couple of things to note about the `<task>` element attributes:

- **name** – This **MUST** be unique within the Batch Definition file...i.e. you cannot have two or more tasks with the same name in a single JBJF Batch Definition file.
- **order** – This is a numeric value that **MUST** be in sequence and **MUST** not contain any gaps. For example, 4 tasks **MUST** be numbered [1, 2, 3, 4]. A sequence of [1, 2, 4, 6] will not work.
- **active** – A simple true/false indicator that can control whether the task is actually run.

Coding Directive:

JBJF 1.x.x was developed in a rather short time. To address proper execution order for the tasks I use name and order attributes to keep tasks in sequence. However, it turns out the order **MUST** be in sequence with **NO GAPS** in the numbering. One of the goals in the next major release for JBJF is to remove this requirement.

```
<jbjf-tasks>
  <task name="t001" order="1" active="true">
    <class>org.jbjf.tasks.ReadTextFile</class>
    <resource type="path">./database/data</resource>
    <resource type="filename">master-data.txt</resource>
    <resource type="delimiter">|</resource>
    <resource type="job-stack-key">master.data</resource>
  </task>

  <task name="t002" order="2" active="true">
    <class>org.jbjf.tasks.LoadDelimitedDocument</class>
    <resource type="sql-definition">insert-to-a-table</resource>
    <resource type="connection">jbjf</resource>
    <resource type="plugin-database">mysql-not-persistent</resource>
    <resource type="plugin-cipher">default-cipher</resource>
    <resource type="delimited-document">master.data</resource>
    <resource type="job-stack-key">load.delimited.document</resource>
  </task>

  <task name="t003" order="3" active="true">
```

```

    <class>org.jbjf.tasks.SQLSelectTask</class>
    <resource type="sql-definition">select-from-master-data</resource>
    <resource type="connection">jbjf</resource>
    <resource type="plugin-database">mysql-not-persistent</resource>
    <resource type="plugin-cipher">default-cipher</resource>
    <resource type="job-stack-key">select.master.data</resource>
  </task>

  <task name="t004" order="4" active="true">
    <class>org.jbjf.tasks.LoadResultSet</class>
    <resource type="sql-definition">insert-into-customers</resource>
    <resource type="connection">jbjf</resource>
    <resource type="plugin-database">mysql-not-persistent</resource>
    <resource type="plugin-cipher">default-cipher</resource>
    <resource type="job-stack-key">select.master.data</resource>
  </task>
</jbjf-tasks>

```

jbjf-connections

JBJF Connections are simple XML elements that contain all the individual JDBC properties as separate elements. The name attribute is used as a “key” within the JBJF Batch Definition file and allows Task classes to lookup the connection...JBJF named resources.

Each of the other XML elements are self-explanatory. The following elements do require further discussion though:

- type – Currently not utilized, it provides a way to categorize different connections...filesystem, Oracle, MySQL, etc...There are plans to use this in the future.
- usr – The username (login) for the connection. This can be encrypted and a Cipher plugin can be associated to the connection to decrypt.
- pwd – The password for the connection. This can be encrypted and a Cipher plugin can be associated to the connection to decrypt.

For this tutorial we only have one database connection, the MySQL connection to the jbjf database/schema:

```

<!--
DATABASE CONNECTIONS: Optional
Use these <connection> elements to define database connections
that the JBJF batch job needs to run SQL.
-->
<jbjf-connections>
  <connection name="db-jbjf">
    <type>mysql</type>
    <driver>com.mysql.jdbc.Driver</driver>
    <server>localhost</server>
    <database>jbjf</database>
    <client>jdbc:mysql</client>
    <port>3306</port>
    <usr><![CDATA[usr]]></usr>
    <pwd><![CDATA[pwd]]></pwd>
  </connection>
</jbjf-connections>

```


jbjf-plugins

The plugin definitions provide the “definitions” of what plugins the JBJF batch process needs. Unlike other JBJF resources that use the name attribute as the key/lookup, plugin-definitions use the id attribute. So be attentive to NullPointerExceptions when dealing with plugins...99% of the time it’s because you use the name attribute in the task <resource> when you should be using the id.

Note:

The id attribute is the new standard in JBJF. JBJF 2.0.0(+) will use id attributes, not name attributes when released. In fact, every JBJF element will come with a standard set of attributes, id, name, type, etc...

The following table outlines the attributes and elements for a plugin-definition:

Name	Type	Description/Comments
id	attribute	Key/lookup for the plugin definition
name	attribute	Name, to be utilized by the Design Studio.
type	attribute	Fully qualified name of the Plugin Interface.
active	attribute	A true/false indicator that controls whether the plugin can be utilized at this time.
class	element	A fully qualified Java class name to the Plugin class that implements the type interface. <i>This is an important JBJF concept...the Plugin class MUST implement the type, otherwise you will get a Fatal CastException during initialization.</i>

For this tutorial we have two plugins, one database, one cipher:

```
<jbjf-plugins>
  <plugin-definition
    id="default-cipher"
    name="default"
    type="org.jbjf.plugin.IBJFPluginCipher"
    active="true">
    <class>org.jbjf.services.impl.DefaultCipher</class>
  </plugin-definition>

  <plugin-definition
    id="mysql-not-persistent"
    name="mysql"
    type="org.jbjf.plugin.IBJFPluginDatabase"
    active="true">
    <class>org.jbjf.services.db.MySQLNonPersistentService</class>
  </plugin-definition>
</jbjf-plugins>
```

jbjf-logs

We only need a single log file for this tutorial. We'll create the log4j.properties file in a little bit.

```
<jbjf-logs>
  <log-definition name="default">
    <log4j category="org.adym.batch">./etc/log4j.properties</log4j>
  </log-definition>
</jbjf-logs>
```

jbjf-sql

OK, almost done. This section of the JBJF Batch Definition file contains all the different SQL statements that the process will run. These can be SELECT, INSERT, UPDATE, DELETE or Procedure calls. Like most JBJF entities, each SQL statement is encased in an <sql-definition>. The sql-definition is split between the SQL "text" and any SQL Parameters. For brevity, I'll place one sql-definition in the document, the others you can review in the source.

```
<sql-definition name="insert-to-a-table" order="1">
  <text>
    <![CDATA[
      INSERT INTO master_data (
        gender
        ,first_name
        ,middle_name
        ,last_name
        ,street_address_1
        ,city
        ,state
        ,zip_code
        ,country
        ,email
        ,phone
        ,birthdate
        ,card_make
        ,account
        ,ccv
        ,expire_date
        ,job_title
        ,entered_on
        ,entered_by
        ,modified_on
        ,modified_by
      )
      VALUES (
        ? /* 1-gender                */
        ,? /* 2-first_name            */
        ,? /* 3-middle_name           */
        ,? /* 4-last_name             */
        ,? /* 5-street_address_1      */
        ,? /* 6-city                  */
        ,? /* 7-state                 */
        ,? /* 8-zip_code              */
        ,? /* 9-country               */
        ,? /* 10-email                 */
        ,? /* 11-phone                */
        ,STR_TO_DATE(?, '%m/%d/%Y') /* 12-birthdate          */
        ,? /* 13-card_make             */
        ,? /* 14-account               */
        ,? /* 15-ccv                  */
        ,? /* 16-expire_date           */
        ,? /* 17-job_title             */
        ,CURDATE() /* 18-entered_on      */
        ,USER()    /* 19-entered_by        */
        ,CURDATE() /* 20-modified_on       */
        ,null      /* 21-modified_by       */
      )
    ]]>
  </text>
```

```

<sql-parameters>
  <param name="p001" order="1" type="string">0</param>
  <param name="p002" order="2" type="string">0</param>
  <param name="p003" order="3" type="string">0</param>
  <param name="p004" order="4" type="string">0</param>
  <param name="p005" order="5" type="string">0</param>
  <param name="p006" order="6" type="string">0</param>
  <param name="p007" order="7" type="string">0</param>
  <param name="p008" order="8" type="string">0</param>
  <param name="p009" order="9" type="string">0</param>
  <param name="p010" order="10" type="string">0</param>
  <param name="p011" order="11" type="string">0</param>
  <param name="p012" order="12" type="string">0</param>
  <param name="p013" order="13" type="string">0</param>
  <param name="p014" order="14" type="string">0</param>
  <param name="p015" order="15" type="string">0</param>
  <param name="p016" order="16" type="string">0</param>
  <param name="p017" order="17" type="string">0</param>
</sql-parameters>

</sql-definition>

```

Basically any ? placeholder(s) in the SQL Text requires a corresponding SQL Paramater. If your SQL Text has no ? placeholders, then the SQL Parameters are not required. SQL Parameters are 1-based index, not the typical 0-based index like Java. While not required, it's ALWAYS a good idea to wrap the SQL text in a CDATA. Each ? matches to the <param> and the order attribute for the <param> determines what ? it will replace. For example, consider the following SQL statement:

```

select customer_id, first_name, last_name, city, state, zip
from customers
where zip = ?
      and customer_id > ?

```

We have two parameters positioned in the WHERE clause. So the sql-definition for this SQL statement would look like the following:

```

<sql-definition name="select-customers" order="1">
  <text>
    <![CDATA[
      select customer_id, first_name, last_name, city, state, zip
      from customers
      where zip = ?
            and customer_id > ?
    ]]>
  </text>

  <sql-parameters>
    <param name="p001" order="1" type="string">0</param>
    <param name="p002" order="2" type="int">0</param>
  </sql-parameters>

</sql-definition>

```

So the parameters are matched as last_name like ? : p001, zip = ? : p002, customer_id > ? : p003. The numeric order is determined by the position where the ? is located...or kinda when the ? is encountered in the SQL. So, if we have values of

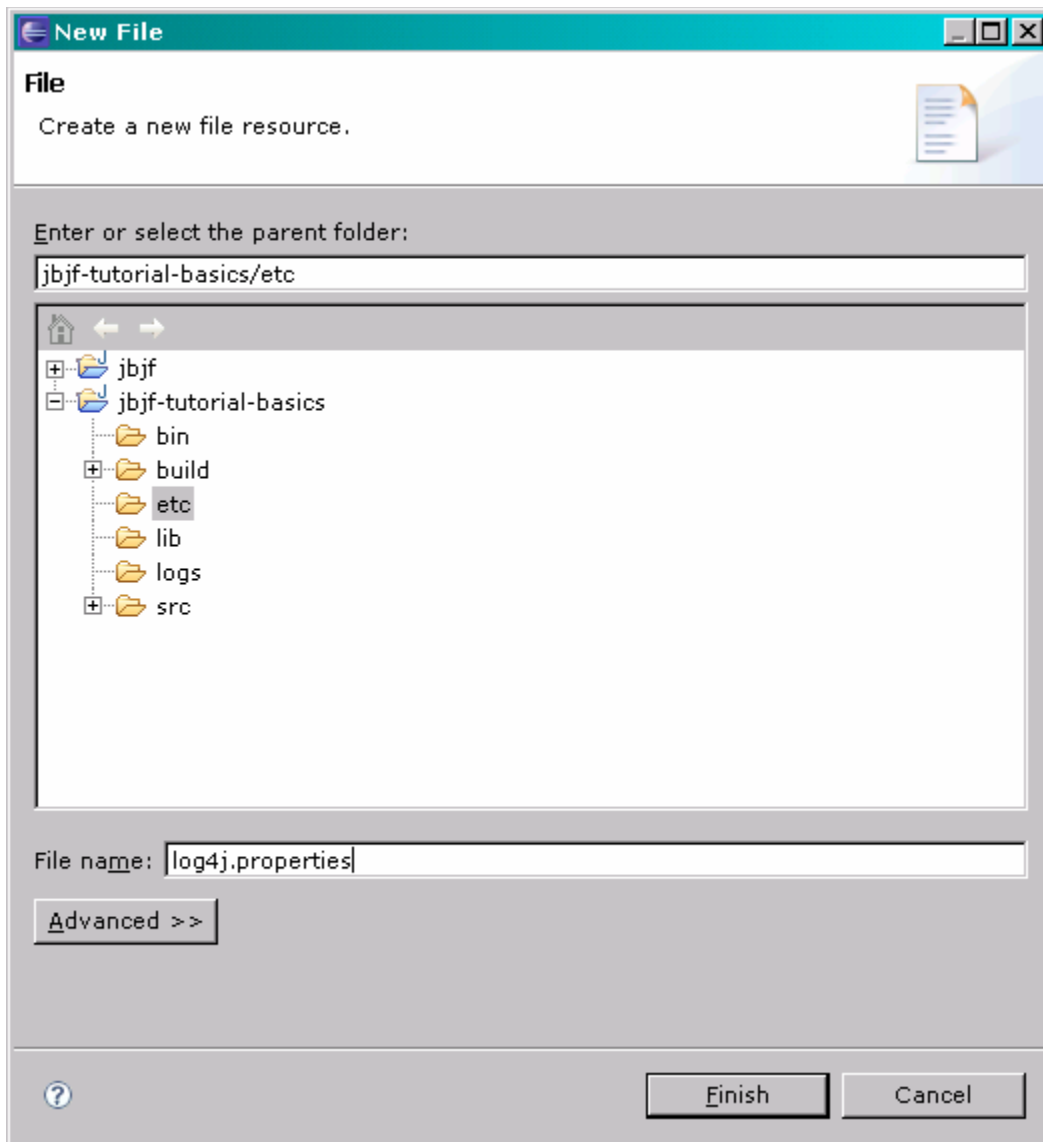
zip=01234 and customer_id=1000 in the Task class, then when JBJF substitutes the values, it will return an SQL statement like the following:

```
select customer_id, first_name, last_name, city, state, zip
from customers
where zip = '01234'
      and customer_id > 1000
```

Notice that the zip is replaced with a quoted string while customer_id is not. This is determined by the type attribute of the <param>. JBJF currently only supports two types of parameters, string and int. However, they really should be listed as quoted or not-quoted, as JBJF will NOT validate whether your actual value is a string or int. In fact, you can pass a float value into customer_id (123.456) and JBJF will simply substitute that value. Dates are an easy workaround by simply using Dates in a string format, 2013-05-23 13:22:34, would be substituted as '2013-05-23 13:22:34'.

Log4j Properties

As a final step, we need to create a log4j properties file. Without diving into all the details of log4j, we really just need to match up our log4j category in the properties file with the category attribute we put in the <log4j> entry in the <jbjf-logs> XML element. The following is a workable properties file. Again, right-click on the ./etc folder in the project and select New > File. In the File dialog, name the file log4j.properties:



Then copy the following code snippet and paste it into the log4j.properties file:

```
# See http://logging.apache.org/log4j/docs/manual.html
# for log4j configuration.

# See http://logging.apache.org/log4j/docs/api/org/apache/log4j/PatternLayout.html
# for conversion pattern formatting.

log4j.category.org.jbjf.tutorial=DEBUG, logfile, commandline

log4j.appender.logfile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.logfile.file=./logs/jbjf-database-tutorial.log
log4j.appender.logfile.DatePattern='.'yyyy-MM-dd
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
log4j.appender.logfile.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

log4j.appender.commandline=org.apache.log4j.ConsoleAppender
log4j.appender.commandline.layout=org.apache.log4j.PatternLayout
log4j.appender.commandline.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
```

I've **bolded** and *italicized* the key entries for the properties file. You'll not the need to create a sub-directory in the `${project-dir}` called logs...`${project-dir}/logs`.

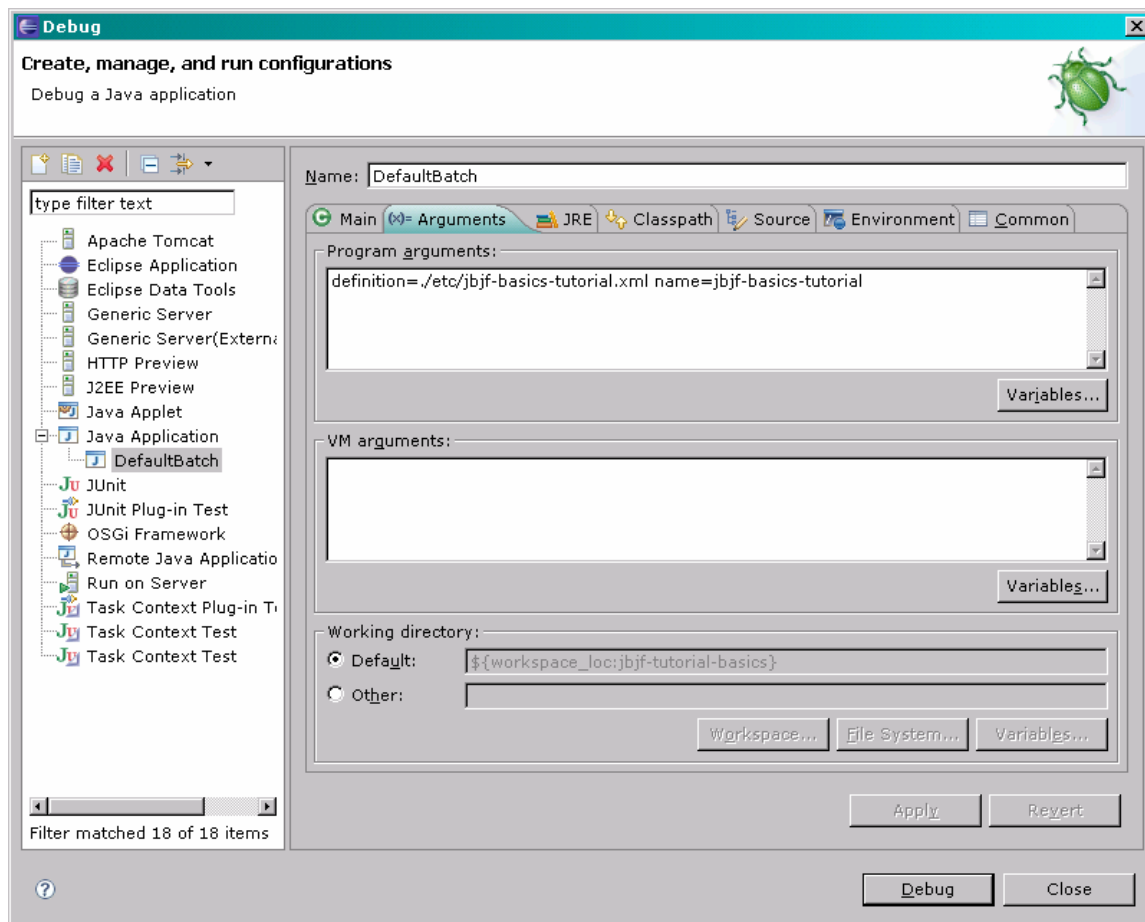
Running the Batch Job

There are a couple of ways to execute the batch job. The easiest way involves using the Eclipse Launch facility. Another way is to use the traditional jarfile and script file process. More involved, but not impossible. The basics tutorial includes code, jarfiles and script files for both approaches.

Launch Facility

Included in the source is a *.launch file that you can use in Eclipse. Simply load it in as a Run (or Debug) configuration and run as needed.

Ordinarily we would only need to supply a single command line argument, the JBJF Batch Definition file. The argument is passed in as key=value pairs.



Click the Apply button and then the Debug/Run button. This will run the application now, passing the command line argument as requested. Logging results and output will appear on the Console View of Eclipse...the amount of output will depend on the level of logging you set in the Log4j properties file. In my case, I have the DEBUG log level and my output is as follows:

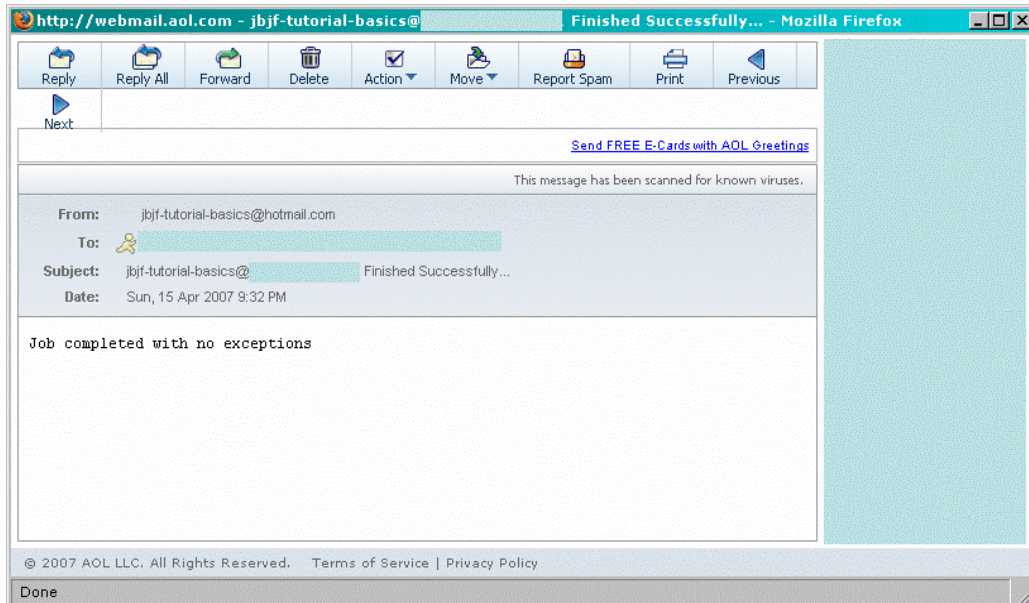
```
2013-06-08 20:06:29,507 [main] DEBUG jbjf.standard - Found Plugin DefaultCipher class
[org.jbjf.services.impl.DefaultCipher]
2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin MySQLPersistentService class
[org.jbjf.services.db.MySQLPersistentService]
2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin null class
[org.jbjf.services.db.MySQLNonPersistentService]
```

```

2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin mysql-not-persistent class
[org.jbjf.services.db.MySQLNonPersistentService]
2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin DefaultCipher class
[org.jbjf.services.impl.DefaultCipher]
2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin MySQLPersistentService class
[org.jbjf.services.db.MySQLPersistentService]
2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin mysql class
[org.jbjf.services.db.MySQLNonPersistentService]
2013-06-08 20:06:29,649 [main] DEBUG jbjf.standard - Found Plugin default-cipher class
[org.jbjf.services.impl.DefaultCipher]
2013-06-08 20:06:29,773 [main] INFO jbjf.standard - Starting jbjf-database-tutorial-001...
2013-06-08 20:06:29,773 [main] INFO jbjf.standard - Work Unit ... Starting ...class
org.jbjf.tasks.ReadTextFile
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Initialize Work Unit...Start...
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Gathering resources...
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Gathering delimiter [ | ]
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Gathering path [ ./database/data ]
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Gathering job-stack-key [ master.data ]
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Gathering filename [ master-data.txt ]
2013-06-08 20:06:29,773 [main] DEBUG jbjf.standard - Initialize Work Unit...Complete...
2013-06-08 20:06:29,773 [main] INFO jbjf.standard - ReadTextDocument()...Start...

```

Finally, if your SMTP email host, recipients and sender properties are correct, then you'll receive an email at those email recipients that indicates a successful batch job run. In the example below, I sent my email to a freebie recipient:



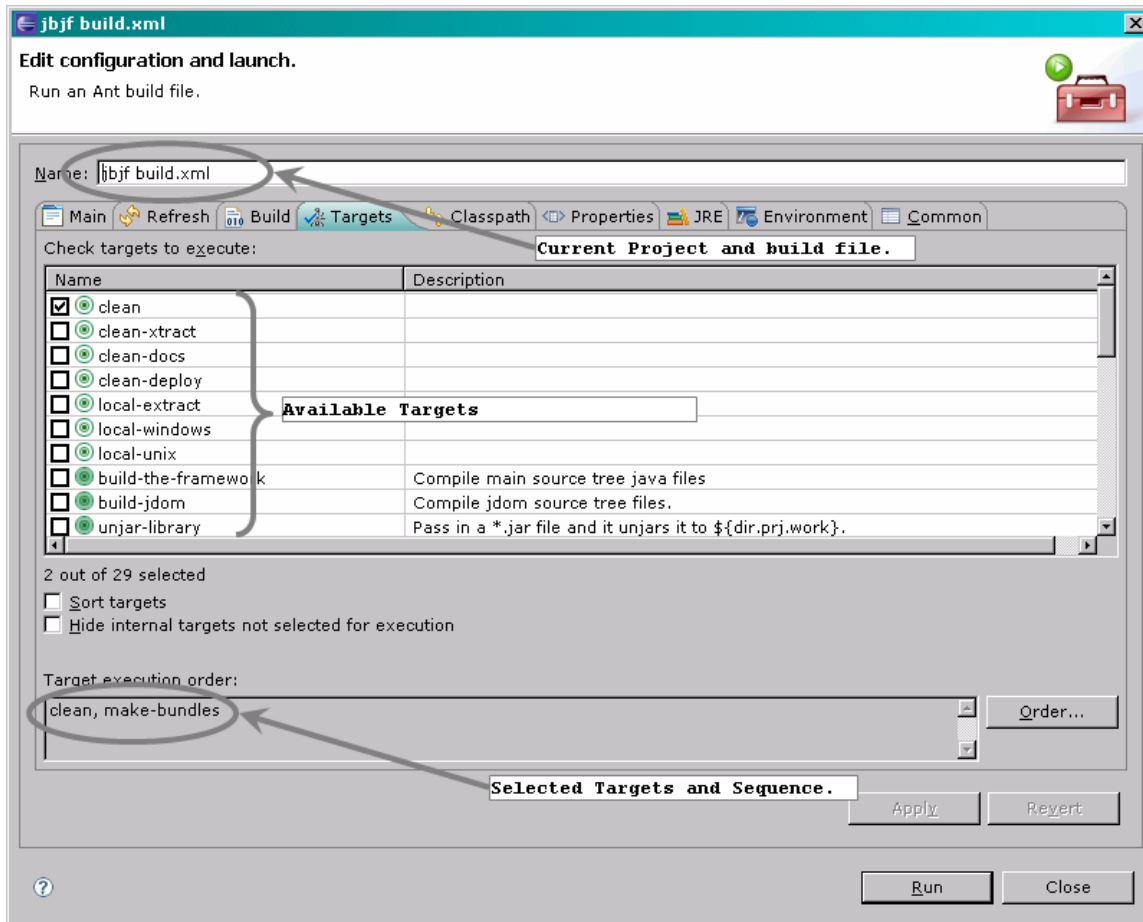
Console Configuration

The traditional approach is to compile our code into jarfiles then write script files that setup a CLASSPATH and call into the main() method of the Java class. To work through this section you should be familiar with Ant build files and scripting languages such as Windows BAT/CMD, Linux SH/BASH.

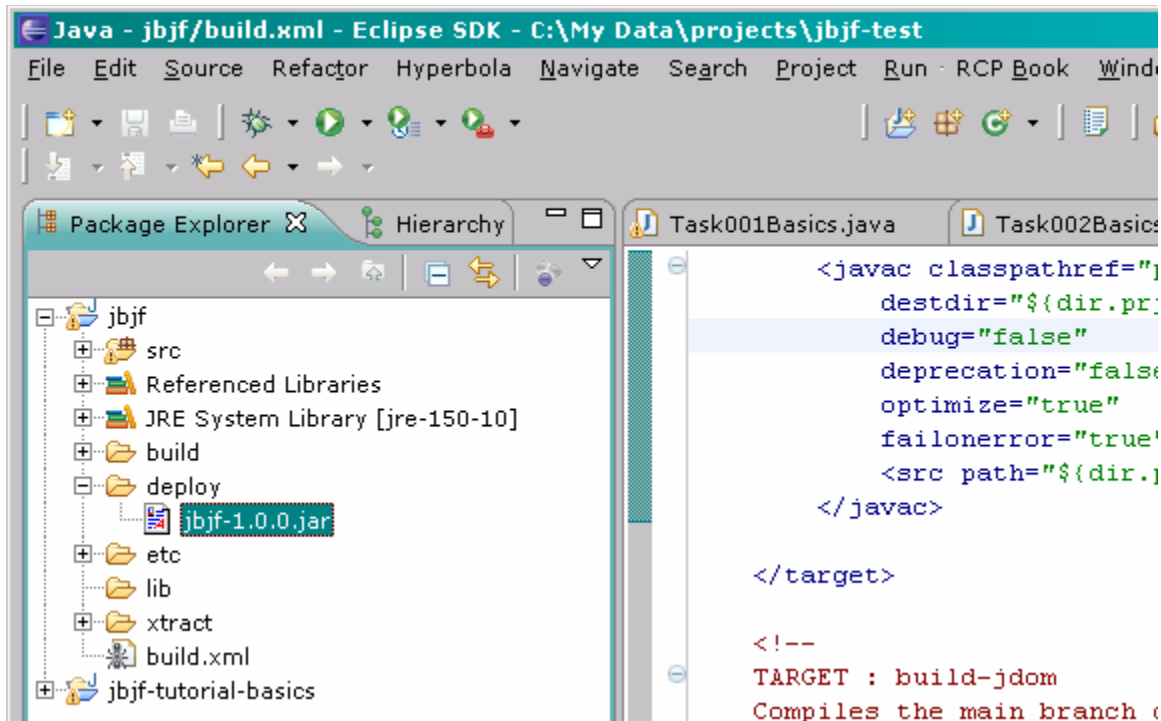
If you are currently using a source package with your tutorials, you'll need to work through this section. If you're already using a JBJF binary package, you can skip to the next section.

Start by building the jbjf into a jarfile. Move to the jbjf Project and right-click on the build.xml file contained at root of the Project directory. Select the Run As -> Ant Build... This brings up the Ant build dialog. Depending on your Eclipse preferences/setting, you should see the Targets Tab. You need to select two targets in the following order, by default the make-local target will already be selected. As such, you need to uncheck it, then check the clean and make-bundles to get the proper sequence:

- ✓ clean
- ✓ make-bundles

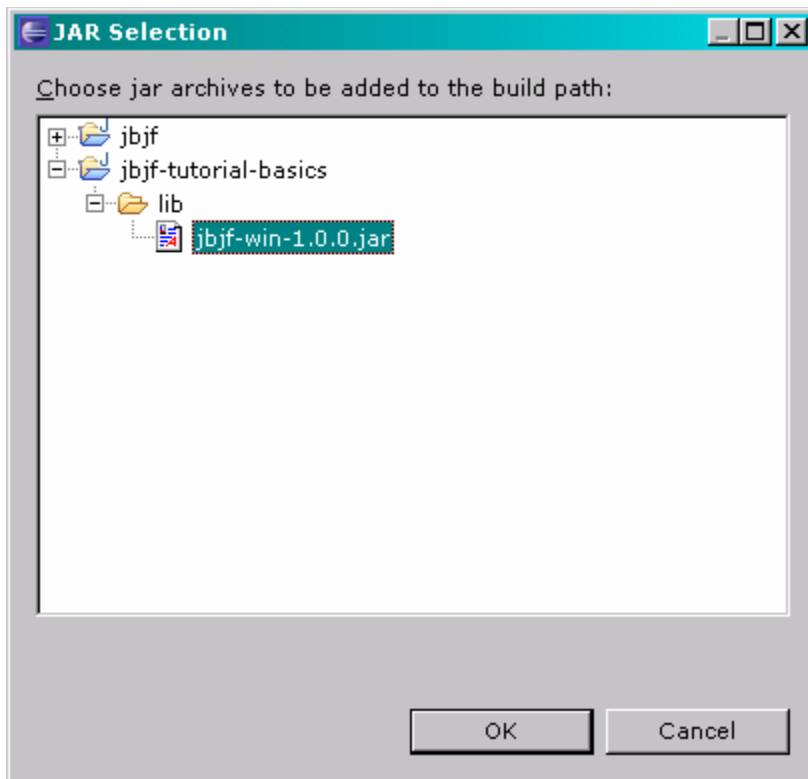


Click the Run button to have Ant compile and build the jarfiles. The make-bundles Ant target builds bundled jarfiles, the jbjf code along with all the supporting libraries for jbjf. There are multiple jarfiles for the Windows and *nix platforms. When working with the jarfile, make sure you copy the correct platform file. When the build completes the jbjf-<platform>-<ver>.jar file will be in the ./deploy directory of the jbjf project. You may have to select the jbjf project and press the F5 key to refresh the project. This will reveal the ./deploy directory so you can view the jbjf-<platform>-<ver>.jar file.

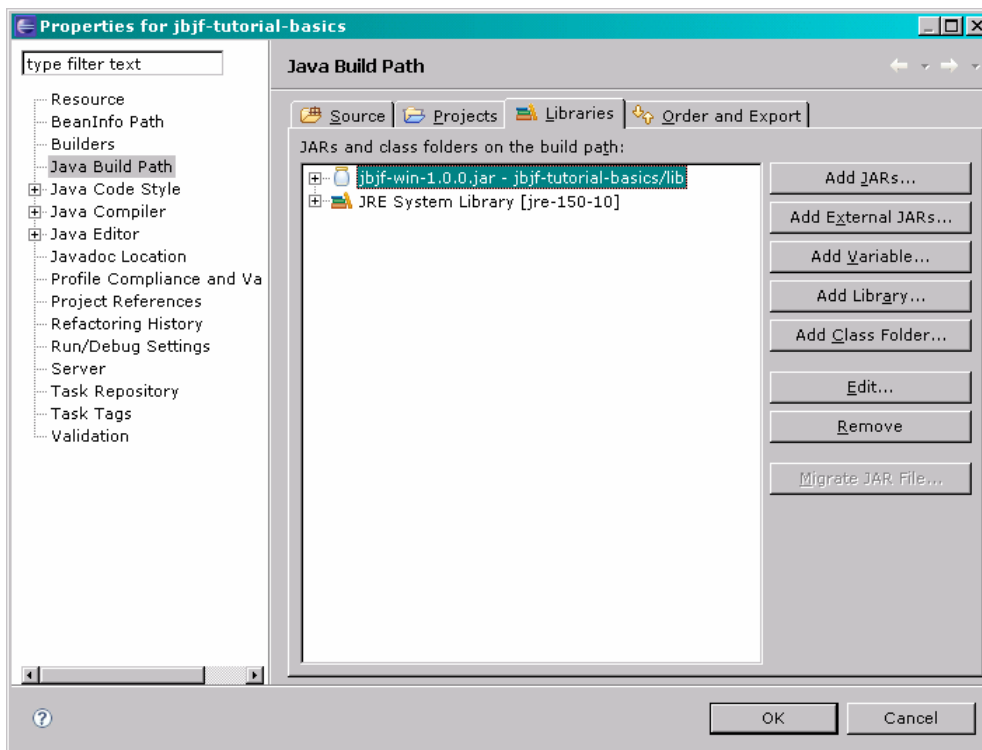


Copy the jbjf-<platform>-<ver>.jar file over to the \${tutorial-project-dir}/lib. From here, we'll be building a jarfile with our tutorial code in it, then use both jarfiles in the script files. When using the Ant build.xml file contained within the \${tutorial-project-dir}, make sure the jbjf-<platform>-<ver>.jar file is placed within the \${tutorial-project-dir}/lib directory. The tutorial Ant build file expects the jbjf-<platform>-<ver>.jar file to be in the ./lib directory of the tutorial project directory. Otherwise, adjust the Ant build.xml file to coincide with the correct location.

Finally, we need to swap out the source JBJF Project with the new jbjf-<platform>-<ver>.jar file. Right-click on the \${tutorial-project-dir} and select Properties from the pop-up menu. In the Properties dialog click the Java Build Path item. Click the Projects tab, select/highlight the jbjf project and hit the Delete key. Now click the Libraries tab, click the Add JARs button and locate the jbjf-<platform>-<ver>.jar file in the tutorials ./lib directory.



Click the OK button to add the jbjf-<platform>-<ver>.jar file to the tutorials project reference libraries.



Click the OK button to close out of the Properties dialog.

Before we can build the tutorials project with Ant, we need an Ant build.xml file. The following is a simple build.xml file that will do the job:

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
USAGE:
.....
Builds and packages the XML Batch Framework for a Java batch job.
-->

<project name="jbjf" default="make-local" basedir=".">
  <description>Author: Adym S. Lincoln</description>
  <description>Created: Nov 28, 2006</description>
  <description>Copyright: ©2006 Adym S. Lincoln ALL RIGHTS RESERVED</description>
  <description>File used to build and package the Java Batch Job Framework</description>

  <property name="dir.prj.base"           value="." />
  <property name="dir.prj.src"             value="${dir.prj.base}/src" />
  <property name="dir.prj.libs"            value="${dir.prj.base}/lib" />
  <property name="dir.prj.xtract"          value="${dir.prj.base}/xtract" />
  <property name="dir.prj.etc"             value="${dir.prj.base}/etc" />
  <property name="dir.prj.deploy"          value="${dir.prj.base}/deploy" />
  <property name="dir.prj.jdom"            value="${dir.prj.base}/jdom" />
  <property name="dir.prj.build"           value="${dir.prj.base}/build" />
  <property name="dir.prj.docs"            value="${dir.prj.base}/docs" />
  <property name="dir.prj.api"             value="${dir.prj.docs}/api" />

  <property name="cvs.pkg.main"            value="jbjf-basics-tutorial" />
  <property name="cvs.pkg.src"             value="${cvs.pkg.main}/src" />
  <property name="cvs.pkg.jdom"            value="${cvs.pkg.main}/jdom" />
  <property name="cvs.pkg.etc"             value="${cvs.pkg.main}/etc" />
  <property name="cvs.pkg.bin"             value="${cvs.pkg.main}/bin" />
  <property name="cvs.pkg.lib"             value="${cvs.pkg.main}/lib" />
  <property name="cvs.pkg.test"            value="${cvs.pkg.main}/testing" />

  <property name="dir.prj.work"            value="${dir.prj.xtract}/${cvs.pkg.main}" />
  <property name="dir.prj.compile"          value="${dir.prj.work}/classes" />
  <property name="dir.prj.compile.lib"      value="${dir.prj.work}/lib" />
  <property name="dir.prj.compile.src"      value="${dir.prj.work}/src" />

  <!--
  JAR Deployment:
  Name of the final jar file for JBJF.
  -->
  <property name="jar.file.main"           value="jbjf-basics-tutorial" />
  <property name="jar.file.version"        value="1.0.0" />

  <path id="project.class.path">
    <pathelement location="${dir.prj.libs}/jbjf-win-1.0.0.jar" />
    <pathelement location="${dir.prj.work}/classes/**" />
  </path>

  <target name="clean">
    <echo message="Cleanup work environment..." />
    <antcall inheritAll="false" target="clean-xtract" />
    <antcall inheritAll="false" target="clean-docs" />
    <antcall inheritAll="false" target="clean-deploy" />

    <echo message="Delete ${dir.prj.build}/" />
    <delete>
      <fileset
        dir="${dir.prj.build}/"
        includes="**/*" />
      </delete>
    </target>

    <target name="clean-xtract">
      <echo message="Delete ${dir.prj.xtract}" />
      <delete dir="${dir.prj.xtract}" />
    </target>
  </target>
</project>
```

```

</target>

<target name="clean-docs">
    <echo message="Delete ${dir.prj.api}" />
    <delete dir="${dir.prj.api}" />
</target>

<target name="clean-deploy">
    <echo message="Delete ${dir.prj.deploy}" />
    <delete dir="${dir.prj.deploy}" />
</target>

<!--
Copy the source files from the local location into a working
directory tree.
-->
<target name="local-extract">
    <echo message="Local Extract : ${dir.prj.xtract}" />
    <mkdir dir="${dir.prj.xtract}" />
    <mkdir dir="${dir.prj.work}" />

    <copy toDir="${dir.prj.work}" overwrite="yes" verbose="yes">
        <fileset dir="${dir.prj.base}">
            <exclude name="build/**" />
            <exclude name="bin/**" />
            <exclude name="doc/**" />
            <exclude name="docs/**" />
            <exclude name="etc/**" />
            <exclude name="lib/**" />
            <exclude name="xtract/**" />
            <exclude name="logs/**" />
            <exclude name=".project" />
            <exclude name=".classpath" />
            <exclude name="build.xml" />
            <exclude name="javadoc.xml" />
        </fileset>
    </copy>

    <!--
    CR, CRLF, LF : Convert CR, CRLF, LF to the proper setting
    based on the OS. Also, replace <TAB> with spaces.
    -->
    <echo message="fixcrlf      : ${dir.prj.work}" />
    <fixcrlf
        srcdir="${dir.prj.work}"
        includes="**/*"
    />

</target>

<!--
TARGET : build-engine
Compiles the main branch of source code for all java classes
that make up the feed engine.
<change>
1.0.1; ASL; 04/04/2007; Removed the jdom dependency...
</change>
    depends="build-jdom"
-->
<target name="build-the-framework" description="Compile main source tree java files">
    <echo message="destdir      : ${dir.prj.compile}" />
    <echo message="src path      : ${dir.prj.xtract}/${cvs.pkg.src}" />
    <mkdir dir="${dir.prj.compile}" />
    <javac classpathref="project.class.path"
        destdir="${dir.prj.compile}"
        debug="false"
        deprecation="false"
        optimize="true"
        failonerror="true">
        <src path="${dir.prj.xtract}/${cvs.pkg.src}" />
    </javac>

</target>

```

```

<target
  name="make-local"
  depends="clean,local-extract,build-the-framework">

  <antcall inheritAll="false" target="jar-the-project">
    <param name="jar.file.name" value="${jar.file.main}.jar" />
  </antcall>

</target>

<!--
TARGET : jar-the-engine
Packages the classes/ into a single JAR file for deployment.
-->
<target name="jar-the-project">
  <echo message="deploy dir: ${dir.prj.deploy}" />
  <echo message="destfile : ${dir.prj.deploy}/${jar.file.engine}" />
  <echo message="basedir : ${dir.prj.compile}" />
  <echo message="manifest : ${dir.prj.compile.src}/manifest" />
  <mkdir dir="${dir.prj.deploy}" />

  <jar
    destfile="${dir.prj.deploy}/${jar.file.name}"
    basedir="${dir.prj.compile}"
    includes="*"
    update="true">
  </jar>

</target>

</project>

```

Create a new file in the tutorials project at the root of the project and paste the above build.xml file into the file. Using the same technique as the JBJF build, compile and build the tutorial project...right-click on the build.xml file and select Run As -> Ant Build... Then select the clean and make-local in that order. Once the build completes, you should have a jarfile called jbjf-basics-tutorial.jar file in a ./deploy directory.

Now let's focus on the script files that will run our batch job. For this first tutorial we'll develop a single script file and since I'm on Windows I'll be using BAT/CMD scripting. Conversion of BAT/CMD into SH/BASH is simple enough. To run a java class from a command line we really only require a few items:

- ✓ Java Class name
- ✓ CLASSPATH
- ✓ Command line arguments

For Windows, you'll need to use the "\" in place for directory path separators. One very big gripe I have with Windows is this particular issue. Not only does the "\" cause you to rewrite the code when you port it to BASH/SH, but it's not consistent between versions of Windows. I've found some version of Windows only require a single "\" into Java, while other versions require two, "\\". Anyway, be attentive to this issue as you may encounter an exception related to this.

The following is a sample BAT/CMD script file that will allow you to run the tutorial code:

```

rem *
rem *****
rem Simple command console script file to run a tutorial batch job.
rem *****
rem *
set COMMAND_CLASS=org.jbjf.core.DefaultBatch
@echo off

rem Capture the current CLASSPATH variable...
rem
set tmpCLASSPATH=%CLASSPATH%

rem Set our CLASSPATH to the needed jarfiles...
rem

```

```

set CLASSPATH=.;..\lib\jbjf-<platform>-<ver>.jar;..\deploy\jbjf-basics-tutorial.jar;%CLASSPATH%

java -cp "%CLASSPATH%" %COMMAND_CLASS% "definition=..\etc\jbjf-basics-tutorial.xml" "name=jbjf-
tutorial-basics"

if errorlevel 1 goto failed

:successful
    @echo on
    @echo Feed has completed
    @goto done

:failed
    @echo on
    @echo Feed has failed
    @goto done

:end-if-usage

:done
@echo off
set CLASSPATH=%tmpCLASSPATH%
@echo on

```

Create a new file called run-tutorial.bat in the ./bin tutorial directory. Paste the above code into the file and save the file. You need to edit the code and replace the <plat>-<ver>.jar with the correct platform and version for the jarfile. Next, you'll need to adjust the log4j setting in the jbjf-basics-tutorial.xml file, change the ./etc/log4j.properties to ../etc/log4j.properties. Next, adjust the ./etc/log4j.properties file, set the ./logs/jbjf-basics-tutorial.log to ../logs/jbjf-basics-tutorial.log.

To run this script file open up a command prompt console and change the directory to the \${jbjf-tutorial-dir}/bin directory. Do a dir command to get a file list...your script file should be on the list. Now just execute the script file. If all goes well, you should see output similar to the following:

```

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>run-tutorial.bat

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>rem *

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>rem *****

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>rem Simple command console script file to run a
tutorial batch job.

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>rem *****

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>rem *

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>set COMMAND_CLASS=org.jbjf.core.DefaultBatch
2007-04-16 22:22:34,423 [main] INFO org.jbjf.tutorial - Starting jbjf-tutorial-basics...
2007-04-16 22:22:34,423 [main] INFO org.jbjf.tutorial - Work Unit ... Starting ...class
org.jbjf.tasks.Task001Basics
2007-04-16 22:22:34,423 [main] DEBUG org.jbjf.tutorial - Initialize Work Unit...Start...
2007-04-16 22:22:34,433 [main] DEBUG org.jbjf.tutorial - Initialize Work Unit...Complete...
2007-04-16 22:22:34,433 [main] DEBUG org.jbjf.tutorial - Task [ Task001Basics() ]...Starting...
2007-04-16 22:22:34,433 [main] DEBUG org.jbjf.tutorial - Task [ Task001Basics() ]...Complete...
2007-04-16 22:22:34,433 [main] INFO org.jbjf.tutorial - Work Unit ... Complete ...
2007-04-16 22:22:34,433 [main] INFO org.jbjf.tutorial - Finishing jbjf-tutorial-basics...
Feed has completed

C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>

```

When you run this script you may encounter the following exception:

```

<snip>
C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>set COMMAND_CLASS=org.jbjf.core.DefaultBatch
log4j:ERROR Could not read configuration file [./etc/log4j.properties].
java.io.FileNotFoundException: .\etc\log4j.properties (The system cannot find the path specified)
    at java.io.FileInputStream.open(Native Method)

```

```
at java.io.FileInputStream.<init>(Unknown Source)
at java.io.FileInputStream.<init>(Unknown Source)
at org.apache.log4j.PropertyConfigurator.doConfigure(PropertyConfigurator.java:297)
at org.apache.log4j.PropertyConfigurator.configure(PropertyConfigurator.java:315)
at org.jbjf.util.APILog4j.<init>(Unknown Source)
at org.jbjf.core.AbstractBatch.initBatch(Unknown Source)
at org.jbjf.core.AbstractBatch._runBatch(Unknown Source)
at org.jbjf.core.AbstractBatch._main(Unknown Source)
at org.jbjf.core.DefaultBatch.main(Unknown Source)
log4j:ERROR Ignoring configuration file [./etc/log4j.properties].
log4j:WARN No appenders could be found for logger (org.jbjf.tutorial).
log4j:WARN Please initialize the log4j system properly.
Feed has completed
```

```
C:\My Data\projects\jbjf\jbjf-tutorial-basics\bin>
</snip>
```

To correct this, edit the JBJF Batch Definition file and adjust the log4j properties file directory path. When running the tutorial from Eclipse's Launch Facility, the working directory is `${project-dir}`. But the console command script file runs from `${project-dir}/bin`. Thus, the log4j properties file is actually at `../etc/log4j.properties`.