Java Batch Job Framework

User Guide

Author: Adym Lincoln, Java Batch Job Framework

Acknowledgments

Apache Software Foundation - This product includes software developed by the Apache Software Foundation (http://www.apache.org/). We would like to acknowledge the terrific work the Apache Software Foundation has provided.

Oracle - This product includes software developed by the Oracle (Sun Microsystems) (http://www.sun.com/). We would like to acknowledge their contributions to the Open Source initiative.

Bouncy Castle - The JBJF utilizes the Bouncy Castle encryption library and we would like to acknowledge their great work.

Eclipse - We use the Eclipse Development environment for all the JBJF projects. We would like to acknowledge their great work.

OpenOffice.org – The very document you're reading was created with OpenOffice products. We would like to acknowledge the great work they have done and the excellent products this organization puts out.

What is JBJF?

Java is tantamount with the web and the internet. Countless websites use Java and J2EE to process and store transactions related to any number of different needs. Websites will capture data and process it using Java/J2EE to a backend database for storage. These websites exist in all kinds of different market verticals including retail, manufacturing, insurance, financial markets and banking. Yet, the majority of the website's processing is still handled by batch oriented processing. Batch processes running behind the scenes, are reading and processing data from the website's database or a mirror database. Frameworks like Struts and Ruby on Rails help standardize websites and wrap common operations to make websites more stable, faster and ultimately more reliable. Batch processing has few frameworks and few standards. The lack of standards or frameworks for batch processing leads to a batch process environment using mixed technologies, both open source and commercial. The mixture in turn causes maintenance issues, learning curves, vendor involvement and increases difficulty in the production environment.

The Java Batch Job Framework, hereinafter referred to as JBJF, provides a Java component framework that you can write Java batch processes with. Because it's a component framework, the JBJF enforces a standard approach to creating Java batch processes, hence implementing a pattern approach to batch process design. This standardization leads to reuse, easier maintenance, shorter learning curves and faster development.

JBJF is based on two essential concepts, divide and conquer and a task-list concept. The developer analyzes the batch process they need to implement and breaks that batch process down into individual tasks (divide and conquer). Each task can then be programmed into one or more JBJF Tasks. You then sequence the Tasks in the proper order (task-list). The collection and sequence of tasks constitutes the batch process. Using JBJF, a developer will sub-class key Java classes from JBJF to create tasks. Once all your tasks are written, you then define the batch process using an XML configuration file, commonly referred to as the JBJF Batch Definition file, or simply definition file. The JBJF Definition file is written in XML, with elements that are self-documenting. Thus, developers with little or no XML knowledge can learn at an accelerated pace. Within the Batch Definition file, you define the sequence of the Tasks (task-list) to implement the batch process. Once all the pieces are coded, you compile, package and run your job using a standard Java command line syntax or connect it to your batch scheduler.

As you'll see in this user guide, building a JBJF job stream is easy, simple and fast. The reusable concept stretches beyond JBJF and represents the real ROI for JBJF. Tasks developed for one JBJF job will be usable in other JBJF jobs. As more tasks are developed you can start to share, reuse even publish these tasks in a code repository. Because we use Java, the library is Platform independent, thus JBJF works on Unix, Windows or almost any OS that runs a JVM.

JBJF comes packaged with basic services such as:

- Logging (log4j)
- Email
- Zip/GZip/Tar Archiving
- FTP
- Encryption
- JDBC Database connections
- Out-of-Box Tasks

Many of these services are optional, controlled by the inclusion or removal of key XML elements in the JBJF batch definition file. There are also key attributes you can use to control activation and deactivation of key tasks and components.

JBJF was developed with the following goals:

- Platform independence. Jobs can be migrated between Unix and Windows with little to no modification. As a developer, you no longer need to worry about any impact of switching platforms.
- Promotes reuse. Because functionality is wrapped within a task that adheres to a common Interface within JBJF, another JBJF batch process can easily copy and integrate that task, in many cases the task can be used as-is by simply changing the XML elements that feed that task. Over time, a library of tasks can be built and stored in a code repository for easy publication and use.
- Shorten development time for Java based batch processes. By abstracting the majority of parameters from the batch process to the JBJF XML Batch Definition file, we decrease the amount of code overall. Extension of a single class (AbstractTask) to handle each step in any batch process means all task sub-classes are developed in the same fashion, same manner, thus coding becomes easier and faster. Reuse of functioning tasks means less testing and more reliability.
- Exponential Learning Curves – Since all Tasks are extended from the same Interface, each batch process becomes easier, faster and more reliable. As a developer learns how to write one Task, he is learning how to write all tasks.
- Faster/Shorter Test Cycles – Tasks written and in use in one process can be used in another. As such, testing of the new process is easier, since the focus of the testing can be centered on the new Tasks only.

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.

Overview

As already mentioned in the beginning, the JBJF operates on two primary concepts, divide and conquer and a task list. While not essential to developing JBJF batch processes, understanding these two aspects will expedite many unknowns when you start coding. See the Essential Concepts section for complete details.

JBJF is defined and controlled via an XML file called a JBJF Batch Definition file. Each batch process will have a separate JBJF Definition file that will provide parameters for database connections, ftp definitions, SQL statements, export definitions, directories, general purpose and custom parameters that the batch process needs. The JBJF definition file will also include a list of Task Definitions (ITask) classes that will be used in the batch process. Finally, a Process Sequence will be in place that defines what order the Tasks are run in. As you'll see, this XML file is simple to understand, easy to control, easy to change and the elements are named to be self-explanatory.

Software Architecture

The architecture of the Java Batch Job Framework is built on top of series of open source components; see the References and Acknowledgments for more information:

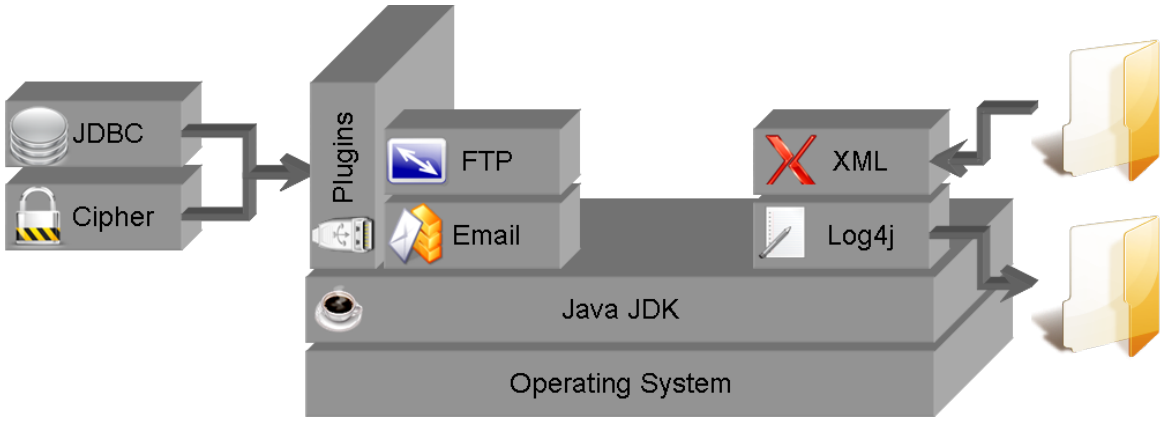
- JAXB XML
- Bouncy Castle Encryption
- Apache common networks
- Apache Log4J
- JDBC drivers
- Sun Microsystems' Email Library

Component	Class/Category	Description/Comments
JAXB/XML	Java Library	Sun Microsystems Java XML Binding Library.
Cipher - Bouncy Castle	Java Library	[1]This is an open source encryption library.
FTP - Apache Common Networks	Java Library	Apache Software Foundation's implementation of a FTP client that can transfer files to/from any host platform. This was chosen as it works with both Windows and Unix platforms and seems to provide the broadest application at the time of development.
Log4j - Apache Log4J	Java Library	Apache Software Foundation's industry standard implementation of a logging component that can do a variety of logging options.

[1] Bouncy Castle can be modified to utilize a single encryption method. The current Bouncy Castle library contains all the different methods. JBJF only uses the DES portion. We may be able to scale down the Bouncy Castle library to just the DES components.

JDBC - JDBC drivers	Java Library	Varies with databases. The current JBJF focused on Oracle, but there are sufficient XML sub-elements to handle most SQL based database engines. Subsequent versions will expand into other databases such as SQL Server, MySQL, and ODBC.
Email	Java Library	Sun Microsystems email library.

The following diagram illustrates how the architecture is structured:



Plugins are a custom extension architecture introduced in JBJF 1.3.0. The Plugins architecture provides the extensibility of JBJF for JDBC Databases and Cipher (Encryption/Decryption) services. In reality, Plugins allow you to customize JBJF in any way you see fit. See the Plugins User Guide and Tutorials for complete details.

Philosophy - Rethinking Batch Job Design

JBJF provides a component framework to develop batch processes, allowing you (the developer or architect) to build a batch processing landscape. But JBJF also requires a different approach, a different philosophy you might say, to batch process development. To effectively use JBJF you need to change your thinking about batch process development, specifically how to design the individual Tasks.

A traditional batch process design takes a sequential top-down approach. We typically grab the data at the start (top) of the batch process, apply business logic, maybe do summations, extract the results to a file and finally deliver (transfer/FTP/Email) that file to someone or some entity. All this happens within a single code structure that can range from primitive scripting (DOS BAT or Bash Shell) to intelligent scripting (Perl, PHP) to complied code like C/C++ or some mixture. Typically, the individual batch process steps get packaged into individual methods or procedures designed to fit each step in the batch flow:

- readDatabase()
- applyCalulations()
- doSummations()

- doExtract()
- doFTP()

The main procedure then just runs each step to carry out the batch process. This approach works ok, and certain steps may even be reusable. You can copy code to the next batch process, but you need to change the methods to meet the needs of the new requirements. Also, if the batch process requirements have changed in any way, you need to adjust/add/remove methods to fit the new job stream. This approach also means full testing of the new process.

JBJF also has a sequential characteristic; a task-list is executed in a given sequence. Each task is provided a list of resources from the Batch Definition file that it needs to complete the work. As each task runs, it collects the resources from the JBJF Batch Definition file, resolves resources and runtime plugins and then runs the task. It's resource management that is the key difference between traditional batch process development and JBJF. It is also resource management that is the new philosophy you need to learn to effectively use JBJF. You no longer write a process method to select rows from an Oracle database using SQL, you write a Task that accepts a database definition resource, an sql definition resource and runs the SQL against the database. You don't care what Database platform (SQL Server, Oracle, MySQL) is being used, let the database definition 'define' that for you. Your SQL Select Statement is 'defined' in the sql definition and again you don't care what database it gets runs against, just the one in the database definition.

If we take our initial batch process:

- Read data from a database
- Apply Calculations
- Summations
- Extract/Export
- FTP

And we package each step as an individual class/task:

- SQLReadDatabase
- ApplyCalculations
- DoSummations
- DoExtract
- DoFTP

Now we can develop a parent class (MyBatchJob) that will run each of the individual Task classes (runTask()) method, thus carrying out the batch process. As tasks run, they utilize resources from the JBJF Batch Definition file, specifically Java class objects created and stored when the JBJF Batch Definition file gets parsed. Because the resources are created from XML elements in the JBJF Batch Definition file, it's easy to change a particular task, by simply changing the XML element data being supplied to the task. A new batch process can be created by simply creating a new JBJF Batch Definition file.

And while not all Tasks can be reused, you'll find many of them can … with the right resource management design.

The key difference here is the traditional design doesn't really focus on reuse. Steps are developed for a single use, specific to the process requirements. Yes you can use ${ENV} variables to make steps more flexible, but ${ENV} usage integrates your process to the Operating System platform, making it difficult to migrate to a different OS platform. The JBJF design directs you to have your task collect it's resources, parameters and variables from the JBJF Batch Definition file. Thus, tasks are written in a fashion that encourages the use of parameters without binding the process to the OS. Reuse is accomplished by simply changing the resources and parameters. As developers become better at JBJF design, they also find better ways to write Tasks in a more reusable fashion. And finally, the more reuse you utilize, the smaller and shorter your testing becomes, hence a measurable Return On Investment for JBJF.

How JBJF Helps

So, how does JBJF help me address the batch process environment? Well, we've already pointed out a number of helpful benefits in the Philosophy section:

- Abstraction from the OS platform.
- Reuse of Tasks.
- Smaller and shorter Testing cycles

In addition to the above lists, JBJF also provides the following:

- Standardized development – Because all Tasks are implemented from a single Interface, every Task is structurally the same. As developers and designers work with one Task, they are learning how to work with all Tasks. While JBJF is not a standard, JBJF provides a foundation for standardized development. JBJF has two core classes that you extend or implement in order to create a batch process, AbstractProcess and AbstractTaskRuntime. Thus, JBJF helps you immediately because it is "simple". With only two classes at the core, it's easily digested and small in scope.
- Plugins – The introduction of plugins means you can now extend JBJF using simple plugins developed outside the JBJF framework. Then drop those compiled plugins into the "plugins" directory and JBJF will load them during startup. Use the plugins-definitions in the JBJF Batch Definition file to link those plugins to your batch process. And finally resource those custom plugins into your Tasks.
- Common Services – JBJF comes out of the box with a number of common services utilized by almost all batch processing. Email, logging, encryption and JDBC database management are pre-built into JBJF and ready to use with your tasks.

Essential Concepts

Without question, this is probably the most important chapter in the guide. JBJF has some essential concepts that are vital to understanding and working with the JBJF. In this section we'll discuss these concepts and outline their importance and usage.

Divide and Conquer – Task List Concept

While a bit pugnacious, the Divide and Conquer is a crucial concept when you begin JBJF design. Also referred to as a Task List Concept, the idea is instead of seeing your batch process as a single stream, see it as a series of individual Tasks. Break down your batch process into smaller and smaller segments. The level of breakdown will vary, depending on each batch process. Obviously too many segments becomes self-defeating. For example, a Task that simply sets a string/text value to all lowercase is too small to be of any value. While the reverse also holds true, a Task that does too much becomes cumbersome and not reusable. However, large Tasks will inevitably be implemented for special purposes.

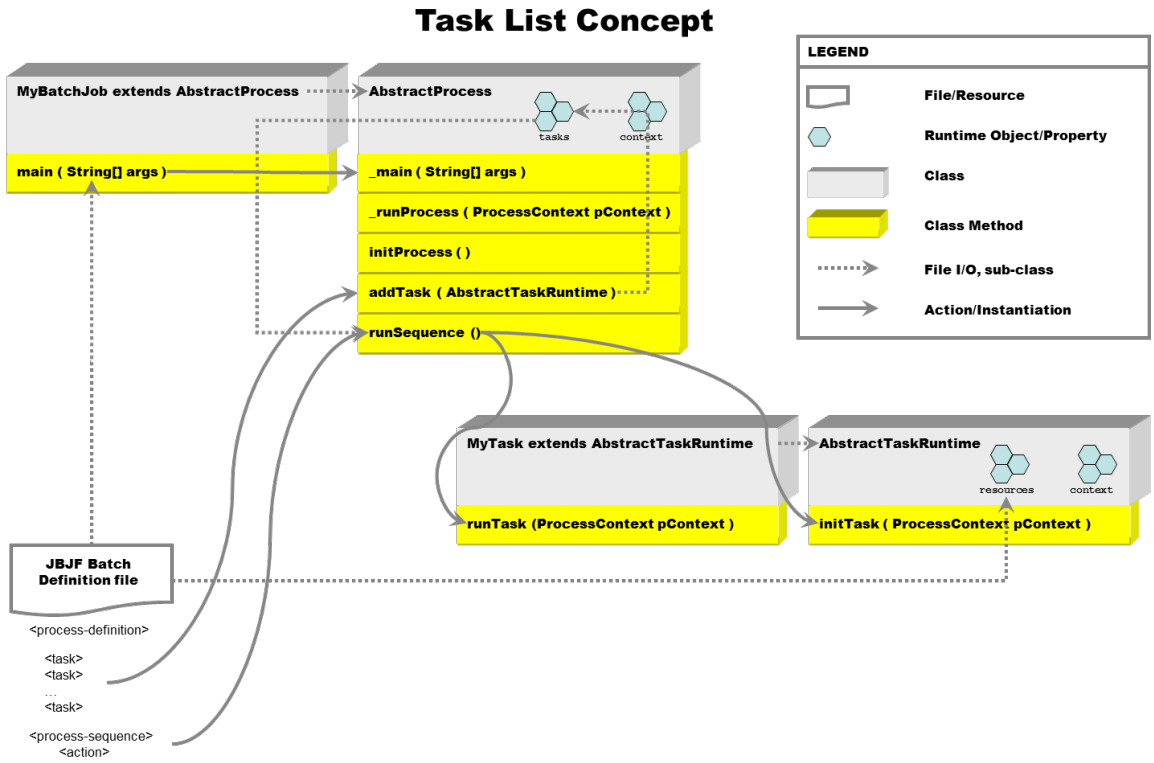
Essentially a task is a small, single step, representing part of a larger process or batch flow. You use the divide and conquer strategy to come up with the list of Tasks. Tasks are generally simple and singular in nature … i.e. copying a file, ftp a file, run an SQL statement, export a record set, etc... The tasks are managed by a parent batch process that handles one or more tasks. The sequence of all tasks, the task list, constitutes the batch process.

The task list is controlled and managed through JBJF using the AbstractBatch class. You have the choice of extending this class to create your own Batch level control class, or you can use the built-in DefaultBatch included with JBJF. The out-of-the-box functionality for JBJF is to iterate through all the Task sub-classes and run the following method sequence:

- initTask ()
- preTask ()
- runTask ()
- postTask ()

Each task within your batch process will extend the AbstractTaskRuntime or implement the ITaskRuntime interface, essentially forcing the implementation of the runTask() method. You'll typically override the initTask(), pre-processing any resources your task needs. The preTask() and postTask() are optional and provide flexibility in your Task design. The JBJF Framework then iterates through all the assigned Task sub-classes listed in the JBJF Batch Definition file, running the runTask() method, thus implementing the batch process. The traditional choice is to extend AbstractTask, thus inheriting the default initTask() implementation and class property management. More about this in the Basic Tutorials.

The task list concept is illustrated in the following diagram:



A brief overview of JBJF:

- The entry point from the command line is the traditional Java main() method. The main() method is implemented in your batch process class (AbstractProcess sub-class), MyBatchJob in the above diagram. The JBJF Batch Definition file is provided via the command line arguments. Command line arguments are passed in as key=value pairs and stored in the job stack (ProcessContext) as a key to a value/object.
- The command line arguments are forwarded from the AbstractProcess sub-class, MyBatchJob, as is. The AbstractProcess._main() expects a JBJF Batch Definition file. This XML file is parsed and stored as a large class object (IJBJFProcessDefinition) that stores all the elements, values and parameters as individual Java class objects and collections.
- Control then moves from the _main() method to _runProcess() and initProcess() methods as the JBJF Definition file is parsed and services such as email, archiving and logging are setup and established. Part of the JBJF Definition file will contain a list of fully qualified Java class names for the AbstractTaskRuntime sub-classes that make up the task-list. Each of these sub-classes will be instantiated using Class.forName() and then added to the Task Definitions collection.
- Once XML parsing is complete, services are established and the task-list is created, control is passed to the AbstractProcess.runBatch() method. In runProcess(), the Process Sequence is run, essentially running the Task Definitions in the proper sequence, thus running the batch process.

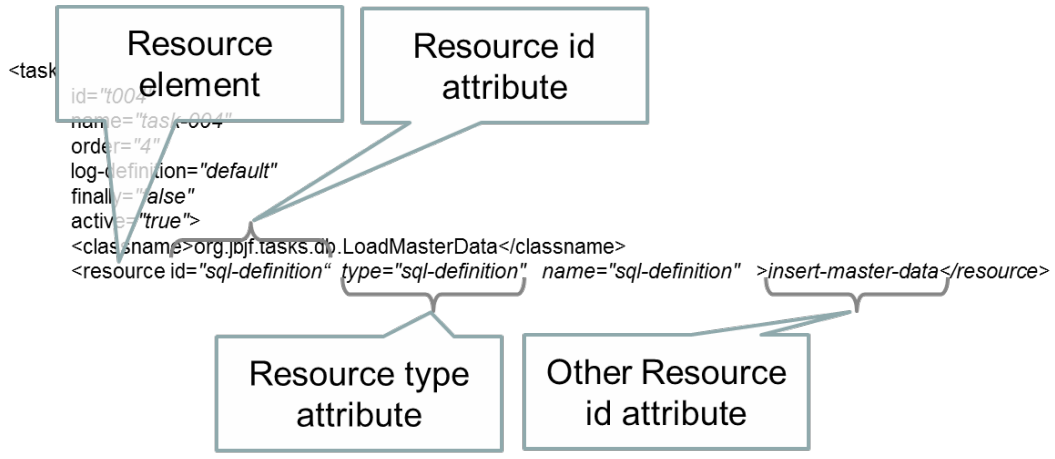
Named Resources and Resource Management

Another essential concept for the JBJF is named resources. A task within a batch process requires resources such as database connections, ftp connections, sql definitions and filesystem objects to complete its work. The JBJF Batch Definition file defines these resources as XML elements. But how do they find their way to the correct task?

Each element of the JBJF Batch Definition file comes with an id attribute. The id attribute generally needs to be unique within it's parent XML element, such as sql-definitions, ftp-definitions, database-definitions, etc…but those id values can be reused in another parent XML element. So, you can have one sql-definition with an id=id001 and a database-definition with an id=id001. You can not have two or more sql-definitions with id=id001, and you can not have two or more database-definitions with id=id001. The same applies to task-definitions, account-definitions, plugin-definitions, export-definitions, log-definitions. Pretty much any 1st tier element ending in –definitions.

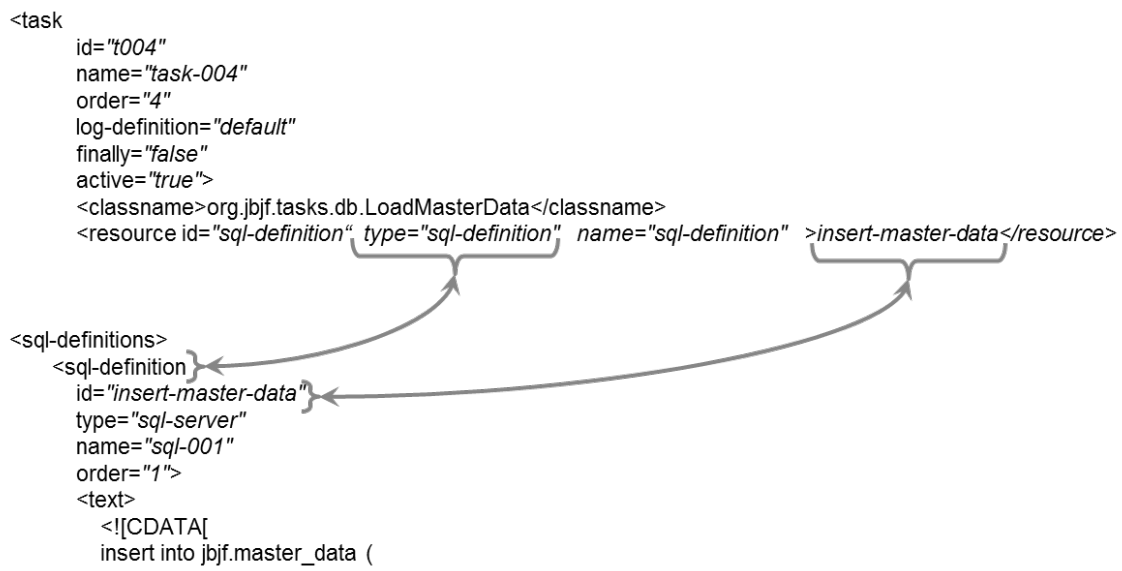
There are some id attributes that must be unique across the entire JBJF Batch Definition file. These are generally the plugin-definitions. To be safe, each id attribute should be unique across the entire JBJF Batch Definition file though. There are no id attributes that need to be unique across JBJF Batch Definition files though. The id attribute is free-form text and can be any length you wish, but a generally rule of thumb is to keep them short and unique. Use the name attribute for descriptions or brief indicators of usage or meaning.

A Task includes one or more resource elements, where each resource itself has a unique id attribute and a type attribute. However, it is the value node in a resource element that contains the id of the actual resource you are linking to the task. Also, the resource type attribute indicates what resource type you want (database-definition, ftp-definition, sql-definition, etc…).



When the JBJF Batch Definition file gets parsed and stored as Java class objects, the id attribute serves as a key to the individual elements such as task-definition, sql-definition, ftp-definition, database-definition, etc…, aka the Java objects. We then use the resource

elements in the task-definition to link the various resources the task will need. To make the link, the resource element uses the type attribute and the value node together. This concept is illustrated in the following diagram:



In the above example, task t004 wants a resource of type=”sql-definition” with an id of ‘insert-master-data’. Further down in the JBJF Batch Definition file are the sql-definition elements. Listed within the sql-definitions is an sql-definition element with an id=”insert-master-data”. So the link is made and the resource will get picked up in the initTask() of the t004 Task.

Required Resources

Introduced in version 1.2.0, the required resources is an optional integrity check built into JBJF. Required Resources is enabled and done at the Task level, thus you need to code your individual tasks to utilize it. It is NOT an automatic feature.

Required Resources implements a pseudo-contract between JBJF and your task. You populate the Required Resources list with the names of those <resource> XML elements that are required in order for your task to work. At this time there is no Optional Resources list built in. While the List can be populated at anytime prior to the runTask() method, it’s a best practice to do this in the Default Constructor of your Task class. This is by design, since every AbstractTask sub-class requires a Default Constructor, hence you are guaranteed that the list gets populated up front.

You then “engage” the Required Resources in your Task by coding a simple if test using the function hasRequiredResources() (built into the AbstractTask super class). The method hasRequiredResources() will compare your list of Required Resources to the current list of XML <resource> tags and returns a True/False value based on the outcome. If the function returns True, then no exception is thrown. Otherwise, an

exception is thrown indicating the resource or resources that are missing from the <task-definition> element. Should your particular Task require further processing of Required Resources, you can easily catch the exception and do further checks.

The following code snippet from the JBJF CopyFile task illustrates the Required Resources template. I've removed much of the extraneous code for brevity:

```
public class CopyFile extends AbstractTask {

    /**
     * Default constructor.  Sets up the required resources.
     */
    public CopyFile() {
        super();
        mtaskRequired = new ArrayList();
        getRequiredResources().add("source");
        getRequiredResources().add("target");
    }



    /**
     * The <code>CopyFile</code> task will expect at least two
     * resources that should be defined in the JBJF Batch Definition
     * XML file, a source filename and a target filename.  The following
     * is an example <code>CopyFile</code> task definition:
     * <p>
     */
    @Override
    public void runTask ( HashMap pjobParameters ) throws Exception {
        /*
         * Enforce the required resources...
         */
        if ( hasRequiredResources() ) {
            String  lstrSource = (String)getResources().get( "source" );
            String  lstrTarget = (String)getResources().get( "target" );
            File lfileSource = new File( lstrSource );
            File lfileTarget = new File( lstrTarget );

            // Copies the file
            FileChannel lfisInput = null;
            FileChannel lfosOutput = null;

            try {
                // magic number for Windows, 64Mb - 32Kb
                //
                int     mbCount = 64;
```

It's very, Very, VERY important that you don't declare the mtaskRequired ArrayList() in your individual class, let the AbstractTask super class manage this list. Don't ask me how I know this, just trust me.

This is a very simple (basic) implementation of the Required Resources, but it provides the foundation to enforce a simple "contract" between the JBJF Framework and your tasks.

Process Context

The JBJF also relies on a central ProcessContext that gets passed between various Task classes during the batch process. We refer to this as the Process Context, and it serves as a poor man's indexed table. The Process Context represents the sole communication channel between Tasks. Thus, should one task (task1) need to pass results to a sub-sequent task (task2), then task1 would "put" those results onto the Process Context using a String key near the end of its runTask() method. Then task2 would "get" those results off the Process Context prior to or at the beginning of its runTask(). This process can be repeated from one task to another.

Under the covers, the Process Context is a HashMap collection and because of this you need to keep a few things in mind when using it:

- Objects and results that you put onto the Process Context need a unique key across the entire batch process. Otherwise, if you put something onto the Process Context with an existing identifier, then a "replace" is done. Generally, a key can be hard-coded within the runTask() method, but this can result in a limited reuse for the Task. You'll see in some of the advanced tutorials how to use <resource> XML elements to store keys for known objects/results intended for the Process Context. Using <resource> XML elements as keys means a Task can be reused easier. You can also integrate these keys as "required resources" to ensure they are coded correctly for the Task.
- A HashMap collection also gives you the freedom to explicitly do a "replace" if you wish. Simply use an existing key and the object/results stored at that key are replaced with your new item. This can be handy if you wish to have two or more tasks use the same set of data and modify results, thus storing the results in the same space.
- You are free to do manual memory management with the Process Context. If you store a large ResultSet on the Process Context and then you wish to remove that object, you are free to use the removeItem() and pull the object off the Process Context and let it get GC'ed by the JVM.

Java Batch Job Definition file

At the start of every JBJF batch process is the JBJF Batch Process Definition file, commonly called the JBJF Batch Definition file, or simply definition file. In this chapter we'll discuss the birth of the definition file as well as some of the theory and decisions that led to its final form. While you may glance over some of the theory and decision making, we highly recommend that you read through the Structure section to get a feel for the definition file. For those already versed in XML, it's probably enough to simply open up and look directly at a Batch Definition file, since most of the XML element names are self-documenting. The primary purpose of the JBJF Process Definition file is abstraction from the Operating System, abstraction from the Databases, really abstraction in general.

Narrative

abstraction *- A mechanism and practice to reduce and factor out details so that one can focus on a few concepts at a time.*

(Courtesy of Wikipedia - http://en.wikipedia.org/wiki/Abstraction_(computer science))

For JBJF, abstraction is the basis for the Batch Definition file. Any Java batch process has a number of common elements that it will rely on to operate. Information such as the batch process name, directories, userids, passwords and logging all play an essential role in providing a batch process the resources it needs. Some resources are required by all batch processes, other resources are needed by one batch process, but not another. So, one important question is how to supply each individual batch process the necessary resources in an easy, flexible and repeatable way. The answer is to "abstract" batch process elements into a form that can be processed, stored and retrieved. For JBJF, the outcome of "abstraction" should be a conceptual form (or object) that can be utilized by a Java object. This abstract form can then be supplied to a given batch process through the command line or some other argument container, thus allowing the batch process to process the abstract form. Finally, this process can be repeated for another batch process by simply copying the abstract form and changing the values within.

Easy is a subjective term, but we have chosen XML as the easy abstract form when listing resources and data. XML requires no special tools to create them, and by naming the XML elements correctly, the XML content can be self-documenting. Flexible refers to the ability to conform to ones needs. For a JBJF batch process this implies "optional" data and parameters. For XML it means to supply, or "not" supply certain elements. Thus, XML satisfies two of our needs for an abstract form, flexibility is attained by the inclusion or exclusion of optional XML elements. Finally, repeatable is easy, as we simply copy an existing batch process's JBJF Batch Definition file and change the XML data for the new batch process. The new batch process then receives a new command line argument that points to the new JBJF Batch Definition file.

Strategy

Part of the creation of the JBJF Definition file involved what services a batch process requires. Much of this initial analysis focused on where the JBJF would be utilized in an Enterprise or Business. The final decision was that a midrange tier, Unix/Windows, servers would be the target tier for JBJF.

That said, the following services were determined as optional for any batch process running in the midrange tier. These services also represent the broadest range that most batch processes require:

- Email - SMTP Only
- File Transfers (FTP)
- Archiving
- Database Access and SQL
- Export/Extract

The services were assumed to be optional, not every batch process would need these. The following items however were thought to be required items for any batch process:

- Directories
- General Purpose
- Tasks
- Logging
- Custom

As you'll see in sub-sequent sections, just how these services get implemented as an XML element and eventually a Java class object.

Structure

XML is an excellent language for coding data. It also has an excellent structure for creating configuration files such as the JBJF Batch Definition file. The basic structure of the JBJF Definition file is the use of top-level XML elements that define/group a single service. These top-level elements are placed at the first level of the JBJF Definition file, one level in from the root element. For instance, the Directories service is grouped under the <jbjf-directories> element. Thus, individual <directory> elements contain one piece or segment of the batch process's directory tree. Subsequent services are grouped under similar elements:

- Email <process-email>
- Directories <process-directories>
- FTP <ftp-definitions>
- Export <export-definitions>
- Sequence <process-sequence>
- Tasks <task-definitions>
- General Purpose <process-parameters>
- Database <database-definitions>
- SQL <sql-definitions>

- ✔ Logs <log-definitions>
- ✔ Plugins <plugin-definitions>
- ✔ Accounts <account-definitions>

These top-level elements follow a certain naming convention, using process- as the prefix indicates something defined at the Process level. Using -definitions usually implies a Task resource. They are easy to spot and we've tried to name them properly to be self-documenting. The inner XML elements are also organized in a similar fashion, with the data and parameters being directly linked in the same genre as the top-level element.

Another key goal of the JBJF Definition file was to keep the XML depth shallow. An XML file with too many levels of XML elements becomes difficult to work in and starts to lose its "easy" labeling. As such, the JBJF Definition file goes no deeper than 3 levels, in most cases only 2 levels. The absence of any GUI to manage the JBJF Definition file means you'll be coding it by hand, thus simplicity is key.

The following is an example JBJF Batch Definition file. The XML file contains "all" the primary elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file generated by XMLSpy v2010 (http://www.altova.com)-->
<!DOCTYPE jbjf-batch-job
[
  <!ENTITY _JBJF_TEST_DB_     "db-jbjf"          > <!-- _DB_JBJF_TEST_       -->
  <!ENTITY _JBJF_DEV_DB_      "db-jbjf"           > <!-- _DB_JBJF_DEV_        -->
  <!ENTITY _BASE_DIR_         "."              > <!-- _DIR_BASE_           -->
  <!ENTITY _EMAIL_FAIL_       "jbjf.help@gmail.com"    > <!-- _FAIL_EMAIL_         -->
]>
<process-definition
    xsi:noNamespaceSchemaLocation="../jbjf-definition.xsd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <!--
    PROCESS PARAMETERS :
    These are batch level settings and configuration values.
    -->
    <process-parameters>
        <name>load-master-data</name>
        <status-code-key success="0" failure="1">jbjf-status-code</status-code-key>
        <mode>normal</mode>
    </process-parameters>

    <!--
    PROCESS EMAIL :
    Contains all email related settings and configurations for the
    batch process.  The classname is the name of the JBJF Runtime
    object that will provide the Email service to the JBJF Runtime.
    See the JBJF documentation for complete details on how this
    service is managed and customized.
    -->
    <process-email
        format="text"
        classname="org.jbjf.plugins.DefaultEmail"
        active="true">
        <email-host>
            <!--
             -->
            <user><![CDATA[jbjf.help@gmail.com]]></user>
            <password><![CDATA[link1964]]></password>
            <server>smtp.gmail.com</server>
            <port>465</port>
```

```
            <ssl>true</ssl>
        </email-host>
        <notifications>
            <email id="el-001">jbjf.help@gmail.com</email>
            <email id="el-002">adymlincoln@gmail.com</email>
        </notifications>
        <email-sender>jbjf.help@gmail.com</email-sender>
        <email-success email="jbjf.help@gmail.com">JBJF Load Master Data - Completed Successful-
ly</email-success>
        <email-failure email="&_EMAIL_FAIL_;">JBJF Load Master Data - Failed</email-failure>
    </process-email>


    <process-directories>
        <directory name="base"      id="base"   pathtype="relative">.</directory>
        <directory name="archivist" id="ark"    pathtype="relative">archives</directory>
        <directory name="log4j"     id="log"    pathtype="relative">etc</directory>
        <directory name="plugins"   id="plug"   pathtype="relative">plugins</directory>
    </process-directories>


    <task-definitions>
        <!--
        CLEAN : Cleanup any files from the previous run...
        -->
        <task
            id="t001"
            name="t001"
            order="1"
            log-definition="default"
            finally="false"
            active="true">
            <classname>org.jbjf.tasks.RemoveFiles</classname>
            <resource id="file-001" type="String" required="true" name="file-to-re-
move">.\wip\master-data.txt</resource>
        </task>

        <!--
        COPY : Copy the input file to the work-in-progress folder...
        -->
        <task
            id="t002"
            name="t002"
            order="2"
            finally="false"
            active="true">
            <classname>org.jbjf.tasks.CopyFile</classname>
            <resource id="source" type="String"   required="true" name="source">.\data\in-
bound\master-data.txt</resource>
            <resource id="target" type="String"   required="true" name="target">.\wip\master-da-
ta.txt</resource>
        </task>

        <!--
        DATA : Load the text file into a Text Document object...
        -->
        <task
            id="t003"
            name="task-003"
            order="3"
            log-definition="db-log"
            finally="true"
            active="true">
            <classname>org.jbjf.tasks.ReadTextDocument</classname>
            <resource id="job-stack-key"    type="context-key"      required="true" name="text.-
Document Key">master-data</resource>
            <resource id="file-name"        type="context-key"      required="true" name="file-
name"        >master-data.txt</resource>
            <resource id="delimiter"        type="context-key"      required="true" name="delim-
iter"        >&#x09;</resource>
            <resource id="source"           type="context-key"      required="true" name="source"
>.\wip</resource>
        </task>
```

```
        <!--
        MASTER : Load the text file document into the master_data
        database table...
        -->
        <task
            id="t004"
            name="task-004"
            order="4"
            log-definition="default"
            finally="false"
            active="true">
            <classname>org.jbjf.tasks.db.LoadMasterData</classname>
            <resource id="sql-definition"       type="sql-definition"       required="true"
name="sql-definition"   >insert-master-data</resource>
            <resource id="database-definition"  type="database-definition"  required="true"
name="&_JBJF_TEST_DB_;" >&_JBJF_TEST_DB_;</resource>
            <resource id="delimited-document"   type="delimited-document"   required="true"
name="master-data"      >master-data</resource>
            <resource id="tablename"            type="java.lang.String"     required="true"
name="database-table"   >master_data</resource>
            <resource id="plugin-cipher"        type="plugin-cipher"        required="true"
name="default-cipher"   >default-cipher</resource>
            <resource id="plugin-database"      type="plugin-database"      required="true"
name="mysql-db"         >mysql-not-persistent</resource>
            <resource id="truncate"             type="flag"                 required="false"
name="database-table"  >true</resource>
        </task>

    </task-definitions>


    <process-sequence>
        <action concurrent="false" category="task" name="task-001" id="t001" thread="false">
            <action-name>Run Task 1</action-name>
            <icon>./etc/task-001.png</icon>
        </action>
        <action concurrent="false" category="task" name="task-002" id="t002" thread="false">
            <action-name>Run Task 2</action-name>
            <icon>./etc/task-001.png</icon>
        </action>
        <action concurrent="false" category="task" name="task-003" id="t003" thread="false">
            <action-name>Run Task 3</action-name>
            <icon>./etc/task-003.png</icon>
        </action>
        <action concurrent="false" category="task" name="task-004" id="t004" thread="false">
            <action-name>Run Task 4</action-name>
            <icon>./etc/task-004.png</icon>
        </action>
    </process-sequence>


    <database-definitions>
        <database-definition
            persistent="false"
            name="big-task-db"
            default="false"
            id="db-jbjf"
            plugin-cipher="default">
            <type>mysql</type>
            <user><![CDATA[dc08be7cb3c042d7]]></user>
            <driver>com.mysql.jdbc.Driver</driver>
            <database>jbjf</database>
            <client>jdbc:mysql</client>
            <port>3306</port>
            <password><![CDATA[2f95ab3434968b5bc0beba6e01893810]]></password>
            <server>localhost</server>
        </database-definition>

    </database-definitions>


    <log-definitions>
        <log-definition
            category="org.adym.batch"
```

```
            name="default-name"
            id="default"
            default="true">./etc/log4j.properties</log-definition>
    </log-definitions>


    <sql-definitions>
        <sql-definition
            id="insert-master-data"
            type="sql-server"
            name="sql-001"
            order="1">
            <text>
                <![CDATA[
                insert into jbjf.master_data (
                     gender                /*  p001    */
                    ,first_name
                    ,middle_name
                    ,last_name
                    ,street_address_1      /*  p005    */
                    ,city
                    ,state
                    ,zip_code
                    ,country
                    ,email                 /*  p010    */
                    ,phone
                    ,birthdate
                    ,card_make
                    ,account
                    ,ccv                   /*  p015    */
                    ,expire_date
                    ,job_title
                    ,entered_on
                    ,entered_by
                    ,modified_on
                    ,modified_by
                )
                VALUES (
                     ?  /*  p001    */
                    ,?
                    ,?
                    ,?
                    ,?  /*  p005    */
                    ,?
                    ,?
                    ,?
                    ,?
                    ,?  /*  p010    */
                    ,?
                    ,STR_TO_DATE(?, '%m/%d/%Y')
                    ,?
                    ,?
                    ,?  /*  p015    */
                    ,?
                    ,?
                    ,CURRENT_DATE()
                    ,CURRENT_USER()
                    ,CURRENT_DATE() /*  p020    */
                    ,CURRENT_USER()
                );
                ]]>
            </text>

            <sql-parameters>
                <sql-parameter id="p001" name="p001" order="1" type="string">1</sql-parameter>
                <sql-parameter id="p002" name="p002" order="2" type="string">1</sql-parameter>
                <sql-parameter id="p003" name="p003" order="3" type="string">1</sql-parameter>
                <sql-parameter id="p004" name="p004" order="4" type="string">1</sql-parameter>
                <sql-parameter id="p005" name="p005" order="5" type="string">1</sql-parameter>
                <sql-parameter id="p006" name="p006" order="6" type="string">1</sql-parameter>
                <sql-parameter id="p007" name="p007" order="7" type="string">1</sql-parameter>
                <sql-parameter id="p008" name="p008" order="8" type="string">1</sql-parameter>
                <sql-parameter id="p009" name="p009" order="9" type="string">1</sql-parameter>
                <sql-parameter id="p010" name="p010" order="10" type="string">1</sql-parameter>
```

```
            <sql-parameter id="p011" name="p011" order="11" type="string">1</sql-parameter>
            <sql-parameter id="p012" name="p012" order="12" type="string">1</sql-parameter>
            <sql-parameter id="p013" name="p013" order="13" type="string">1</sql-parameter>
            <sql-parameter id="p014" name="p014" order="14" type="string">1</sql-parameter>
            <sql-parameter id="p015" name="p015" order="15" type="string">1</sql-parameter>
            <sql-parameter id="p016" name="p016" order="16" type="string">1</sql-parameter>
            <sql-parameter id="p017" name="p017" order="17" type="string">1</sql-parameter>
            <!--
            <sql-parameter id="p001" name="p018" order="18" type="string">1</sql-parameter>
            <sql-parameter id="p001" name="p019" order="19" type="string">1</sql-parameter>
            <sql-parameter id="p001" name="p020" order="20" type="string">1</sql-parameter>
            <sql-parameter id="p001" name="p021" order="21" type="string">1</sql-parameter>
             -->
        </sql-parameters>

    </sql-definition>

</sql-definitions>


<export-definitions>
    <export-definition name="xport-001" header="true" id="x001">
        <destination>/usr/apps/my-app</destination>
        <filename>big-file.txt</filename>
        <format>ascii</format>
        <delimiter>|</delimiter>
        <fields>
            <field type="String" order="1" name="field-001" id="f001">field001</field>
            <field type="String" order="2" name="field-002" id="f002">field002</field>
        </fields>
    </export-definition>
    <export-definition name="xport-002" header="false" id="x002">
        <destination>/usr/apps/the-apps</destination>
        <filename>the-big-file.txt</filename>
        <format>ascii</format>
        <delimiter>|</delimiter>
        <fields>
            <field type="int" order="1" name="field-001" id="f001">field001</field>
            <field type="String" order="2" name="field-002" id="f002">field002</field>
        </fields>
    </export-definition>
</export-definitions>

<plugin-definitions>
    <plugin-definition
        id="default-cipher"
        name="default-cipher"
        default="true"
        type="org.jbjf.plugin.IJBJFPluginCipher"
        active="true">
        <classname>org.jbjf.services.impl.DefaultCipher</classname>
    </plugin-definition>

    <plugin-definition
        id="mysql-not-persistent"
        name="mysql-not-persistent"
        type="org.jbjf.plugin.IJBJFPluginDatabase"
        default="false"
        active="true">
        <classname>org.jbjf.services.db.MySQLNonPersistentService</classname>
    </plugin-definition>

</plugin-definitions>

<account-definitions>
    <account-definition type="standard" name="acct-standard"
        default="true" id="acct-001" plugin-cipher="default-cipher">
        <password><![CDATA[acct-pwd-001]]></password>
        <account-number><![CDATA[4444444444444444444]]></account-number>
        <user><![CDATA[acct-usr-001]]></user>
        <ccn><![CDATA[771]]></ccn>
        <expiration><![CDATA[08/2011]]></expiration>
    </account-definition>
    <account-definition type="credit" name="acct-not-standard"
```

```
            default="false" id="acct-002" plugin-cipher="default-cipher">
            <password><![CDATA[acct-pwd-002]]></password>
            <account-number><![CDATA[55555555555555555555]]></account-number>
            <user><![CDATA[acct-usr-002]]></user>
            <ccn><![CDATA[772]]></ccn>
            <expiration><![CDATA[08/2012]]></expiration>
        </account-definition>
    </account-definitions>

</process-definition>
```

Element Details

In this section we list and discuss all the XML elements for the JBJF Batch Definition file.

XML Element	Description/Comments
process-parameters	This is a required element and contains a variety of sub-elements that fit within the general purpose category. The sub-elements are as follows: **name** - Name of the batch process. Used in the emails. **status-code-key** – Defines a status code for the entire batch process that defines a success and failure status code for the batch process. **mode** – A normal|legacy indicator that defines how the JBJF Framework will run the process. Normal utilizes the new process-sequence. Legacy uses the list of Tasks.
name	Contains the name of the batch process. This is currently used for email subject lines.
status-code-key	The XML element value is the key on the Process Context where the status code is stored. The success and failure attributes define the return codes for a success and failure of the batch process.
mode	A text indicator that defines how JBJF will run the batch process: normal – Uses the new process-sequence. legacy – Runs the 1.x.x way, where each Task must be ordered.
process-email	An optional element, this will define all the parameters needed to connect to the email server. Additional elements define stakeholders that get notified as well as success and failure emails.
notifications	This parental element contains a list of email recipients that need to receive an email from the batch process.
email	Represents a single recipient email address. The element has the following attributes: attachments - A Y/N character that indicates whether the recipient should receive attachments. The attachment is the archive created, so the enable-archivist will need to be Y/1 in order to create an attachment.
email-host	Contains the name of the email host server. The current version of JBJF only supports SMTP.
email-sender	Contains a real or fictitious email address that will be on the email notification as the sender. Many users will name this something close to the batch process name, then setup filters on the email client to group emails.
email-success	Provides a target email address to notify upon successful completion of the batch process. Introduced in 1.2.1 as a remedy to have an "optional" success email. There were many users who suggested that no email needs to be sent when a batch process finishes successfully...i.e. only notify me when there's a problem.
email-failure	Provides a target email address to notify upon failure of the batch process. Introduced in 1.2.1 as a remedy to have an "optional" success email. There were many users who suggested that no email needs to be sent when a batch process finishes successfully...i.e. only notify me when there's a problem.
process-directories	A parental element that contains a list of <directory> sub-elements, where each <directory> represents a relative or absolute pathway for the batch process's directory tree.
directory	Contains a single relative or absolute pathway for the batch process's directory tree. The element comes with the following attributes: - name - A unique name within the <jbjf-directory> element that gets used as a lookup/search key when you wish to retrieve this path. - addressing - A word that indicates the type of path stored. An absolute path

	is resolved as-is. A relative path will be resolved with the "base" directory appended to the front...thus, if base = /usr/apps and data = inbound, then when you fetch the data directory, you'll receive /usr/apps/inbound as a value. The following directory names are reserved by JBJF: - base - Represents the base directory path for the batch process and all "relative" addressing <directory> elements. - archivist - Represents a relative or absolute top-level path where any archive files will be stored. The archivist will create timestamp (YYYY-MM-DD-HH-MI) sub-directories for each run of the batch process. Thus, if archivist = /usr/apps/batch/archives with addressing="absolute". When you run your batch process, a new directory will be created, /usr/apps/batch/archives/yyyy-mm-dd-hh-mi/ that contains the zipfile archive.
task-definitions	Contains the tasks that will be used in this batch process.
task	A parental element that contains all the data to define a single task for the batch process. There will be a <task> element for each AbstractTaskRuntime sub-class. The <task> element has the following attributes: - name - A unique name within the <task-definitions> that identifies this AbstractTaskRuntime sub-class. - order - A numeric value that indicates the order that the task should be executed in. This is only required when running in legacy mode - active - A true/false value that indicates whether the task gets executed or not. This is an excellent attribute for testing large job streams.
class	Contains the fully qualified name of an AbstractTaskRuntime sub-class to put in the job stream.
resource	A free-form element that can point to other XML elements in the JBJF Batch Definition file that this task needs, such as <database-definition>, <sql-definition>, <ftp-definition> or <export-definition>. This is part of the named resource concept mentioned earlier. The attributes for this element are: - id - A unique text identifier for the definition element. - type - This can be an XML element name to a resource that the task will need to run. For instance, a type=connection and a value db-one would indicate the task needs a database connection and the name of that connection is db-one. Internal resources recognized by JBJF include: - database-definition - A <database-definition> element. - sql-definition - An <sql-definition> element. - ftp-definition - An <sql-definition> element. - export-definition - An <export-definition> element. If type doesn't match one of the above XML elements, than it is assumed to be a custom value and gets stored on the job stack using type as the key and element value for the value.
database-definitions	A top-level element that stores all the database connection definitions for the batch process.
database-definition	Stores the parameters necessary to define and establish a JDBC database connection. The attributes for this element are; - name - A unique name to identify the given <connection>. When parsed, this element will be stored into a JBJFDatabaseConnection class object.
type	A flexible element that is typically set to a specific SQL based database engine. Recognized types include: - oracle - - mysql - - odbc - - access - - sql-server – This element isn't really used other than for annotations.
driver	A fully qualified name of the JDBC driver class.
server	A fully qualified name for the server.
database	Name of the database.
port	Port number. Some database engines use a port such as Oracle and MySQL.
usr	Encrypted userid for database access.
pwd	Encrypted password for database access.
client	Some JDBC drivers are tailored for different clients.
log-definitions	A top-level element for different log4j definitions.
log-definition	Contains the log4j configuration file and category. This element has not been

	fully explored yet. JBJF currently utilizes this for a default logger in Log4j.
log4j	Points to a single log4j properties file. The element contains the following attributes: - category - A special text value that maps to the desired logger in the log4j properties file.
ftp-definitions	A top-level element that groups all the <ftp-definitions> for the batch process/process.
ftp-definition	An XML element that defines a single ftp transfer operation. When defining an ftp-definition, be attentive to whether the FTP is a "pull" from a remote server to the batch process's host server or a "push" from the batch process's host server to a remote server. The difference between a "pull" and a "push" affects how you code the <target> and <source> elements.
source	The <source> XML element for an <ftp-definition> represents the directory path (relative or absolute) of the file that is getting transferred. For a "pull" type FTP, the <source> will be the directory path on the "remote" server. For a "push" type FTP, the <source> will be the directory path on the localhost where the batch process is running.
target	The <target> XML element for an <ftp-definition> represents the directory path (relative or absolute) where the file gets transferred to. For a "pull" type FTP, the <target> will be the directory path on the "localhost" where the batch process is running. For a "push" type FTP, the <target> will be the directory path on the "remote" server.
filename	The <filename> for an <ftp-definition> represents the filename of both the <source> and <target>. The current version of the JBJF only supports FTP using the same filename. If you need to transfer a <source> to a different filename, then you'll need to provide a custom <resource> on the specific task. Then write a special FTP task sub-class that uses your custom <resource> filename in place of the JBJF <filename>.
server	The <server> for an <ftp-definition> represents the "remote" server.
usr	The <usr> for an <ftp-definition> represents the encrypted text for the userid that will be used to login to the remote server for the FTP.
pwd	The <pwd> for an <ftp-definition> represents the encrypted text for the password that will be used to login to the remote server for the FTP.
sql-definitions	A top-level element that contains all the various SQL statements that get used by the batch process tasks.
sql-definition	An XML element that encapsulates a single SQL statement and optional parameters. A single <sql-definition> XML element will get mapped to a JBJFSQLDefinition object and added to an sql-definitions collection within the JBJFBatchDefinition object. Use the getSQLDefinitions() getter method to return this HashMap collection. The attributes for this XML element are: - name - Part of the named resources sub-system, this contains a unique textual key that must be unique within the <jbjf-sql> element. - type - Not currently used at the moment. - order - Not currently used at the moment.
text	Contains the SQL statement for the <sql-definition>. Make sure you include the CDATA tags to avoid XML parsing problems if your WHERE clause contains a ">" or "<" character.
sql-parameters	A parental XML element that contains one or more <param> elements for the individual placeholders in the SQL statement in the <text> element. See the discussion on SQL Definitions for more details.
param	A single parameter value that gets substitutes in the SQL statement in the <text> element. Placeholders are "?" characters and they get substitutes in order into the SQL statement text. Attributes for this element are: - name - Unique textual key to locate the parameter in the sql-definition element. - order - Defines what placeholder to substitute the parameter in. - type - Currently only "string" and "int" are supported. This determines whether the rendered placeholder contains quotes around it or not. A type="string" will render a parameter value surrounded by quotes, type="string">my_value</param> results in "my_value". If type="int">my_value</param> results in my_value.
export-definitions	A top-level element that contains one or more <export-definition> elements.
export-definition	An XML element that configures a single export/extract action that gets used by one or more tasks in the batch process.

target-file	Contains a full (absolute) or partial directory path and filename of where the export/extract file will be saved.
format-file	A textual value that determines the text format of the file. The following values are predefined by JBJF: csv - Extract values will be comma separate. tab - Extract values will be TAB delimited. Any other value implies a text file and the <delimiter> value is used as a column separator.
delimiter	A single character value that will be placed between individual export/extract values.
resource	A <resource> element for the <export-definition> provides a "key" to the job-stack of what the name of the ResultSet object is. The JBJF expects a <resource> for an <export-definition>, and it should be the name/key of the java.sql.ResultSet object that resides on the main job-stack. Thus, an <export-definition> expects that an SQL statement has already been run and processed and that the ResultSet of that SQL has been stored on the job-stack using the name/key supplied by the <resource> element.
account-definitions	A top level (tier 1) XML element that contains any account-definition elements for the batch process.
account-definition	An XML element that defines an account for charge back purposes.
type	An attribute that indicates what type of account.
name	The name of the account, does not need to be unique.
default	A true/false indicator on whether the account is the default account.
id	A unique text identifier for the account element.
plugin-cipher	The unique identifier of the plugin cipher to use for account, user, password, etc… decryption.
password	The password (if any) for the account.
account-number	

Predefined Tasks

The Java Batch process Framework comes with a set of pre-defined tasks that you may use. Check the package org.jbjf.tasks.* packages and the Javadocs for complete details.

Some of the more popular ones:

Task	Description/Comments
CopyFile	Simple class that can copy a file from one location to another.
FTPPushFile	A simple FTP task. Simply list this task in your <jbjf-tasks> element and list an ftp-definition resource. Then define your file transfer properties and you're done. Be attentive to the concept of source and target here. In a "push" transfer, the "source" refers to the directory/folder where the file is currently residing on the host machine. The target refers to the directory/folder on the remote server where the file is getting transferred. Also, "push" means you're transferring from the server where the batch process is running "to" a remote server.
FTPPullFile	A simple FTP task. Simply list this task in your <jbjf-tasks> element and list an ftp-definition resource. Then define your file transfer properties and you're done. Be attentive to the concept of source and target here. In a "pull" transfer, the "source" refers to the remote directory/folder where the file is currently residing. The target refers to the local directory/folder on the host server where the batch process is running. Put another way, "pull" means you're transferring from the remote server to the local server.
AbstractSQLTasl	Designed to be a base class for any SQL based task.
SQLSelectTask	Will run a Select SQL statement and place the Resultset onto the ProcessContext using a context-key.
SQLActionTask	Will run an action SQL statement, INSERT, UPDATE, DELETE.
ReadTextDocument	Will read in a flat, ascii text file into a DelimitedDocument and place it on the ProcessContext using a context-key.

Tutorials

The JBJF project comes with a number of tutorials. The tutorials are organized in a free form manner, but the first tutorial covers the essential concepts in order to get you acquainted with JBJF. Other tutorials focus on different services that JBJF provides, such as database access, SQL, exporting and FTP.

The following table outlines the tutorials and the concepts covered within that tutorial.

Tutorial	Version	Concepts/Comments
Basics	1.0.0	Basic task-list, logging, email.
Databases	1.0.0	SQL database, job stack.