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.

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

JDom - The JBJF utilizes the JDom library for XML processing and we would like to acknowledge them and their work.

Bouncy Castle - The JBJF utilizes the Bouncy Castle encryption library and 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. Jobs running behind the scenes, not even connected to the Web UI, 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 an environment using mixed technologies, both open source and commercial. The mixture in turn causes maintenance issues, learning curves, vendor involvement and compounds complexity 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 jobs with. While to concrete to be a Design Pattern, it does implement what would be termed a batch process design. Because it's a component framework, the JBJF enforces a standard approach to creating Java batch jobs, 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 a simple task-list concept that provides a fixed pattern for each task within the batch job. The developer analyzes the batch process they need to implement and breaks that batch process down into individual tasks. Each task can then be programmed into a single JBJF Task. The collection of tasks as a whole constitutes the batch process. Using the JBJF task-list concept, you sub-class key Java classes from JBJF to create your batch job and tasks. Once all your tasks are written, you then define the batch job 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. Once all the pieces are coded, you compile, package and run your job using a standard Java command line syntax.

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, Linux, Windows or almost any OS that runs a JVM.

JBJF comes packaged with basic services such as:

- Logging
- Email
- Zip/GZip Archiving
- FTP
- Encryption
- JDBC Database connections

Many of these services are optional, controlled by the inclusion or removal of key XML elements in the JBJF batch definition file. By combining XML configuration with these services, JBJF is a simple yet powerful component that can expedite the development of batch jobs.

JBJF was developed with the following goals:

- Platform independence. Jobs can be migrated between Unix, Linux 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 job 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 jobs. By abstracting the majority of parameters from the batch job 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 job 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.

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 job.

Overview

As already mentioned in the beginning, the JBJF operates on two primary aspects, a task list concept and an XML configuration (definition) file. While not essential to developing JBJF batch processes, understanding these two aspects will expedite many unknowns when you start coding. In this section we will discuss the various architectures important to JBJF.

JBJF is defined and controlled via an XML file called a JBJF Batch Definition file. Each batch job 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 job needs. The JBJF definition file will also include a list of Task (ITask) classes that make up the batch job. 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-documenting.

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:

- JDom XML DOM Implementation
- JDom support libraries (saxpath and jaxen)
- Xerces XML Parser (now included in many JRE/JDK)
- Bouncy Castle Encryption
- Apache common networks
- Apache Log4J
- JDBC drivers
- Sun Microsystems' Email Library

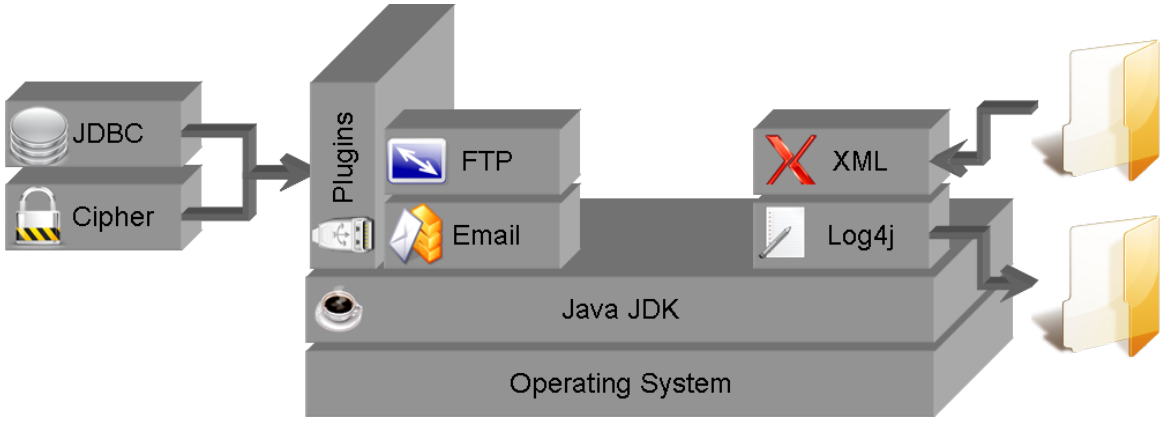
For brevity, JDom, saxpath and jaxen will be grouped in with the Xerces XML Parser since all of these libraries are used solely for XML file processing. JBJF doesn't use these components directly.

Component	Class/Category	Description/Comments
XML - Xerces XML Parser	Java Library	Apache Software Foundation's implementation of a SAX (Event) based XML parser. Used exclusively by JDom.
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.

[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.

		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.
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 new architecture introduced in JBJF 1.3.0. The Plugins architecture provides the extensibility of JBJF for JDBC Databases and Cipher (Encryption/Decryption) services.

Philosophy - Rethinking Batch Job Design

JBJF provides a component framework to develop batch jobs, allowing you (the developer or architect) to build a batch landscape. But JBJF also requires a different approach, philosophy you might say, to batch process development. The JBJF and philosophy work together to create a unified batch process development strategy.

A traditional batch job design takes a sequential top-down approach. We typically grab the data at the top (start) of the batch job, 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 Korn Shell) to intelligent scripting (Perl, PHP) to

sophisticated complied code like C/C++ or some mixture. Typically, the individual batch job 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 just has to run each step to carry out the steps, thus executing the batch process. This approach works ok, but isn't very reusable. You can copy code to the next batch job, but you need to change the methods to meet the needs of the new requirements. Also, if the batch job purpose has 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 order. But it sees the batch job as a series of tasks, where each task provides one part of the overall job. We can now take each step and package it into a Java sub-class. Each sub-class comes equipped with a common method, runTask(), where we put the Java code that runs the functionality of the task. This is the first phase to rethinking batch job design and is the basis for reuse.

If we take our initial batch process of steps:

- 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 classes runTask() method, thus carrying out the batch job duties. 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 sending a different JBJF Batch Definition file (XML) in with the new parameters.

The key difference here is the traditional design doesn't really focus on reuse. Steps are developed as a single use item, specific to the process. yes you can use ${ENV} variables to make steps more flexible, but this becomes difficult to manage and reuse as the number of batch processes increases. The JBJF design directs you to have your task collect it's parameters and variables from the JBJF Batch Definition file. Thus, tasks are written in a fashion that encourages the use of parameters. Reuse is accomplished by simply changing the parameters. As developers become better at JBJF design, they also find better ways to write Tasks in a more reusable fashion.

How JBJF Helps

So, how does JBJF help me address the batch job environment? First off, we've simplified the development process. 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 job, AbstractBatch and AbstractTask. Thus, JBJF helps you immediately because it is "simple". With only two classes at the core, it's easily digested and small in scope.

Use of a standard development approach means classes become similar in nature and similar in how they are implemented. This similarity means easier maintenance and training. Similarity in implementation promotes reuse, faster development and a measurable ROI. Reuse means less testing as functional (proven) tasks can now be reused in other batch processes with little or no re-testing. As you can see there is a chain reaction of events that occur that allow JBJF to help.

JBJF also encapsulates much of the mundane batch process functionality and services such as initialization, closure, logging and instantiation. We then use an XML Batch Definition file to provide JBJF with key data and parameters that define your batch job. Thus, JBJF helps out by removing much of the mundane iteration, initialization and maintenance that normal batch development incurs.

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.

Task List Concept

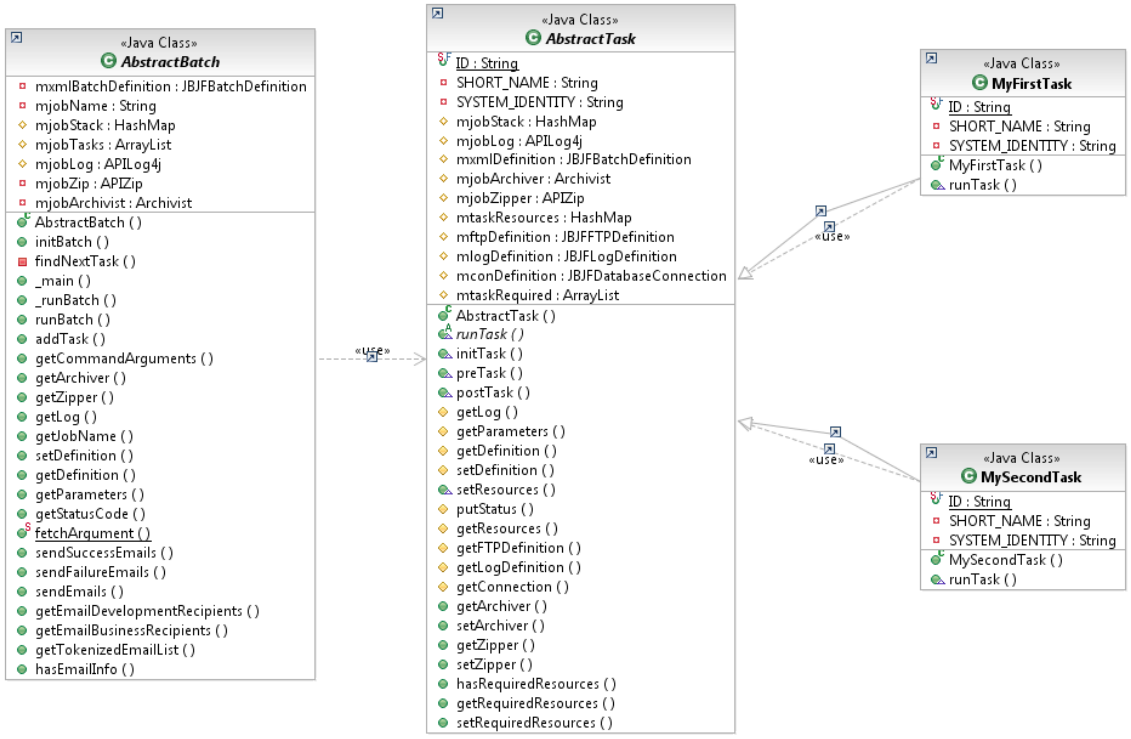
As mentioned earlier, the central concept of the Java Batch Job Framework is the task list concept. Essentially a task is a small, single step, representing part of a larger process or batch flow. Tasks such as this include, copying a file, ftp a file, run an SQL statement, export a record set, etc... The tasks are managed by a parent batch process that handles one or more tasks. The combination 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:

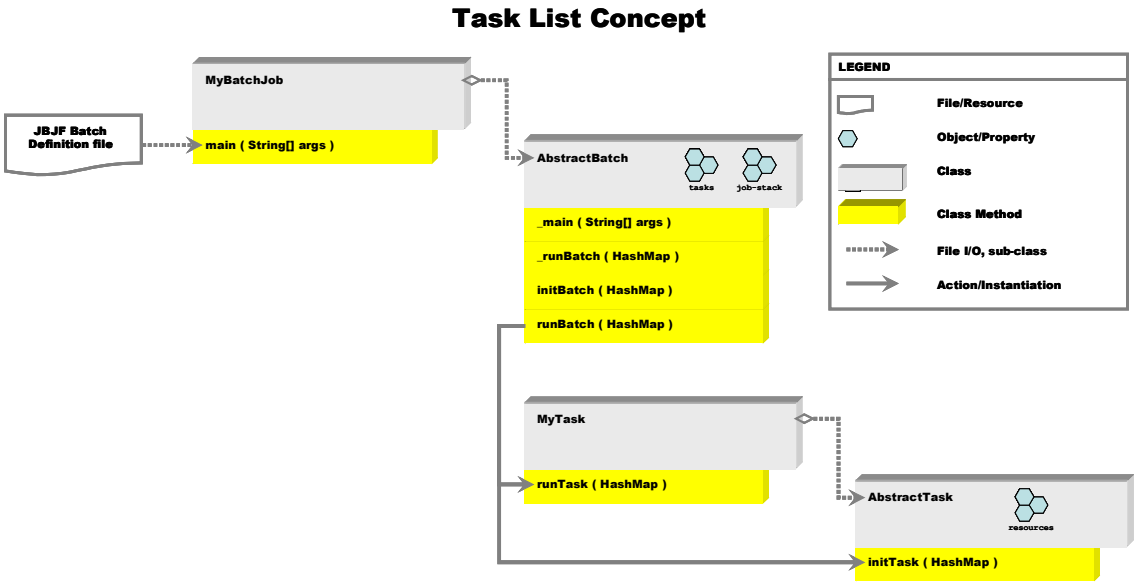
- initTask()
- runTask()

Each task within your batch job will extend the AbstractTask or implement the ITask interface, essentially forcing the implementation of the runTask() method. The JBJF then iterates through all the assigned Task sub-classes listed in the JBJF Batch Definition file, running the runTask() method, thus processing the batch job. The traditional choice is to extend AbstractTask, thus inheriting the default initTask() implementation and class property management. More about this in the Basic Tutorials.

These classes are illustrated in the following class diagrams along with two Junit test classes:



The task list concept is illustrated in the following diagram:



A brief overview of the Task List concept:

- ✔ The entry point from the command line is the traditional Java main() method. The main() method is implemented in your batch job class (AbstractBatch 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 (HashMap) as a key to a value/object.

- The command line arguments are forwarded from the AbstractBatch sub-class, MyBatchJob, as is. The AbstractBatch._main() expects a JBJF Batch Definition file. This XML file is parsed and stored as a large class object (JBJFBatchDefinition) that stores all the parameters as individual Java class objects and collections.
- Control then moves from the _main() method to _runBatch() and initBatch() 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 AbstractTask sub-classes that make up the task-list. Each of these sub-classes will be instantiated using Class.forName() and then passed to addTask() to be placed onto a special task collection (ArrayList).
- Once XML parsing is complete, services are established and the task-list is created, control is passed to the AbstractBatch.runBatch() method. In here, the runBatch() method will iterate through each AbstractTask sub-class, running first the initTask() that resides in AbstractTask (unless you over-ride this). Second, the runTask() method which exists within your custom AbstractTask sub-class, MyTask.

Named Resources

Another essential concept for the JBJF is named resources. A task within a batch job requires resources such as database connections, ftp connections 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?

When we discuss the JBJF Batch Definition file you will notice that many of the XML elements in the JBJF Batch Definition file come with a name attribute. When the JBJF Batch Definition file gets parsed and stored as Java class objects, the name attribute serves as a key to the Java class object. This concept is illustrated in the following diagram:

```
<jbjf-tasks>
       <task name="t001" order="1" active="true">
           <class>org.adym.batch.tasks.MyFirstTask </class>
           <resource type="sql-definition">first-ftp</resource>
           <resource type="connection">source</resource>
       </task>
       <task name="two" order="2" active="false">
           <class>org.adym.batch.tasks.MySecondTask </class>
```

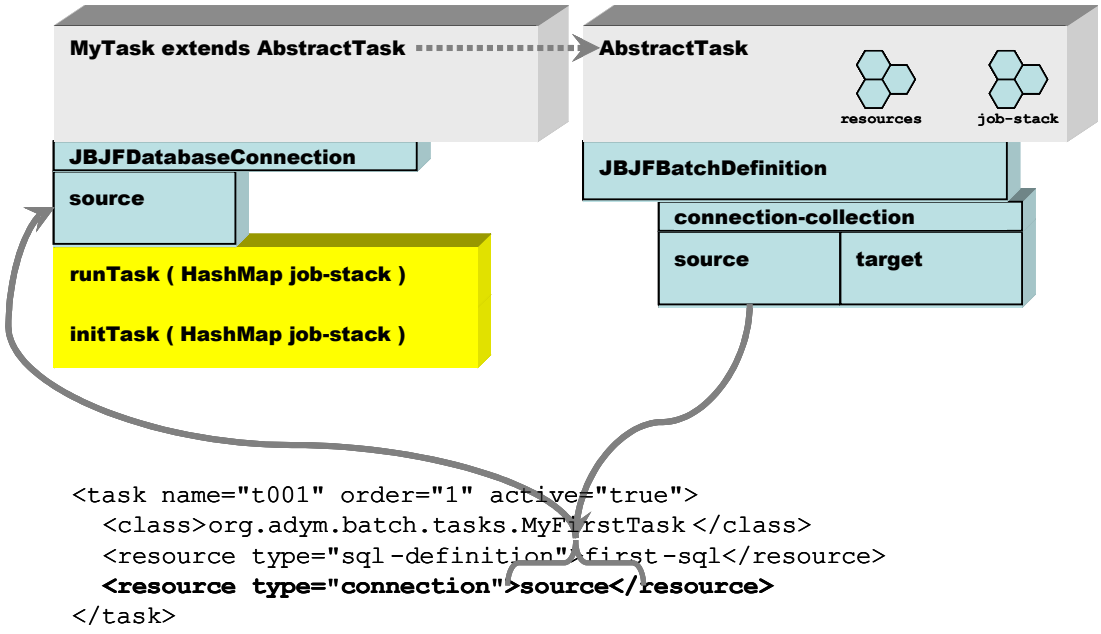
Resource Type

Named Resource

```
   <!--
   DATABASE CONNECTIONS: Optional
   Use these <connection> elements to define database connectio ns
   that the JBJF batch job needs to run SQL.
   -->
   <jbjf-connections>
       <connection name="source">
           <type>oracle</type>
           <driver>oracle.jdbc.driver.OracleDriver</driver>
           <server>xxxxx.adym.com</server>
           <database>XE</database>
           <port>1521</port>
           <usr><![CDATA[b93d56fd7ab6b966]]></usr>
           <pwd><![CDATA[5f9598653f15a033edf0ab366c57c9b9]]></p wd>
           <client>jdbc:oracle:thin</client>
       </connection>
       <connection name="target">
           <type>filesystem</type>
           <driver>file.jdbc.driver.FileSystem </driver>
```

Notice in task t001 a resource of type=”connection” with a name of source is being indicated. Further down in the Batch Definition file are the jbjf-connection elements. Listed within the jbjf-connection is a connection element, name=”source”. The XML elements eventually get mapped to Java class objects and your task sub-class will pull these named resources in during the initTask() execution:

Named Resources




```
<task name="t001" order="1" active="true">
  <class>org.adym.batch.tasks.MyFirstTask </class>
  <resource type="sql-definition">first-sql</resource>
  <resource type="connection">source</resource>
</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 traditionally done in the Default Constructor of your 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 "enable" 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> element. Of course the False value is a moot value in lieu of the exception being thrown. Again, this is by design. 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 JBJF and your task.

Job Stack

The JBJF also relies on a central HashMap collection that gets passed between various Task sub-classes during the batch job. We refer to this as the job-stack, and the HashMap serves as a poor man's indexed table. The job-stack 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" objects and results onto the job-stack near the end of it's runTask() method. Then task2 would "get" those objects and results off the job-stack at the beginning of its runTask(). This process can be repeated from one task to another.

Because the job-stack is a HashMap type collection you need to keep a few things in mind when using it:

- Objects and results that you put onto the job-stack need a unique key. Otherwise, a "replace" is done. Generally, a key can be hard-coded within the runTask() method, but this can result in a limited reuse scope. 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 job-stack. Using <resource> XML elements as keys means a Task can be reused easier.
- 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.

Java Batch Job Definition file

At the start of every JBJF batch job is the JBJF Batch Job 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.

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 job has a number of common elements that it will rely on to operate. Information such as the batch job name, directories, userids, passwords and logging all play an essential role in providing a batch job the resources it needs. Some resources are required by all batch jobs, other resources are needed by one batch job, but not another. So, one important question is how to supply each individual batch job the necessary resources in an easy, flexible and repeatable way. The answer is to "abstract" batch job 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 job through the command line or some other argument container, thus allowing the batch job to process the abstract form. Finally, this process can be repeated for another batch job 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 job 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 job's JBJF Batch Definition file and change the XML data for the new batch job. The new batch job 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 job 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 job running in the midrange tier. These services also represent the broadest range that most batch jobs require:

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

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

- 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 job's directory tree. Subsequent services are grouped under similar elements:

- Email <jbjf-email>
- FTP <jbjf-ftp>
- Export <jbjf-export>
- Tasks <jbjf-tasks>
- General Purpose <jbjf-parameters>
- Database <jbjf-connections>
- SQL <jbjf-sql>
- Logs <jbjf-logs>
- Plugins <jbjf-plugins>

These top-level elements follow a certain naming convention, using jbjf- as the prefix. 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 structure of our JBJF Batch Definition file is as follows:

- General Purpose Parameters
- Tasks
- Directories
- Connections (Optional)
- FTP Definitions (Optional)
- SQL Definitions (Optional)
- Export Definitions (Optional)

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
FILE        : jbjf-base-definition.xml
DATE        : February 12, 2007
DEVELOPER   : Adym S. Lincoln
PURPOSE     :
Used to test the various classes in the org.adym.jbjf.xml package.
These are XML parsing and storage classes that provide a Java class
representation of the JBJF Batch Definition file.  See the JUnit
classes in the ./testing source folder.

This XML file also contains a sample JBJF Batch Definition file that
can be copied and modified for your purposes.

Don't edit or change anything in this file unless a change has been
done to the underlying XML class.
-->
<jbjf-batch-job>

    <jbjf-parameters>
        <name>jbjf-base-definition</name>
        <enable-archivist>Y</enable-archivist>
        <enable-email>Y</enable-email>
    </jbjf-parameters>

    <jbjf-email>
        <notifications>
            <email attachments="Y">lincolna@hotmail.com</email>
            <email attachments="N">lincolnb@hotmail.com</email>
            <email>lincolnc@hotmail.com</email>
```

```
    </notifications>
    <email-host>smtp.host.org</email-host>
    <email-sender>jbjf-batch-job@hotmail.com</email-sender>
</jbjf-email>

<!--
DIRECTORIES: Required
In this element you need to define the directories and directory
parts that get used in the JBJF Batch job tasks.  Each <directory>
element comes with a name attribute and an addressing attribute.  Some
<directory> names are reserved for JBJF.  The addressing attribute
is either relative or absolute.  A relative addressing means the
directory will be appended to the base <directory> element.  An
addressing of absolute means the <directory> element is taken
as-is, no appending of the base element:
- base - Indicates a base directory from which all "relative"
addressed <directory> elements are built.  Because the base
<directory> is a starting point, an addressing attribute here
doesn't make sense.
- archivist - Used by the Archivist to build and store zipfile
archive.
-->
<jbjf-directories>
    <directory name="base" addressing="relative">.</directory>
    <directory name="archivist" addressing="relative">archives</directory>
    <directory name="log4j" addressing="relative">etc</directory>
</jbjf-directories>

<!--
TASKS: Required
In this element you need to define the tasks that will be used
in the batch job.  Each task contains the following attributes
and elements:
- name - Unique within the tasks.  The name defines the key
that will get used.
- order - Numeric indicator that defines where in the task-list
that the task gets run.
- active - A TRUE/FALSE indicator that defines whether the task
will get run or not.  Used primarily for testing and debugging.
- class - A fully qualified class name that gets used to instantiate
the task class.
- resource - These XML elements serve as pointers to other XML
elements within the JBJF Definition file for one or more resources
that the task needs to complete it's work.  A resource can also
be a simple String property/value that the task will need.
-->
<jbjf-tasks>
    <task name="t001" order="1" active="true">
        <class>org.adym.batch.tasks.MyFirstTask</class>
        <resource type="ftp-definition">first-ftp</resource>
        <resource type="connection">source</resource>
    </task>
    <task name="two" order="2" active="false">
        <class>org.adym.batch.tasks.MySecondTask</class>
    </task>
</jbjf-tasks>

<!--
DATABASE CONNECTIONS: Optional
Use these <connection> elements to define database connections
that the JBJF batch job needs to run SQL.
-->
<jbjf-connections>
```

```
    <connection name="source">
        <type>oracle</type>
        <driver>oracle.jdbc.driver.OracleDriver</driver>
        <server>xxxxx.adym.com</server>
        <database>XE</database>
        <port>1521</port>
        <usr><![CDATA[b93d56fd7ab6b966]]></usr>
        <pwd><![CDATA[5f9598653f15a033edf0ab366c57c9b9]]></pwd>
        <client>jdbc:oracle:thin</client>
    </connection>
    <connection name="target">
        <type>filesystem</type>
        <driver>file.jdbc.driver.FileSystem</driver>
        <server>jbjf-002.adym.com</server>
        <database>XE1</database>
        <port>1522</port>
        <usr><![CDATA[b93d56fd7ab6b966]]></usr>
        <pwd><![CDATA[5f9598653f15a033edf0ab366c57c9b9]]></pwd>
        <client>jbjf:oracle:thin</client>
    </connection>
</jbjf-connections>

<!--
LOG DEFINITIONS:
Contains one or more log4j property files.  There should
always be a log-definition with a name="default" to ensure that
at least one logfile is created.  Each log-definition has a single
child element called log4j that contains a category= attribute
and a full/partial path and filename to the log4j properties
file.  The category attribute should map to a category line
item in the log4j properties file.
-->
<jbjf-logs>
    <log-definition name="default">
        <log4j category="org.adym.batch">./etc/log4j.properties</log4j>
    </log-definition>
</jbjf-logs>

<!--
FTP DEFINITIONS: Optional
Should your JBJF batch job need to do FTP file transfers, you'll
need to put in <ftp-definition> elements for the file transfers
that you need to perform.
-->
<jbjf-ftp>
    <ftp-definition name="first-ftp">
        <source>dir/my-1-filename.txt</source>
        <target>dir/sub-dir</target>
        <filename>my_ftp_file.txt</filename>
        <server>ftp-file-server.adym.org</server>
        <usr><![CDATA[b93d56fd7ab6b966]]></usr>
        <pwd><![CDATA[5f9598653f15a033edf0ab366c57c9b9]]></pwd>
    </ftp-definition>
</jbjf-ftp>

<!--
SQL DEFINITIONS: Optional
These <sql-definitions> define one or more SQL statements that
will be used by the JBJF batch job.  Like all "named" resources,
the name attribute should be unique within the JBJF Batch Definition
file.  The SQL can be shared within many tasks.  The <sql-definition>
should contain the following attributes and elements:
- name - Unique within the JBJF Batch Definition file.  This will
```

```
    serve as the key for locating this SQL statement.
    - text - Contains the SQL statement for this <sql-definition>.  Make
    sure you surround this with a CDATA tag to ensure that a < or >
    within a potential WHERE clause doesn't break the XML parser.
    - sql-parameters - Contains one or more <param> elements that
    contain parameters for the SQL statement.  These get substituted
    for "?" placeholders in the SQL text.
    -->
    <jbjf-sql>
        <sql-definition name="sql-test-001" type="select" order="1">
            <text>
                <![CDATA[
                select * from my_table;
                ]]>
            </text>

        </sql-definition>

        <sql-definition name="sql-test-002" order="2">
            <text>
                <![CDATA[
                begin
                    ? := schema.pkg_adym.do_function (
                        ?
                    );
                end;
                ]]>
            </text>

            <sql-parameters>
                <param name="p002" order="2" type="string">pvalue</param>
            </sql-parameters>

        </sql-definition>

    </jbjf-sql>

    <!--
    EXPORT DEFINITIONS: Optional
    These <export-definition> elements define one or more export
    definitions that the JBJF batch job needs.  These get used in
    one or more tasks.  An <export-definition> is tailored for the
    export and/or extracting of data to a specific location.
    -->
    <jbjf-export>
      <export-definition name="x001">
            <target-file>./data/my-export-file.txt</target-file>
            <format-file>text|csv|other</format-file>
            <delimiter>|</delimiter>
            <resource type="resultset">sql-test-001</resource>
      </export-definition>

    </jbjf-export>

</jbjf-batch-job>
```

Element Details

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

XML Element	Description/Comments

jbjf-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 job. Used in the emails. **enable-archivist** - A single Y/N or 0/1 character that indicates whether to utilize the archivist service or not. Use of the archivist means you'll need to create an AbstractBatch sub-class and over-ride the runBatch() method. You can still call super.runBatch() to iterate through all the AbstractTask sub-classes, but you'll need to put in code that manages the archivist and places files in there for archiving. **enable-email** - A Y/N or 0/1 character that indicates whether to utilize the email service.
name	Contains the name of the batch job. This is currently used for email subject lines.
enable-archivist	A Y/N or 0/1 character that indicates whether the batch job will utilize the archivist (zipfile) or not. When using the archivist, you need to create an AbstractBatch sub-class, the DefaultBatch will not work.
enable-email	A Y/N or 0/1 character that indicates whether the batch job will utilize the email service.
jbjf-email	An optional element, but required when the enable-email is set to Y/1. This contains various sub-elements that are used to configure the email server, connection and recipients.
notifications	This parental element contains a list of email recipients that need to receive an email from the batch job.
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 job 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.
jbjf-directories	A parental element that contains a list of <directory> sub-elements, where each <directory> represents a relative or absolute pathway for the batch job's directory tree.
directory	Contains a single relative or absolute pathway for the batch job'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 job 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 job. Thus, if archivist = /usr/apps/batch/archives with addressing="absolute". When you run your batch job, a new directory will be created, /usr/apps/batch/archives/yyyy-mm-dd-hh-mi/ that contains the zipfile archive.
jbjf-tasks	Contains the task-list of AbstractTask sub-classes.
task	A parental element that contains all the data to define a single task for the batch job. There will be a <task> element for each AbstractTask sub-class. The <task> element has the following attributes:

	- name - A unique name within the <jbjf-tasks> that identifies this AbstractTask sub-class. - order - A numeric value that indicates the order that the task should be executed in. The order should generally be sequential with no gaps, but JBJF will just use the numeric value as a placement value, thus 1, 2, 3, 4 could be 10, 20, 30, and 40. - 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 AbstractTask 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 need, such as <connection>, <sql-definition>, <ftp-definition> or <export-definition>. This is part of the named resource concept which we will cover later. The attributes for this element are: - 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: - connection - A <connection> element in the <jbjf-connections>. - sql-definition - An <sql-definition> element in the <jbjf-sql>. - ftp-definition - An <sql-definition> element in the <jbjf-ftp>. - export-definition - An <export-definition> element in the <jbjf-export>. 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.
jbjf-connections	A top-level element that stores all the database connection definitions for the batch job.
connection	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 -
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.
jbjf-logs	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.
jbjf-ftp	A top-level element that groups all the <ftp-definitions> for the batch job/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 job's host server or a "push" from the batch job'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 job 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 job 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.
jbjf-sql	A top-level element that contains all the various SQL statements that get used by the batch job 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.
jbjf-export	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 job.
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.

Parsing and Storage

When the JBJF Batch Definition file gets parsed, it will get stored into various objects corresponding to the various XML elements. These objects in turn get stored in traditional Java HashMap collections which you access with a standard getter method. The following table outlines the various XML elements, Java Class object and getter methods for the JBJF Batch Definition. For complete details see the Javadocs for JBJF.

XML Element	Java Class Object	Getter/Access	Description/Comments
jbjf-batch	JBJFBatchDefinition	getDefinition()	Available in both the AbstractBatch and AbstractTask, this returns the entire JBJFBatchDefinition object. Thus, the getDefinition() is available in all batch job and task sub-classes.
jbjf-parameters	JBJFBatchDefinition	getName() isArchivistEnabled() isEmailEnabled()	These properties are available directly from the JBJFBatchDefinition object.
jbjf-directories	JBJFBatchDefinition	getDirectories()	Returns a HashMap collection of JBJFDirectoryItem objects.
jbjf-tasks	JBJFBatchDefinition	getTasks()	Returns a HashMap collection of JBJFTaskItem objects. This is not the same as the ArrayList collection that contains the instantiated class objects that get run.
jbjf-connections	JBJFBatchDefinition	getConnections()	Returns a HashMap collection of JBJFDatabaseConnection objects.
jbjf-ftp	JBJFBatchDefinition	getFTPDefinitions()	Returns a HashMap collection of JBJFFTPDefinition objects.
jbjf-sql	JBJFBatchDefinition	getSQLDefinitions()	Returns a HashMap collection of JBJFSQLDefinition objects.
jbjf-logs	JBJFBatchDefinition	getLogDefinitions()	Returns a HashMap collection of JBJFLogDefinition objects.
jbjf-export	JBJFBatchDefinition	getExportDefinitions()	Returns a HashMap collection of JBJFExportDefinition objects.
notifications		getEmailAttachments() getEmailNotifies()	These return HashMap collections of the email addresses based on whether the recipient wishes to receive attachments of not.
name	JBJFBatchDefinition	getName()	Returns the batch job name...element <name>.
enable-archivist	JBJFBatchDefinition	isArchivistEnabled()	Returns a Boolean indicator based on the <enable-archivist> element value.
enable-email	JBJFBatchDefinition	isEmailEnabled()	Returns a Boolean indicator based on the <enable-email> element value.
email	HashMap	getEmailAttachments() getEmailNotifies()	Use the get() method of the HashMap collection to fetch individual recipient email addresses.
email-host	JBJFBatchDefinition	getEmailHost()	Returns the email host name.
email-sender	JBJFBatchDefinition	getEmailSender()	Returns the email sender value.
directory	JBJFDirectoryItem	getName() getDirectory() isRelative() getAddressing()	A class object that contains properties, getter and setter methods to process a single <directory> element.
task	JBJFTaskItem	getClassName() getTaskName() getActive() isActive() getOrder() fetchOrder()	A class object that contains properties, getter, setter and custom access method to process and manage a single <task> element.
connection	JBJFDatabaseConnect	getClient()	Traditional class object with

	ion()	getDatabase() getDriver() getName() getPass() getPort() getServer() getType() getUser()	properties, getter and setter methods. The User and Pass getter methods return "cleartext" values. The decryption is done when the class gets initialized.
log-definition	JBJFLogDefinition()	getCategory() getName() getProperties()	Traditional class object with properties, getter and setter methods. The getProperties() method is a bit mis-leading, it really returns the properties filename.
ftp-definition	JBJFFTPDefinition	getFilename() getName() getPass() getServer() getSourceDirectory() getTargetDirectory() getUser()	Traditional class object with properties, getter and setter methods. The User and Pass getter methods return "cleartext" values. The decryption is done when the class gets initialized.
sql-definition	JBJFSQLDefinition	getName() getSQLOrder() getSQLType()	Traditional class object with properties, getter and setter methods. The Order and Type properties are used at the moment.
sql-parameter	JBJFSQLParameter	getName() getOrder() getType() getValue()	Traditional class object with properties, getter and setter methods.
export-definition	JBJFExportDefinition	getFile() getFileFormat() getName() getResources() getTextDelimiter()	Traditional class object with properties, getter and setter methods.
jbjf-plugins	JBJFBatchDefinition	getPluginDefinitions()	Top level element that contains all the plugins needed by this batch process.
plugin-definition	JBJFPluginDefinition	getPluginDefinition(String)	A single plugin definition with the all the basic attributes that allow the JBJF to locate and load the plugin.
class	JBJFPluginDefinition	getPluginDefinition(String)	Indicates the actual name (not type) of the plugin class.

The parsing and storage of batch job parameters follows a pattern of XML Element to JBJF<Class> to JBJF Storage. This pattern is repeated for the each XML element in the JBJF Batch Definition file. Some XML elements are stored as simple class properties, while others get stored in HashMap collections. The difference is whether an XML element is a list or a single parameter. This is usually apparent, for instance email-host, email-sender, enable-email and enable-archivist are all basic class properties. Elements such as sql-definition, ftp-definition, email, connections and export-definition are stored as an individual JBJF<Class> but then added to a HashMap collection, as there could be one or more of these types of elements. With experience and time you'll become familiar with the storage aspects.

Predefined Tasks

The Java Batch Job Framework comes with a set of pre-defined tasks that you may use. Check the package org.adym.jbjf.tasks for the available tasks. The following table lists

these AbstractTask sub-classes and a brief synopsis on the usage. For complete details on these predefined Tasks please consult the Java docs.

Step/Method	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 job 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 job is running. Put another way, "pull" means you're transferring from the remote server to the local server.
SQLSelectUnit	This task will take an <sql-definition> from within the JBJF Batch Definition file and execute it against a given <connection>. When you define this in your JBJF Batch Definition, just make sure you list two <resource> elements that point to the correct <sql-definition> and <connection>. <uow name="one" order="1" active="true"> <class>org.adym.jbjf.tasks.SQLSelectUnit</class> <resource type="sql-definition">select-lmig10-series</resource> <resource type="connection">ocp16d</resource> </uow> <uow name="two" order="2" active="true"> The SQLSelectUnit then takes the recordset returned from the database and puts it onto the job parameters stack using the name of the <sql-definition> as the key...thus, from the above example, select-lmig10-series.
SQLUpdateUnit	Similar to the SQLSelectUnit, this will run an UPDATE statement against a database. Again, two <resource> elements should be defined, one for the <sql-definition> that contains the UPDATE and another <resource> for the <connection> to the desired database.
SQLDeleteUnit	Similar to the SQLSelectUnit, this will run a DELETE statement against a database. Again, two <resource> elements should be defined, one for the <sql-definition> that contains the DELETE statement and another <resource> for the <connection> to the desired database.
SQLBaseUnit	This is a base class defined and extended by the other SQL- classes. Should you wish to write a custom SQL unit of work, you can extend this class and it will manage your <sql-definition> and <connection> objects for you. Simply use the getStatement() and getConnection() methods to fetch them.

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.
Filesystems	1.0.0	Exporting, FTP.