Java Batch Job Framework

Plugins 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 and/or utilizes 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 are JBJF Plugins?

JBJF Plugins are a new architecture being introduced into JBJF. JBJF Plugins provide developers a way to enhance and extend JBJF without having to know intimate details and specific code within the JBJF framework. Prior to Plugins the original JBJF could be enhanced, but it required intimate knowledge of the code and it also required you to download the source code and directly replace specific Java classes. Yuk!

The JBJF Plugins architecture allows you to enhance and extend JBJF with only a compiled JBJF runtime binary (no source code is needed). You need to gain some simple knowledge of the specific plugin category you are developing along with the Java Interface for that Plugin. Because the plugins are done outside the source code and "dropped" in as a runtime binary, enhancements can be done at any time and anyplace. All that is required is the JRE/JDK 6 and JBJF 1.3.0(+). See the JBJF Plugins Tutorial for complete details...

The initial release of JBJF Plugins will support the follow Plugin Categories:

Cipher	Encryption and Decryption. You write a simple Plugin class that implements the IJBJFPluginCipher or extends the IAbstractJBJFPluginCipher. Ciphers are now allocated at both the Batch and the "Task" level. A nagging issue with JBJF has been the ability (really inability) to use custom encryption keys and other encryption algorithms with your batch process. A simple switch from the default DES/CBC encryption to something like AES involved large amounts of custom code, class replacement and rebuilding of JBJF. Now the Cipher Plugin API allows us to write a custom implementation against whatever encryption library we want. Our Plugin gets compiled and packaged into a common jarfile. The jarfile then gets "dropped" into a "plugins" directory alongside the JBJF install. Tasks can then just reference the Cipher Plugin API to have text encrypted/decrypted with whatever cipher plugin you wish.
Database	JDBC Database types. Another hefty flaw in JBJF was total lack of any DAO layer. Database tasks were typically written to a specific database platform, MS SQL Server only, Oracle only or MySQL only. So, while the tasks could be reused for other like platform SQL work, they were limited to that one database platform only, hence reducing the ability to re-use the core task. Unlike a traditional DAO which usually targets a specific Data Model, the JBJF Database Plugin is designed to accept an XML <connection> element from a JBJF Batch Definition file and return a generic java.sql.Connection. So the Database Plugins a less of a DAO and

	more of a JDBC Database Platform. What it does allow however is that Tasks can now be written to the IJBJFPluginDatabase interface and are insulated from the specific JDBC database type (MS SQL Server, Oracle, MySQL). Using the new Database Plugin API, you write a simple Plugin class that implements the IJBJFPluginDatabase or extends the IAbstractJBJFPluginDatabase. The plugin then gets compiled into a common jarfile and dropped into the "plugins" directory.

The JBJF Team has plans to expand the Plugins to the following:

Email	A simple API that provides an email service for your batch process and/or individual tasks. A batch process and/or task will have the ability to dispatch email along with various file attachments.
Utility	A simple API that allows you to attach lists, collections, reporting or other free form objects to a given task during processing. Working with JBJF I noticed a need to have values that get created in one task available for use in another task. While the Job Stack can provide this functionality, the job stack keys need to be hard coded into the <task> elements as <resource> elements.

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

The JBJF Plugins uses the Java 6 (1.6) standard services API to process, load and manage Plugins. The choice of Java versus OSGi or another services API was the built-in nature of the JDK. By using the JDK API no additional jarfiles need to be included in JBJF deployments. The compromise is that you need to use Java 6, so if your JBJF is using 5 or earlier, you'll need to upgrade the JRE/JDK.

Now for the bad news. The new Plugins architecture "will" break some of your existing batch and task classes...I've already encountered some pain in this area. The pain can range from minor to moderate, depending on how much exposure your tasks have to your particular DAO and Cipher layers. In my case, I had luckily created base (Abstract) classes that dealt with each different DAO type (Oracle, MS SQL Server, etc...), so the breakage was minor, create a new Abstract class that pre-processed the plugin and then changed all my concrete Task classes to extend the new Abstract. Your breakage will vary depending on how your DAO Tasks were designed. The worst case scenario is that the Task classes did direct DAO. Of course, you can always just leave your current job portfolio at the 1.2.2 or earlier version and just create new processes using the 1.3.0 or better.

Software Architecture

The Plugin Architecture starts with a base Interface called IJBJFPlugin. This is a simple bean like interface that provides all the key methods and basic getters/setters for the properties for a Plugin:

Property	Description
id	A required indicator unique within a JBJF Batch Definition file that allows JBJF to lookup a given Plugin/Service.
name	An optional indicator that provides a simple description of the Plugin.
type	A required attribute that can be either descriptive (cipher) or specific to the actual type of Plugin (org.jbjf.plugin.IJBJFPluginCipher).
active	A boolean (Y/N or true/false) indicator on whether the current plugin is active and usable or not.

These four basic properties are listed in the JBJF Batch Definition file as attributes to a new JBJF XML Element, plugin-definition. The following is a sample illustration of how the JBJF Batch Definition file elements will be implemented:

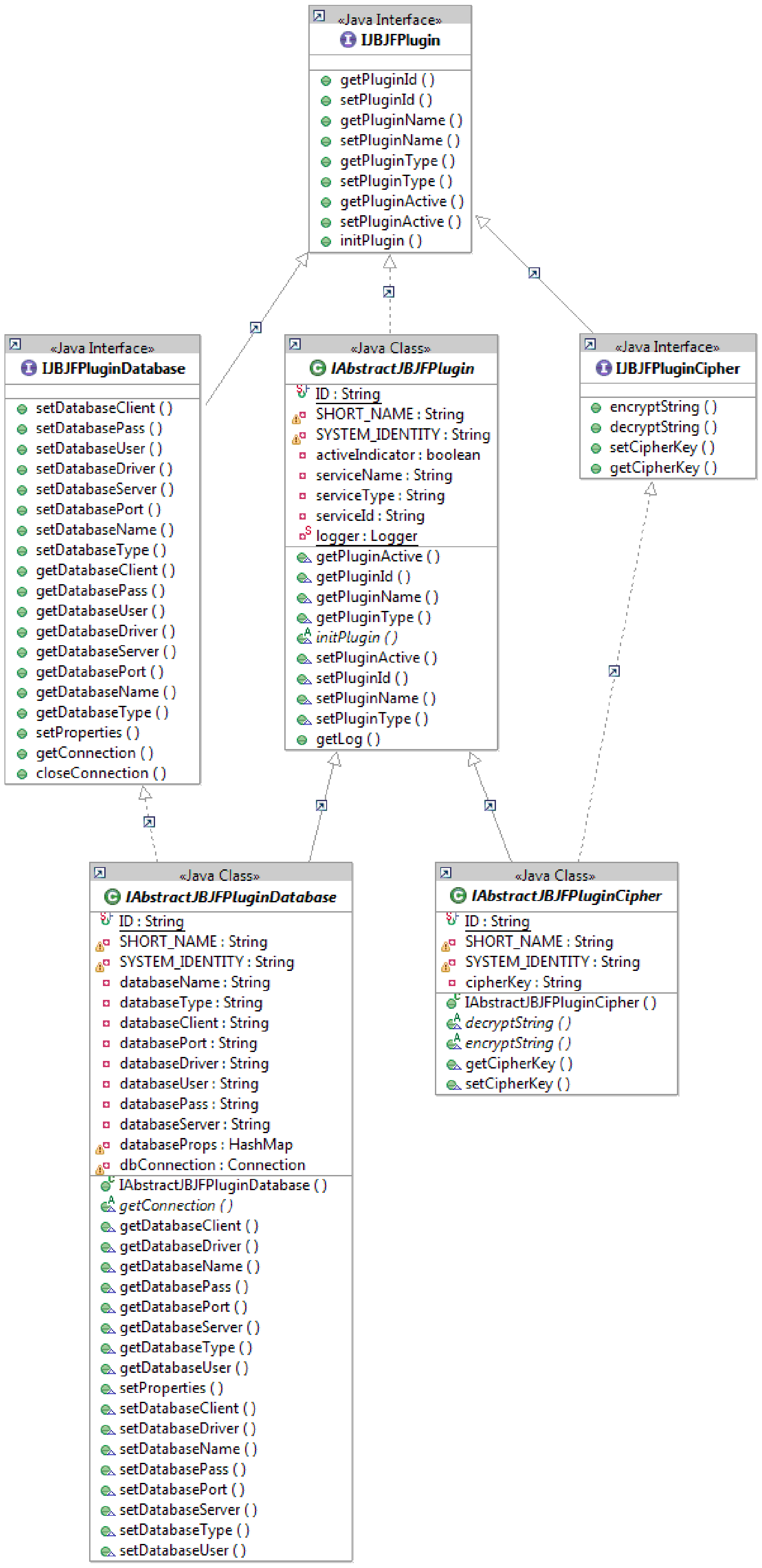
```
<jbjf-plugins>
    <plugin-definition
        id="plugin001"
        name="default-cipher"
        type="org.jbjf.plugin.IJBJFPluginCipher"
        active="true">
        <class>org.jbjf.services.impl.DefaultCipher</class>
    </plugin-definition>
```

```
        <plugin-definition
            id="plugin002"
            name="oracle-database"
            type="org.jbjf.plugin.IJBJFPluginDatabase"
            active="true">
            <class>org.jbjf.services.db.OracleJBJFService</class>
        </plugin-definition>
    </jbjf-plugins>
```

The IJBJFPlugin Interface has a supporting Abstract class that does a partial implementation, providing getter/setter methods for each class property. There is an abstract method signature called initPlugin() that each Plugin will need to implement though. The initPlugin() accepts two parameters, a JBJF Plugin Definition object and the current Log4j logger for the parent batch process.

```
    /* (non-Javadoc)
     * @see org.jbjf.services.api.IJBJFService#initService(java.util.HashMap)
     */
    //TODO: What to pass as a parameter into the initService...
    public abstract void initPlugin (
         JBJFPluginDefinition jobPlugin
        ,APILog4j             logFile
    ) throws Exception;
```

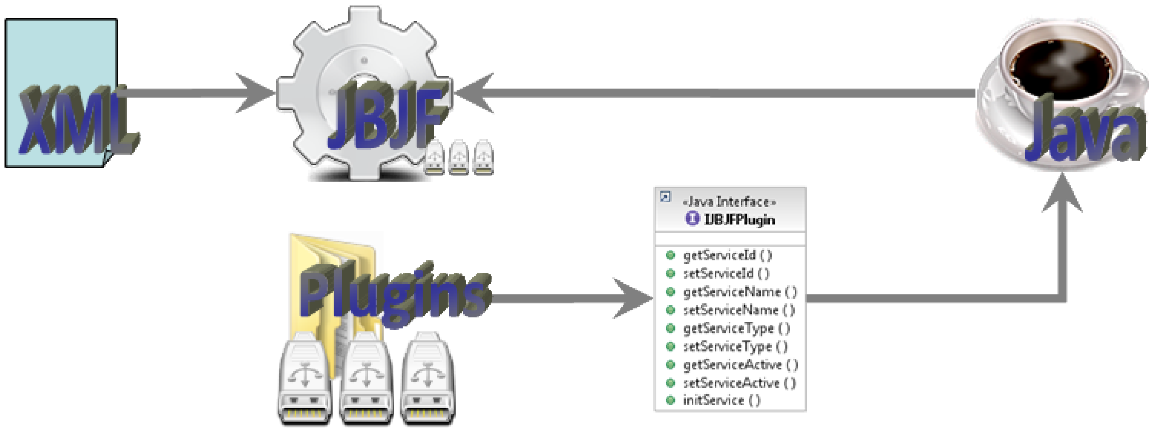
The individual Plugin categories (Cipher, Database, etc...) then extend the IJBJFPlugin and declare specific methods and properties for those Plugin types. Thus, IJBJFPluginCipher provides methods for encrypting and decrypting Strings as well as getters/setters for the encryption key. Similar methods for the IJBJFPluginDatabase, methods for getting a connection and getters/setters for the individual JDBC connection properties such as Client, Server, Port, etc... The following class diagram illustrates the approach:


«Java Interface»
IJBJFPlugin
getPluginId ()
setPluginId ()
getPluginName ()
setPluginName ()
getPluginType ()
setPluginType ()
getPluginActive ()
setPluginActive ()
initPlugin ()
«Java Interface»
IJBJFPluginDatabase
setDatabaseClient ()
setDatabasePass ()
setDatabaseUser ()
setDatabaseDriver ()
setDatabaseServer ()
setDatabasePort ()
setDatabaseName ()
setDatabaseType ()
getDatabaseClient ()
getDatabasePass ()
getDatabaseUser ()
getDatabaseDriver ()
getDatabaseServer ()
getDatabasePort ()
getDatabaseName ()
getDatabaseType ()
setProperties ()
getConnection ()
closeConnection ()
«Java Class»
IAbstractJBJFPlugin
ID : String
SHORT_NAME : String
SYSTEM_IDENTITY : String
activeIndicator : boolean
serviceName : String
serviceType : String
serviceId : String
logger : Logger
getPluginActive ()
getPluginId ()
getPluginName ()
getPluginType ()
initPlugin ()
setPluginActive ()
setPluginId ()
setPluginName ()
setPluginType ()
getLog ()
«Java Interface»
IJBJFPluginCipher
encryptString ()
decryptString ()
setCipherKey ()
getCipherKey ()
«Java Class»
IAbstractJBJFPluginDatabase
ID : String
SHORT_NAME : String
SYSTEM_IDENTITY : String
databaseName : String
databaseType : String
databaseClient : String
databasePort : String
databaseDriver : String
databaseUser : String
databasePass : String
databaseServer : String
databaseProps : HashMap
dbConnection : Connection
IAbstractJBJFPluginDatabase ()
getConnection ()
getDatabaseClient ()
getDatabaseDriver ()
getDatabaseName ()
getDatabasePass ()
getDatabasePort ()
getDatabaseServer ()
getDatabaseType ()
getDatabaseUser ()
setProperties ()
setDatabaseClient ()
setDatabaseDriver ()
setDatabaseName ()
setDatabasePass ()
setDatabasePort ()
setDatabaseServer ()
setDatabaseType ()
setDatabaseUser ()
«Java Class»
IAbstractJBJFPluginCipher
ID : String
SHORT_NAME : String
SYSTEM_IDENTITY : String
cipherKey : String
IAbstractJBJFPluginCipher ()
decryptString ()
encryptString ()
getCipherKey ()
setCipherKey ()

Again, you'll notice for each Plugin Interface, there is a supporting Abstract that provides a partial implementation of the Interface. The Abstracts primarily implement getters/setters to manage the properties only, the remaining method(s) must be implemented by your concrete Plugin class. This is by design.

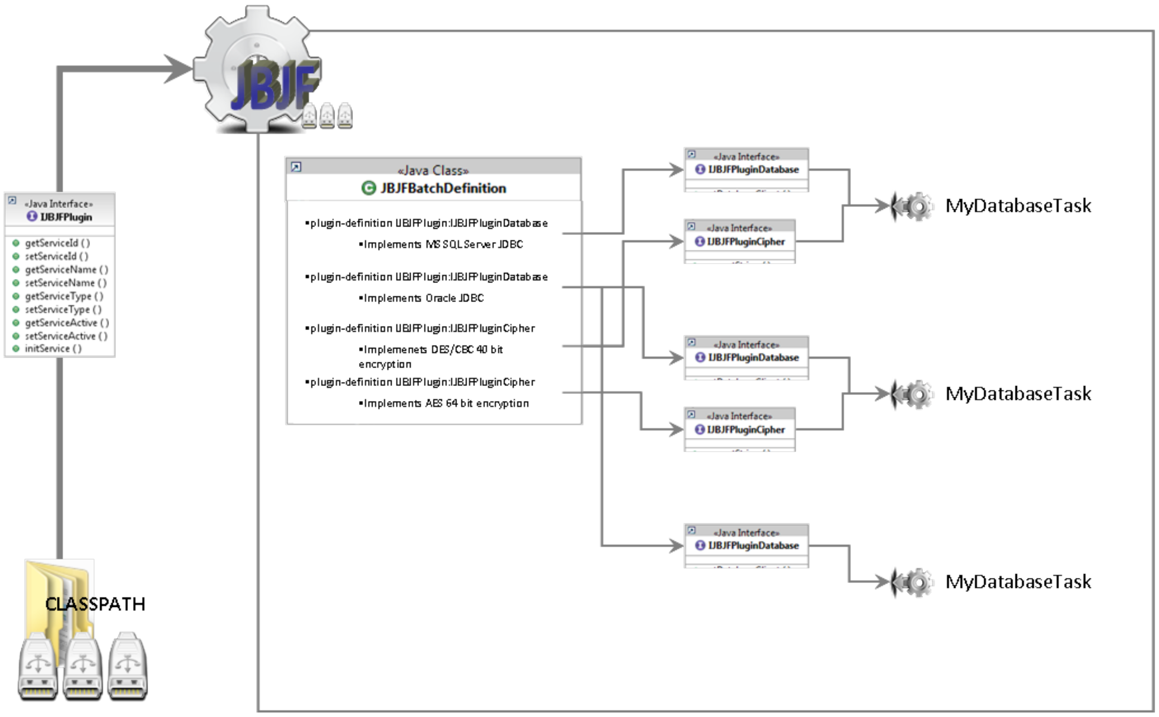
At runtime, the IJBJFPlugin interface is used by the JRE/JDK to identify the Plugins and then load them into the classpath dynamically. From there the JBJF uses the extended interfaces (IJBJFPluginCipher and IJBJFPluginDatabase) to cast and utilize the Plugins with their specific batch process and/or tasks.

The following diagram illustrates the IJBJFPlugin usage:



Once the Plugins have been identified and added to the CLASSPATH, the JBJF can allocate them to the specific batch and task using the specific Plugin Interface/Abstract.

The following diagram illustrates this concept:



In the above example we've got three database tasks involving four different plugins. The first task uses the MS SQL Server database with an encryption Cipher that implements the DES/CBC. The second task, an Oracle database with the AES cipher. The third with the Oracle database only. In all cases the same Task class is used, but different databases and encryption. A couple of key concepts from this are: Database Tasks now have greater reuse with regards to database platforms and encryption ciphers. Before Plugins, a database task usually had to have intimate knowledge about the database platform and the encryption was limited to a single JBJF class.

To summarize, JBJF Plugins are developed against the Interfaces/Abstracts and built outside the JBJF Framework, typically in an individual Java Project. These Plugins then get dropped into a special "plugins" directory (currently ./plugins) reachable by the JBJF runtime library. When JBJF starts up, it searches the "plugins" directory for any classes that fit the IJBJFPlugin interface. These are then added to the classpath and available to your batch process. Finally, your individual JBJF Batch Definition file identifies any necessary Plugins for the batch process via the <jbjf-plugins> element. These plugin classes are located via the id attribute and cast into the runtime instance using the type attribute of the plugin-definition.

Essential Concepts

There are some concepts and procedures that need to be learned in order to fully understand the JBJF Plugins and how they work.

Why Plugins?

Why use Plugins? One of the most nagging issues with JBJF is extensibility. Under the original JBJF, a Developer had to have intimate knowledge of JBJF in order to extend it. For example, to implement a different Cipher object, a developer had to write a class that interfaced with the encryption library. This class then had to be named APIEncryption(), essentially replacing the pre-packaged class in JBJF. This also prevented any chance of using multiple encryption routines within a single batch process without some heavy coding. Even if you coded a new Encryption class, the Task classes that used it had to break the re-use contract by now coding directly to the specific Encryption API thus forming a dependency between them. If someone wanted to re-use your Task, the specific Encryption API had to go with it...This totally defeats and undermines the reusability concept for JBJF. Yuck!

By utilizing Plugins and standard Interfaces (API), we can now design, develop and build Plugins outside the JBJF code. We only need a JBJF runtime with the Plugin Interfaces to be on the Build Path of our Plugin project. We then create a Plugin class that extends or implements the correct Abstract/Interface. Setup the META-INF service file. Build the project to a jarfile and place the jarfile into the “plugins” directory for JBJF. Finally, add the <jbjf-plugins> elements to our JBJF Batch Definition file and allocate the plugins to the proper tasks. Because the JBJF and specifically the task classes are coded to the Plugin Interfaces, we can now reuse those tasks with any plugins of the same nature/category.

META-INF/services

Another critical concept is really associated with the JRE/JDK. The Services API provides the “plugin” piece of the puzzle. By using a special directory within our Plugin project called META-INF/services/, we can place our plugin “instructions” in a simple text file inside this special directory.

The individual Plugin projects are then built into one or more jarfiles (thus including the META-INF “instructions”) and placed into the JBJF “plugins” directory. During the startup cycle of JBJF, the JRE/JDK Services API is initialized with this “plugins” directory. Once initialized JBJF can instruct the Services API to scan, pickup, initialize and dynamically add the Plugins classes to the CLASSPATH. Concurrently, the JBJF then takes the runtime Plugin classes and stores them in a Plugin cache for the batch process.

Java Batch Job Definition file

Of course introducing new architecture into the JBJF framework poses certain risks as well as impacting existing technology. The first and most profound impact of Plugins is the JBJF Batch Definition file. New XML elements are being introduced to identify and manage the Plugins for a given batch process. From the Batch Definition file, Plugins are scanned, rendered into a runtime form and cached. From this cache individual tasks can access and utilize any plugins they need, casting them into the specific Plugin type they require.

XML Elements

The XML Elements follow the same naming conventions and structure of other XML elements. The top level XML element is jbjf-plugins, providing a single XML element that holds all the individual plugin XML elements.

```
<jbjf-plugins>
    <...
        id="plugin001"
        name="default-cipher"
        type="org.jbjf.plugin.IJBJFPluginCipher"
        active="true">
        <class>org.jbjf.services.impl.DefaultCipher</class>
    </...>
</jbjf-plugins>
```

Within this top level element are individual XML elements that identify each individual Plugin that the current batch process needs. These "plugin-definitions" include the four properties that every plugin (no matter what type) must contain:

- Id
- Name
- Type
- Active

You may remember these four properties discussed at the beginning of this User Guide. Along with these properties is a single XML element, class, that contains the fully qualified class name to the Plugin class:

```
        <plugin-definition
            id="plugin001"
            name="default-cipher"
            type="org.jbjf.plugin.IJBJFPluginCipher"
            active="true">
            <class>org.jbjf.services.impl.DefaultCipher</class>
        </plugin-definition>
```

JBJF uses the XML elements to identify which Plugins are needed for this batch process. Thus, only those Plugins needed by the batch process are loaded. Any other Plugins that happen to be scanned will not be loaded or cached for use. This is a very important concept, meaning you can't just drop plugin jarfiles into the "plugins" directory and expect them to magically appear in your batch process(es). The “plugins” directory is designed as a repository, thus there could be many plugins in there, used across many different batch processes. We don’t want a batch process to have to load a bunch of unnecessary plugins, when it only needs a handful.

The <task> element is also impacted. For the initial release of Plugins, multiple plugins can be indicated by using the plugin- prefix in your <resource> element. Future JBJF versions will focus on setting up a “plugin cache” that can be reached by all tasks.

The task XML element now supports a new <resource> type with a prefix of “plugin-“. The XML node value is of course the unique id discussed earlier in the IJBJFPlugin interface.

```
        <task name="t006" order="6" active="true">
            <class>org.jbjf.tasks.RunBasicSQLStatement</class>
            <resource type="sql-definition">my-sql-statement</resource>
            <resource type="connection">my-oracle-db</resource>
            <resource type="sql-results">jsk-sql-results</resource>
            <resource type="plugin-database">oracle-db</resource>
            <resource type="plugin-cipher">default-cipher</resource>
        </task>
```

Like other pre-defined resources in the JBJF Batch Definition file, the plugin- will get automatically loaded as part of the task initialization. You don’t have to explicitly fetch the resource. Like other resources, you can fetch the plugin-definition using the getResources() method of the task and cast it into the specific JBJF Plugin Type object you need. However, the runtime instance is also available from the Task level cache and can be fetched directly into a runtime plugin state:

```
    /* (non-Javadoc)
     * @see org.jbjf.core.AbstractTask#runTask(java.util.HashMap)
     */
    @Override
    public void runTask(HashMap pjobParameters) throws Exception {
        getLog().debug( SHORT_NAME + " ... Start ..." );
        String  lpluginKey = (String)getResources().get("plugin-data-
base");
        String  lconnKey = (String)getResources().get("connection");

        IJBJFPluginDatabase theDB =
getTaskPlugins().getDatabasePlugin(lpluginKey);
        theDB.setProperties( (~~JBJFDatabaseConnection~~)getDefinition().ge
tConnections().get(lconnKey) );
```

You can see in the above code snippet where we grab the plugin-definition key, lpluginKey. This can then be used to fetch a runtime instance of the plugin from the Task level cache:

```
IJBJFPluginDatabase theDB = getTaskPlugins().getDatabasePlugin(lpluginKey);
```

This example is used in the Junit test class, junit.jbjf.tasks.MyFirstPluginTask in the testing source folder.

Runtime Plugins

The XML elements within the JBJF Batch Definition file will drive the JBJF to load and render the plugins into a runtime instance of the class. All runtime instances are cast into a generic IJBJFPlugin and stored at the batch level using the JBJFBatchDefinition object, accessible by the getPluginRuntimes() method. This method returns a special collection See the Javadocs for complete details.

As Plugin Definitions are encountered within the <task> elements, these are then pre-processed by AbstractTask during the initialization phase and placed into a local “task” store, accessible by getTaskPlugins(). Again, see the Javadocs for complete details.

```
<jbjf-plugins>
    <...
        id="plugin001"
        name="default-cipher"
        type="org.jbjf.plugin.IJBJFPluginCipher"
        active="true">
        <class>org.jbjf.services.impl.DefaultCipher</class>
    </...>
</jbjf-plugins>
```

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
Plugins	1.0.0	In progress