Technical Note TN2027

How to Write a JDBC Plugin (With Example)

CONTENTS

With WebObjects 5, all database access goes through the JDBC Adaptor. JDBC is widely supported by database vendors, but not every data source behaves exactly the same way so the adaptor uses a helper class, a subclass of JDBCPlugIn, to customize the JDBC adaptor for a particular database or driver. The JDBC Adaptor for WebObjects 5 ships with built-in support for Oracle 8i and OpenBase. Other data sources may require additional support from custom plugins. This article describes how to build a custom plugin.

Updated: [Jul 11 2001]

Two JDBC Adaptors

In WebObjects 5, all of the frameworks are written in pure Java, and naturally your WebObjects applications use pure Java. However, for historical reasons, some of the development tools such as EOModeler are written in Objective-C. Because of this, EOModeler needs to use the Java Bridge to support access to JDBC. That means that there are actually two different JDBC Adaptors on your system: one for the normal Java runtime, and a second for use only by EOModeler. This scheme is transparent to most users, but it has some implications for plugin writers.

In this document, we refer to the pure Java JDBC Adaptor used at runtime as the "JDBC Adaptor". (This framework is installed on your system as **/System/Library/Frameworks/JavaJDBCAdaptor.framework.**) The other adaptor is the "Bridged JDBC Adaptor" which is used only by EOModeler. (The "Bridged" adaptor is installed on your system as **/System/Library/Frameworks/JDBCEOAdaptor.framework.**) The Bridged JDBC Adaptor does not use any of the pure Java WebObjects frameworks. Instead, it uses the Java wrappers for Foundation and a private Objective-C EOAccess. These wrappers support enough functionality to make the Bridged JDBC Adaptor work for EOModeler. They are not a full implementation of EOF for the Java Bridge. They are considered private to the WebObjects development tools.

Just to clarify, you should always link your WebObjects applications against JavaJDBCAdaptor, not JDBCEOAdaptor. Only EOModeler and custom bundles for EOModeler, which we will discuss below, use JDBCEOAdaptor.

Back to top

JDBCPlugIn

First, we will discuss, how to make a subclass of JDBCPlugIn for use by WebObjects applications, and then we will discuss how to make that plugin functionality available to EOModeler.

You will need to make a subclass of **com.webobjects.jdbcadaptor.JDBCPlugIn**. See the method descriptions accompanying this document for more information. Your plugin can be added to your application's code, or it can be built as part of framework that your application will link against.

In the connection dictionary of your model, you will need to specify the name of your JDBCPlugIn subclass for the "plugin" key. If the value is not a fully qualified class name, the adaptor will try to find a class in the **com.webobjects.jdbcadaptor** package, and will try to append "PlugIn" to the name if it's not already there.

The plugin class is primarily determined by the connection dictionary's "plugin" entry. If there is no value, then a plugin name is guessed from the subprotocol in the "URL" value of the connection dictionary. The JDBC URL always starts with "jdbc:" followed by a subprotocol term, another colon and the rest of the URL. The usual guess for the plugin name is to capitalize the subprotocol and add "PlugIn". For example, "**jdbc:mydb:foobar**" implies a plugin name of "**com.webobjects.jdbcadaptor.MydbPlugIn**". You can control this heuristic by using the static method `JDBCPlugIn.setPlugInNameForSubprotocol()` in your WOApplication.

Once the plugin is chosen, the adaptor will attempt to load a JDBC driver. The driver normally is chosen according to the plugin's implementation of `defaultDriverName()`. However, the user can override this choice by setting a value for "driver" in the connection dictionary.

Most common operations such as fetching, inserting, and updating rows in the database are handled generically using JDBC methods. Your custom plugin can provide its own SQL expression class via `defaultExpressionClass()` to handle any special syntax for the datasource. Normally, you will want to use the default expression factory (`JDBCExpressionFactory`) in conjunction with your custom `JDBCExpression` subclass.

Some aspects of creating tables or reverse engineering will require custom code. There are several methods in the plugin that control how reverse engineering works. Your custom plugin may also provide a custom `EOSynchronizationFactory` subclass via `createSychronizationFactory()` to specialize the schema synchronization functions. It's unusual for a WOApplication to need schema synchronization at runtime so it's perfectly reasonable to use the default implementation which essentially does nothing for schema synchronization.

Back to top

Using Your Custom PlugIn within EOModeler

As explained earlier, EOModeler uses the Bridged JDBC Adaptor and does not have access to your pure Java plugin. You will need to provide a special kind of NSBundle to make your plugin code work within EOModeler. The example project (in Project Builder format) demonstrates how to modify a copy of your pure Java source into a Bridged plugin bundle. (You will still use the pure Java version for your WOApplication project.)

The first step is to create a new project in Project Builder. This should be a Cocoa Bundle. Look at the example project for details.

Here are a few things to note:

You should have a main.c file, even though it contains no source code. This makes the bundle build correctly so that your Bridged Java PlugIn can be loaded in EOModeler.

This bundle should link against the **JDBCEOAdaptor.framework** and the **Foundation.framework**.

There should be one java file for each Bridged JDBC PlugIn that you want to make available in EOModeler. You can have multiple plugins in a single bundle.

Put your plugin code into the top-level package. That is, do not use a package statement in the plugin source file.

Your imports should use the Java wrapper packages, not the pure WebObjects 5 packages. Here's a typical list:

```
import com.apple.cocoa.foundation.*;  // not com.webobjects.foundation
import com.apple.yellow.eoaccess.*;   // not com.webobjects.eoaccess
import com.apple.yellow.eocontrol.*;  // not com.webobjects.eocontrol
```

Be aware that some of the wrapped methods that return collections such as NSArray or NSDictionary will sometimes return null to indicate the empty collection. (This was a common idiom in Objective-C.) Your code needs to be more null-tolerant than the pure Java WebObjects 5 code. Always check for null before using a returned NSArray or NSDictionary. Your code should return empty collections, not null.

The transformation of your pure Java source into Bridged Java requires a few other changes to match the Java Wrappers:

```
replace "NSKeyValueCoding.NullValue" with "HackedUtils.NullValue"
replace "NSArray.EmptyArray" with "HackedUtils.EmptyArray"
replace "NSDictionary.EmptyDictionary" with "HackedUtils.EmptyDictionary"
```

The created bundle should have a file name with the extension ".EOMplugin". It must be installed in a place that EOModeler expects to find bundles, typically in /Developer/EOMBundles/. You can use a symlink if you want to keep the bundle somewhere else on your disk.

Note that these Bridged JDBCPlugIns are only used by EOModeler. The WebObjects 5 runtime needs a pure Java versions of the JDBCPlugIn linked in with your application. The source code is similar but the binary is not the same.

Creating the Bridged JDBCPlugIn for use on Windows is somewhat more complicated. (The pure Java JDBCPlugIn is naturally exactly the same on all platforms.) The Java Bridge on Windows supports on JDK 1.1 and so cannot use the latest JDBC drivers which require JDK 1.2 (or Java 2) support. You must use ProjectBuilderWO since the new Project Builder is not available. The packaging is also slightly different for the Java wrappers on Windows. The Foundation wrapper is in the **package com.apple.yellow.foundation** ("yellow" as opposed to "cocoa").

You custom plugin will typically use its own expression class, which should be a subclass of `JDBCExpression`. To support EOModeler access to your bridged plugin, you must implement the following methods in your Bridged expression class (in addition to the normal API). Your expression class will probably be an inner static class of your plugin class. See the example code for more details.

```
// code fragments from the inner class of CustomPlugIn for Java Bridge support
public static class CustomExpression extends JDBCExpression
  // no-arg constructor
  public CustomExpression()
  public static EOSQLExpression sharedInstance()
```

```
    public static EOSynchronizationFactory sharedSyncFactory()
    public Class _synchronizationFactoryClass()
```

If you choose to implement your own `EOSynchronizationFactory` subclass, it will typically be another inner static class of your plugin. You must add a no-arg constructor to the class (only for the bridged version). See the example code for more details.

```
 // code fragments from the inner class of CustomPlugIn for Java Bridge support
  public static class CustomSynchronizationFactory extends EOSynchronizationFactory
    // no-arg constructor
    public CustomSynchronizationFactory()
```

Back to top

Debugging

While developing a "Bridged" plugin, it may be useful to turn on `EOAdaptorDebuggingEnabled`. One easy way to do this is to execute the following command in a terminal window (assuming that "% " is your csh prompt)....

```
% defaults write NSGlobalDomain EOAdaptorDebugEnabled YES
```

This will cause the Bridged JDBC Adaptor to log extra information into your Console window. You can stop this logging by issuing another command in your terminal window....

```
% defaults write NSGlobalDomain EOAdaptorDebugEnabled NO
```

Consult the man page on defaults if you want more information. (By the way, the domain for EOModeler is "com.apple.EOModeler".)

Back to top

Summary

This article describes how to create a JDBCPlugIn for the WebObjects 5 JDBC Adaptor. It also explains how to make a "Bridged" version of your plugin into an NSBundle so that it can be used within EOModeler. Sample code shows how to build the special NSBundle for EOModeler.

Back to top

Downloadables

	Acrobat version of this Note (60K)	Download
	Binhexed Metrowerks Project File (24K)	Download

Back to top

Appendix A: Method descriptions for JDBCPlugIn

```
// This file contains comments and method signatures only.  A future
// release of WebObjects will contain the usual JavaDoc for the
// JDBCAdaptor.

// package com.webobjects.jdbcadaptor;
// public class JDBCPlugIn


/**
* Constructor, typically just call super from a subclass.
*/
public JDBCPlugIn(JDBCAdaptor adaptor)


/**
* Returns a string identifying the database.
```

```
*/
public String databaseProductName()


/**
 * Returns a fully qualified name of the driver class that this
 * plugin prefers to use.  The adaptor will attempt to load this
 * class when making a connection.
 */
public String defaultDriverName()


/**
 * Default just returns the URL from the connection dictionary.  Subclasses
 * don't normally override this.
 */
public String connectionURL()


/**
 * Subclasses will typically override to use their own expression class.
 */
public Class defaultExpressionClass()


/**
 * Default returns a JDBCExpresionFactory.  Subclasses rarely need to
 * override this.
 */
public EOSQLExpressionFactory createExpressionFactory()


/**
 * Subclasses will typically override to use their own synchronization factory
 */
public EOSynchronizationFactory createSynchronizationFactory()


/**
 * Default returns "EO_PK_TABLE".  Subclasses typically don't override
 * this.  See also newPrimaryKeys().
 */
public String primaryKeyTableName()


// The following four methods are called internally by
// JDBCChannel.describeTableNames()

/**
 * Returns the SQL statement that should be used for getting the list
 * of available tables in a database.  Default returns null, in which
 * case the standard JDBC getTables() method will be used.
 */
public String sqlStatementForGettingTableNames()


/**
 * Returns the String to use for describing the schema pattern in
 * the JDBC getTables() method call. By default it returns null.
 */
public String wildcardPatternForSchema()


/**
 * Returns the String to use for describing the table pattern in
 * the JDBC getTables() method call.  By default it returns '%'.
 */
public String wildcardPatternForTables()


/**
 * Returns a language array or Strings defining the types used in the
 * call to the JDBC getTables() method.  Default returns { "TABLE",
 * "VIEW", "ALIAS", "SYNONYM"};
 */
```

```
public String[] tableTypes()



/**
 * Return the String to use for describing the column pattern
 * name in the JDBC method getColumns().  The default
 * implementation returns '%'.
 */
public String wildcardPatternForAttributes()


/**
 * Returns the pattern string for the catalog of the stored procedures
 * looked up with the standard JDBC getProcedures() method.  The
 * default is null.
 */
public String storedProcedureCatalogPattern()


/**
 * Returns the pattern string for the schema of the stored procedures
 * looked up with the standard JDBC getProcedures() method.  The
 * default is null.
 */
public String storedProcedureSchemaPattern()


/**
 * This method should return an NSArray of NSDictionary for 'count'
 * number of newly inserted object of kind 'entity', where each
 * NSDictionary is suitable to be used as the primary key for the
 * entity.  The default implementation uses a table name given by the
 * primaryKeyTableName() method containing the root entity name and
 * the last primary key inserted into the table. If the row for entity
 * name is not present, then it is automatically created. If the table
 * does not exists, then it is automatically created.  If the primary
 * key contains multiple attributes or is not a Number, or if
 * the adaptor is not able to provide 'count' number of new primary
 * key values, then the default implementation returns null.
 */
public NSArray newPrimaryKeys(int count, EOEntity entity, JDBCChannel channel)


/**
 * Returns an NSDictionary of meta information (including type
 * information) about the datasource.  A subclass usually does not
 * need to overrides this method, but if it does, it should normally
 * call super.
 */
public NSDictionary jdbcInfo()


/**
 * Returns false by default.  Subclasses might override.
 * A pseudo column is an extra column that's created by the database
 * beyond those that were defined in the CREATE TABLE statment.  They're
 * used for internal bookkeeping and are usually read-only.  The adaptor
 * will not reverse engineer the column if this method returns true for
 * its name.
 */
public boolean isPseudoColumnName(String columnName)




// Don't override the following methods...

/**
* Calls createExpressionFactory() when needed and caches result
* Subclasses should not override.
*/
public EOSQLExpressionFactory expressionFactory()
```

```
/**
 * Calls createSynchronizationFactory() when needed and caches
 * result.  Subclasses should not override.
 */
public EOSynchronizationFactory synchronizationFactory()

/**
 * Returns the JDBCAdaptor that is using the plugin.
 */
public JDBCAdaptor adaptor()



// Utility methods for plugin name heuristic

/**
 * Sets the internal mapping of subprotocol to pluginName for guessing
 * which plugin to use.
 */
public static void setPlugInNameForSubprotocol(String pluginName, String subprotocol)


/**
 * Clears the internal mapping of any special plugin to use for
 * subprotocol, restoring the default behavior.
 */
public static void removePlugInNameForSubprotocol(String subprotocol)
```

Back to top
