Sun GlassFish Mobility Platform 1.1 Developer's Guide for Enterprise Connectors

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820–7207
April 2009

090413@21990

Contents

Preface

This guide explains how to develop Enterprise Connectors for Sun GlassFish Mobility Platform 1.1.

Sun GlassFish Mobility Platformis a comprehensive mobility solution that enables offline data access, data synchronization, and secure access to EIS/EAI, ERP, CRM, and database applications.

Sun GlassFish Mobility Platform is based entirely upon open standards, including the following:

- Java Platform, Mobile Edition (Java ME)
- Java Platform, Enterprise Edition (Java EE)
- The dominant industry standard OMA DS and its SyncML protocols. The specifications for Open Mobile Alliance Data Synchronization V1.1.2 and V1.2.1 are available at `http://www.openmobilealliance.org/Technical/release_program/ds_v112.aspx` and `http://www.openmobilealliance.org/Technical/release_program/ds_v12.aspx`.

Who Should Use This Book

This guide is intended for developers who have experience creating Java applications.

Sun GlassFish Mobility Platform Documentation

The Sun GlassFish Mobility Platform 1.1 documentation set is available at `http://docs.sun.com/coll/1918.1`. To learn about Sun GlassFish Mobility Platform, refer to the books listed in the following table.

TABLE P–1 Books in the Sun GlassFish Mobility Platform Documentation Set

Book Title	Description
Sun GlassFish Mobility Platform 1.1 Release Notes	Late-breaking information about the software and the documentation. Includes a comprehensive summary of the supported hardware, operating systems, application server, Java™ Development Kit (JDK™), databases, and EIS/EAI systems.
Sun GlassFish Mobility Platform 1.1 Architectural Overview	Introduction to the architecture of Sun GlassFish Mobility Platform.
Sun GlassFish Mobility Platform 1.1 Installation Guide	Installing the software and its components, and running a simple application to verify that installation succeeded.
Sun GlassFish Mobility Platform 1.1 Deployment Guide	Deployment of applications and application components to Sun GlassFish Mobility Platform.
Sun Glassfish Mobility Platform 1.1 Developer's Guide for Client Applications	Creating and implementing Java Platform, Mobile Edition (Java ME platform) applications for Sun GlassFish Mobility Platform that run on mobile devices.
Sun Glassfish Mobility Platform 1.1 Developer's Guide for Enterprise Connectors	Creating and implementing Enterprise Connectors for Sun GlassFish Mobility Platform intended to run on Sun GlassFish Enterprise Server.
Sun GlassFish Mobility Platform 1.1 Administration Guide	System administration for Sun GlassFish Mobility Platform, focusing on the use of the Sun GlassFish Mobility Platform Administration Console.

For up-to-the-minute information about Sun GlassFish Mobility Platform from the Sun GlassFish Mobility Platform technical team at Sun, see the Enterprise Mobility Blog at `http://blogs.sun.com/mobility/`.

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. To share your comments, go to `http://docs.sun.com` and click Feedback.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–2 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–3 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

◆ ◆ ◆ CHAPTER 1

Introduction to Enterprise Connectors

To provide an interface between data on mobile client devices and data stored in a database or an EIS/EAI system, you create an Enterprise Connector. The Enterprise Connector can be implemented in either of the following ways:

- Using the Java API for RESTful Web Services (JAX-RS)
- Using the Enterprise Connector Business Object (ECBO) API that is provided by Sun GlassFish Mobility Platform.

In conjunction with a client application developed using the Mobile Client Business Object API (MCBO) and, optionally, the JerseyME API, the Enterprise Connector provides a complete solution that allows you to synchronize arbitrary enterprise data between a client device and a database or EIS system, using a Sun GlassFish Mobility Platform Gateway Engine (the Gateway) deployed on the Sun GlassFish Enterprise Server that acts as an intermediary. The Gateway determines what objects need to be synchronized, while the Enterprise Connector implements the operations required to keep the client and the back-end system in sync.

You can use an Enterprise Connector in many ways, including the following:

- To communicate with an EIS/EAI system using a Sun JCA Adapter
- To store and retrieve data in a database using the JDBC API
- To store and retrieve data on the file system
- To communicate with an EIS/EAI system using JAX-WS

This chapter covers the following topics:

Developing Connectors Using the Java API for RESTful Web Services (JAX-RS)

You can create Enterprise Connectors that use the Java API for RESTful Web Services (JAX-RS). A JAX-RS Enterprise Connector is a web application that consists of classes that are packaged as a servlet in a WAR file along with required libraries.

Two classes in the Enterprise Connector use the container-item design pattern to define the resource classes required by JAX-RS. These resource classes include the methods needed to perform Create, Retrieve, Update, and Delete (CRUD) operations on the business objects.

One or more classes in the Enterprise Connector may define the business object or objects, which are the data to be synchronized. The way in which these arbitrary Java objects are serialized is open-ended and is part of the contract between an Enterprise Connector and its corresponding Java ME client. Because the client usually uses the MCBO API, the definition of the business objects is usually similar to that of the client business objects. It is recommended that you use a format that supports data versioning.

The Enterprise Connector may contain additional utility classes.

You can use NetBeans IDE to develop a JAX-RS Enterprise Connector. For details about developing JAX-RS Enterprise Connectors, see Chapter 2, "Creating an Enterprise Connector Using the Java API for RESTful Web Services (JAX-RS)."

Packaging and Deploying JAX-RS Enterprise Connectors

To package an Enterprise Connector that uses the JAX-RS API, compile the Java classes and include them in a WAR file.

To see how the sample JAX-RS Enterprise Connectors provided with Sun GlassFish Mobility Platform are packaged, use the `jar` command or another utility to view the contents of either of the following files in the `gateway` subdirectory of the directory where Sun GlassFish Mobility Platform was unzipped:

```
musicdb-ws.war
salesforce-ws.war
```

Alternatively, you can see the deployed versions by looking in the `applications/j2ee-modules` directory of the Enterprise Server domain for Sun GlassFish Mobility Platform.

The Maven plugin for NetBeans IDE makes available a set of Maven archetypes (templates) that you can use to develop JAX-RS Enterprise Connectors.

Figure 1–1 shows the location of a JAX-RS Enterprise Connector in a single-tier installation. The client communicates with the Gateway tier in two different ways: it can use the MCBO API to communicate with the Gateway, or it can use the JerseyME API to communicate directly with the Enterprise Connector using the JAX-RS API. The Gateway communicates with the Enterprise Connector using the JAX-RS API. The Enterprise Connector in turn communicates with a back-end system using an appropriate application protocol.

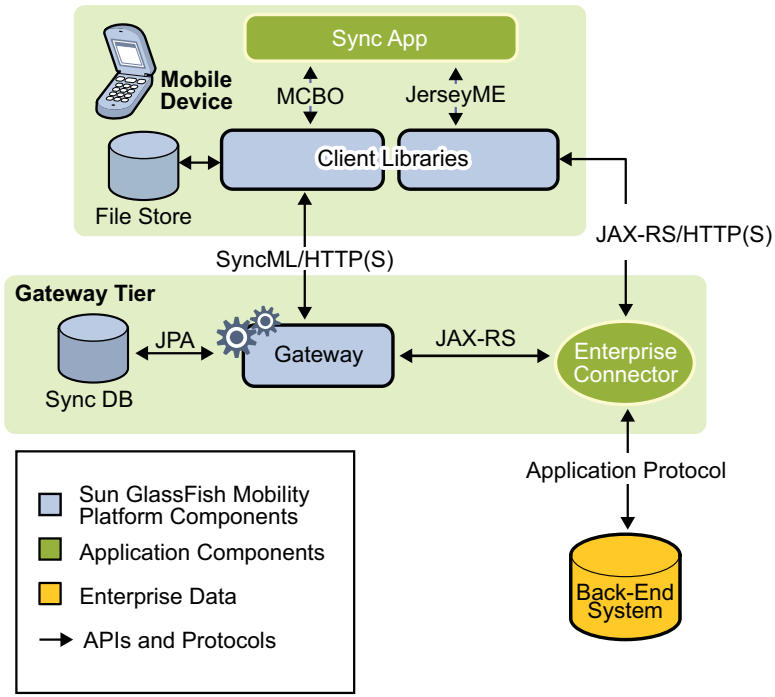

FIGURE 1–1 A JAX-RS Enterprise Connector in a Single-tier Sun GlassFish Mobility Platform Installation

In addition, if the back-end system has a RESTful interface, the client can use the JerseyME API to communicate directly with the back-end system.

You can use a JAX-RS Enterprise Connector in either a single-tier or a two-tier Sun GlassFish Mobility Platform installation. You do not need to use a two-tier installation, however, in order to use a JAX-RS Enterprise Connector remotely. You can deploy the Enterprise Connector on a different system from the one where the Gateway is installed, and then configure the Gateway to point to the URL of the system where the Enterprise Connector is deployed.

To configure your deployed Enterprise Connector, use the Sun GlassFish Enterprise Server Administration Console and the Sun GlassFish Mobility Platform Administration Console. For details, see "Building, Deploying, and Configuring a JAX-RS Enterprise Connector" on page 35.

Developing Connectors Using the Enterprise Connector Business Object (ECBO) API

The Enterprise Connector Business Object (ECBO) API provides classes and methods that allow you to create an Enterprise Connector. An ECBO Enterprise Connector is a Java artifact that consists of extensions of five classes compiled and packaged into a RAR file as a resource adapter, along with other files needed for the resource adapter.

The methods you implement in your Enterprise Connector are called by the Sun GlassFish Mobility Platform libraries deployed on the Gateway. In this respect, the ECBO API is more like a Service Provider Interface (SPI) than an Application Programmer Interface (API).

Each Enterprise Connector defines a particular business object whose data is to be synchronized. It also defines commands and operations that allow the Enterprise Connector to perform Create, Retrieve, Update, and Delete (CRUD) operations on the business objects.

The five classes you must implement are called `BusinessObject`, `BusinessObjectProvider`, `InsertCommand`, `UpdateCommand`, and `DeleteCommand`. The `BusinessObjectProvider` class provides a method to perform the retrieve operation. Chapter 3, "Creating an Enterprise Connector Using the Enterprise Connector Business Objects (ECBO) API," describes in detail how to implement these classes.

The resource file is an XML file that is used by the underlying implementation. It defines a repository name, a workspace name, and a few node types. All Enterprise Connectors use the same set of node types.

The data to be synchronized takes the form of arbitrary Java objects. The way in which these Java objects are serialized is open-ended and is part of the contract between an Enterprise Connector and its corresponding Java ME client. It is recommended that you use a format that supports data versioning.

For details about developing ECBO Enterprise Connectors, see Chapter 3, "Creating an Enterprise Connector Using the Enterprise Connector Business Objects (ECBO) API."

Packaging and Deploying ECBO Enterprise Connectors

To package an Enterprise Connector that uses the ECBO API, compile the five Java classes. Then create a resource adapter using the Java Connector Architecture and include the five classes and the ECBO library in the RAR file for the resource adapter.

To see how the sample ECBO Enterprise Connectors provided with Sun GlassFish Mobility Platform are packaged, use the `jar` command or another utility to view the contents of either of the following files in the `gateway` subdirectory of the directory where the Sun GlassFish Mobility Platform installation bundle was unzipped:

```
ds-jcr-musicdb.rar
ds-jcr-siebel-eway.rar
```

Alternatively, you can see the deployed versions by looking in the `applications/j2ee-modules` directory of the Enterprise Server domain for Sun GlassFish Mobility Platform.

The Maven plugin for NetBeans IDE makes available a set of Maven archetypes (templates) that you can use to develop ECBO Enterprise Connectors.

The same Enterprise Connector works equally well in a one-tier or two-tier Sun GlassFish Mobility Platform installation. (See "Sun GlassFish Mobility Platform Architecture" in *Sun GlassFish Mobility Platform 1.1 Architectural Overview* for details on these two types of installation.) Deploy the Enterprise Connector to the Enterprise Server domain that contains your Sun GlassFish Mobility Platform installation. You can use the Enterprise Server Administration Console or the `asadmin` command to deploy the Enterprise Connector.

In a two-tier Sun GlassFish Mobility Platform installation, an ECBO Enterprise Connector is deployed on the second tier, as shown in Figure 1–2. In this situation, communication between the Gateway and the Enterprise Connector goes through a web service. The Gateway communicates with the web service client. The web service client communicates with the web service endpoint using SOAP/HTTP(S). Finally, the web service endpoint communicates with the Enterprise Connector.

The Enterprise Connector in the figure represents both an ECBO Enterprise Connector and a JAX-RS Enterprise Connector. In a Sun GlassFish Mobility Platform installation, both an ECBO Enterprise Connector and a JAX-RS Enterprise Connector are deployed by default. In this figure, the client application on the mobile device does the following:

- Uses the MCBO API to synchronize data with the ECBO Enterprise Connector
- Uses the JerseyME API to communicate with the JAX-RS Enterprise Connector to retrieve dynamic data, separate from the synchronized data

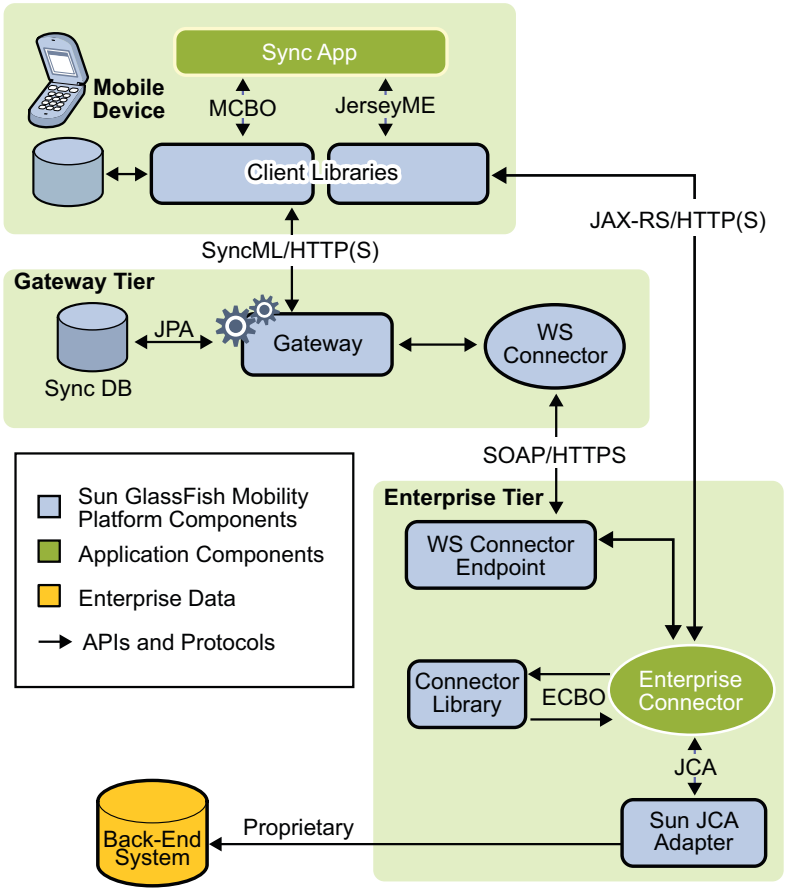

FIGURE 1–2 An Enterprise Connector in a Two-tier Sun GlassFish Mobility Platform Installation

You can deploy an ECBO Enterprise Connector in either a single-tier or a two-tier Sun GlassFish Mobility Platform installation. In order to use an ECBO Enterprise Connector remotely, however, you must deploy it in a two-tier installation, because remote use of an ECBO Enterprise Connector requires the Web Services Connector endpoint to communicate between the Gateway and the remote Enterprise Connector.

To configure your deployed Enterprise Connector, use the Sun GlassFish Mobility Platform Administration Console. For details, see "Deploying and Configuring an ECBO Enterprise Connector" on page 60.

Deciding Which Kind of Enterprise Connector to Use

Consider the following issues in deciding to develop a JAX-RS or an ECBO Enterprise Connector:

Ease of Use

The JAX-RS API is somewhat easier to use than the ECBO API, particularly if you are already familiar with the JAX-RS API. The ECBO API was developed specifically for data synchronization, while the JAX-RS API has many other uses.

Deployment Flexibility

The JAX-RS API provides more flexibility in how you deploy an Enterprise Connector.

If you install Sun GlassFish Mobility Platform and you decide to use the ECBO API, you must remain with your original installation, whether it is a single-tier or a two-tier installation. If you use the ECBO API in a two-tier installation, communication between the Gateway on the first tier and the Enterprise Connector on the second tier requires the Web Services Connector endpoint that is installed as part of the two-tier installation. If you use the ECBO API in a single-tier installation, the Gateway and the Enterprise Connector communicate directly using ECBO API calls.

If you decide to use the JAX-RS API for your Enterprise Connector, however, you gain some flexibility in how you deploy your Enterprise Connector. A JAX-RS Enterprise Connector does not use the Web Services Connector endpoint. Instead, communication between the Gateway and the Enterprise Connector takes place over HTTP/S. This means that you can deploy the Enterprise Connector (and a Sun JCA Adapter, if it uses one) on any system with a GlassFish installation. If you used a single-tier installation, you can deploy the Enterprise Connector locally (on the same system where you installed the Gateway) or remotely (on a different system from the one where you installed the Gateway).

It is easier to develop a JAX-RS Enterprise Connector initially, because you can easily redeploy the Enterprise Connector on the fly, without having to restart the server. Redeploying an ECBO Enterprise Connector, on the other hand, usually requires you to restart the domain when you deploy a new version.

Performance with Local or Remote Deployment

In most cases, an ECBO Enterprise Connector deployed locally (on the same system as the Gateway) will have better performance than a JAX-RS Enterprise Connector deployed locally, because the JAX-RS Enterprise Connector must communicate over HTTP even when it is installed locally.

In most cases, a JAX-RS Enterprise Connector deployed remotely (on a different system from the Gateway) will have better performance than an ECBO Enterprise Connector deployed remotely.

◆ ◆ ◆ CHAPTER 2

Creating an Enterprise Connector Using the Java API for RESTful Web Services (JAX-RS)

A Sun GlassFish Mobility Platform Enterprise Connector that uses the Java API for RESTful Web Services (JAX-RS) typically includes JAX-RS resource classes as well as a Java class that defines the business objects to be synchronized. One of the resource classes may provide a reporting facility. For performance reasons, a connection pool facility may also be provided.

The Java class that defines the business objects fulfils a contract between the Enterprise Connector and the mobile client application that synchronizes data with the back-end system. The serializations of the Enterprise Connector objects must match the serializations of the client application objects, and both kinds of objects must correspond to objects on the back-end system.

The JAX-RS resource classes perform HTTP operations: GET, POST, PUT, and DELETE. These operations are mapped to CRUD operations on the data: create, retrieve, update, and delete.

For details on the JAX-RS API, see the specification at `http://jcp.org/aboutJava/communityprocess/final/jsr311/index.html`. The Reference Implementation (RI), called Jersey, is an open-source project located at `https://jersey.dev.java.net/`. Click the Learn link on the Jersey project page for more information.

This chapter uses the MusicDB JAX-RS sample Enterprise Connector provided with Sun GlassFish Mobility Platform to demonstrate how to develop an Enterprise Connector using JAX-RS. This Enterprise Connector acts as the intermediary between a client application on a mobile device and a back-end system. For this simple application, the back-end system is not a full-fledged EIS/EAI system but an ordinary database that is accessed using the Java Database Connectivity (JDBC) API.

The source code for the MusicDB JAX-RS Enterprise Connector is included in the Sun GlassFish Mobility Platform client bundle. In the directory where you unzipped the client bundle, it is in a ZIP file, `musicdb-ws-3.1.39.zip`, in the subdirectory `sgmp-client-1_1_01-fcs-b02/samples/secure-musicdb/src/connector/jaxrs/`. The ZIP

file unzips as a Maven project. Within this project, the Java source files are in the directory `musicdb-ws-3.1.39/src/main/java/com/sun/mep/ws/musicdb`, in the subdirectories `sync`, `report`, and `util`. The `sync` directory contains the business object classes.

Use NetBeans IDE with the Maven plugin to develop your own Enterprise Connector. The plugin provides templates for the JAX-RS resource classes, but not for the business object.

- "Defining Business Objects" on page 18
- "Using NetBeans IDE to Create a JAX-RS Enterprise Connector" on page 24
- "Defining Operations on Business Objects" on page 26
- "Building, Deploying, and Configuring a JAX-RS Enterprise Connector" on page 35

Defining Business Objects

A business object that you define for a JAX-RS Enterprise Connector represents the object that the Enterprise Connector will synchronize. The object does not need to implement a specific API. However, it needs to match a business object defined by a client application, and the client application typically uses the Mobile Client Business Object (MCBO) API to define its business objects. Therefore, the JAX-RS Enterprise Connector business object looks very similar to that of the client application.

For the MusicDB example, the class that defines the business object is `MusicAlbum`. The `MusicAlbum.java` file begins by importing the Java SE packages it requires:

```
package com.sun.mep.ws.musicdb.sync;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.util.Calendar;
import java.sql.Timestamp;
```

The class code itself begins by declaring a string value to be used by the `serialize` method:

```
public class MusicAlbum {

    private static final String DEFAULT_VALUE = "$$default$$";
```

This example then declares its data properties (there are only five):

```
    /**
     * Album's name.
     */
    String name;
```

```
/**
 * Album's artist.
 */
String artist;

/**
 * Date when the album was published.
 */
Calendar datePublished;

/**
 * Album's rating from 1 to 5.
 */
int rating;

/**
 * Last modified timestamp.
 */
long lastModified;
```

The `lastModified` property is unique to the Enterprise Connector version of the business object; it is not used in the mobile client application. This property allows the Gateway to use timestamps to query the back-end system to determine whether a business object has been updated since the last synchronization. If the Enterprise Connector does not use timestamps, the Gateway computes a digest over the serialized business object, using a hash function, and compares it to the previous digest. Using timestamps is more efficient than using digests.

The code then declares a `StringBuilder`, a user name value, and a flag indicating whether the object will use timestamps or digests:

```
/**
 * String builder used to return SQL commands.
 */
StringBuilder stringBuilder = new StringBuilder();

/**
 * Name of user logged in.
 */
String username;

/**
 * Flag indicating if timestamps should be used instead
 * of digests.
 */
boolean useTimestamps;
```

The user name and the `useTimestamps` flag are used in the `MusicAlbum` business object constructors, which have both two-argument and three-argument forms:

```
public MusicAlbum(String name, String username) {
    this(name, username, false);
}

public MusicAlbum(String username, boolean useTimestamps) {
    this(null, username, useTimestamps);
}

public MusicAlbum(String name, String username, boolean useTimestamps) {
    this.name = name;
    this.username = username;
    this.useTimestamps = useTimestamps;
}
```

The `username` column and the `name` column provide a composite primary key for the `album` table. The `username` column identifies the owner of an album and allows the `MusicAlbumsResource.getBusinessObjects` method to return only the albums for a particular user, so that multiple users can share the `album` table.

Now the class implements its getter and setter methods for the `name`, `artist`, `rating`, `lastModified`, and `datePublished` properties:

```
public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getArtist() {
    return artist;
}

public void setArtist(String artist) {
    this.artist = artist;
}

public int getRating() {
    return rating;
}

public void setRating(int rating) {
    this.rating = rating;
}
```

```
public long getLastModified() {
    return lastModified;
}

public void setLastModified(long millis) {
    lastModified = millis;
}

/**
 * Returns the date published as a string in the format
 * 'YYYYMMDD'.
 */
public String getDatePublished() {
     stringBuilder.setLength(0);
     stringBuilder.append(datePublished.get(Calendar.YEAR));
     int month = datePublished.get(Calendar.MONTH) + 1;
     if (month < 10) {
         stringBuilder.append('0');
     }
     stringBuilder.append(month);
     int day = datePublished.get(Calendar.DAY_OF_MONTH);
     if (day < 10) {
         stringBuilder.append('0');
     }
     stringBuilder.append(day);
     return stringBuilder.toString();
}

/**
 * Sets the date published in the format 'YYYYMMDD'.
 */
public void setDatePublished(String date) {
    datePublished = Calendar.getInstance();
    datePublished.set(Calendar.YEAR,
            Integer.parseInt(date.substring(0, 4)));
    datePublished.set(Calendar.MONTH,
            Integer.parseInt(date.substring(4, 6)) - 1);
    datePublished.set(Calendar.DAY_OF_MONTH,
            Integer.parseInt(date.substring(6, 8)));
}
```

In a client application, a business object takes the form of a file whose name has an extension unique to that business object. For a JAX-RS Enterprise Connector, the filename extension is returned by the `initialize` operation in one of the resource classes rather than being defined in the business object class. (See "The `MyBusinessObjectsResource` Class" on page 27 for details.)

The class uses Java Serialization to implement the `serialize` and `deserialize` methods as follows. The byte array format used in the `serialize` and `deserialize` methods, like the data types of the fields, is part of the contract between the client application and the Enterprise Connector.

```
public byte[] serialize() throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dOut = new DataOutputStream(out);

    dOut.writeUTF(getName());
    dOut.writeUTF(getArtist() != null ? getArtist() : DEFAULT_VALUE);
    dOut.writeUTF(getDatePublished() != null ? getDatePublished() : DEFAULT_VALUE);
    dOut.writeUTF(Integer.toString(getRating()));
    dOut.flush();
    return out.toByteArray();
}

public void deserialize(byte[] array) throws IOException {
    ByteArrayInputStream in = new ByteArrayInputStream(array);
    DataInputStream dIn = new DataInputStream(in);

    setName(dIn.readUTF());
    artist = dIn.readUTF();
    if (artist.equals(DEFAULT_VALUE)) {
        artist = null;
    }
    String date = dIn.readUTF();
    if (date.equals(DEFAULT_VALUE)) {
        datePublished = null;
    }
    else {
        setDatePublished(date);
    }
    rating = Integer.parseInt(dIn.readUTF());
}
```

The class implements `getInsertString`, `getUpdateString`, and `getDeleteString` helper methods. These methods return SQL statement strings to perform database operations on the business objects:

```
/**
 * Returns an SQL insert statement to add this instance
 * to the back-end database.
 */
public String getInsertString() {
    stringBuilder.setLength(0);
    stringBuilder.append(useTimestamps ?
        "INSERT INTO album" :
```

```
            "INSERT INTO album (name, artist, date_published, rating, username)")
            .append(" VALUES ('")
            .append(getName()).append("', '")
            .append(artist).append("', DATE '")
            .append(datePublished.get(Calendar.YEAR)).append("-")
            .append(datePublished.get(Calendar.MONTH) + 1).append("-")
            .append(datePublished.get(Calendar.DAY_OF_MONTH)).append("', ")
            .append(Integer.toString(rating)).append(", '")
            .append(username)
            .append("'");

        if (useTimestamps) {
            Timestamp timestamp = new Timestamp(lastModified);
            stringBuilder.append(", TIMESTAMP '")
                    .append(timestamp.toString())
                    .append("'");
        }
        stringBuilder.append(')');

        return stringBuilder.toString();
    }

    /**
     * Returns an SQL update statement to modify this instance
     * in the back-end database.
     */
    public String getUpdateString() {
        stringBuilder.setLength(0);
        stringBuilder.append("UPDATE album SET artist='")
             .append(artist).append("', date_published=DATE '")
             .append(datePublished.get(Calendar.YEAR)).append("-")
             .append(datePublished.get(Calendar.MONTH) + 1).append("-")
             .append(datePublished.get(Calendar.DAY_OF_MONTH)).append("', rating = ")
             .append(Integer.toString(rating));

        if (useTimestamps) {
            Timestamp timestamp = new Timestamp(lastModified);
            stringBuilder.append(", last_modified = TIMESTAMP '")
                .append(timestamp.toString()).append("'");
        }

        stringBuilder.append(" WHERE name = '").append(getName())
             .append("' AND username = '")
             .append(username)
             .append("'");
        return stringBuilder.toString();
    }
```

```
/**
 * Returns an SQL delete statement to remove this instance
 * from the back-end database.
 */
public String getDeleteString() {
    stringBuilder.setLength(0);
    stringBuilder.append("DELETE FROM album WHERE name = '")
                 .append(getName())
                 .append("' AND username = '" + username + "'");
    return stringBuilder.toString();
}
```

Using NetBeans IDE to Create a JAX-RS Enterprise Connector

The simplest way to create a JAX-RS Enterprise Connector is to use NetBeans IDE 6.5 with the Maven plugin and the Maven archetypes for Sun GlassFish Mobility Platform.

▼ To Install the Maven Plugin

Before You Begin If you do not already have NetBeans IDE 6.5, go to `http://www.netbeans.org/` to download and install it.

On the NetBeans IDE download page, select Java as the bundle to download. This bundle includes Java SE, Web, Java EE, and Java ME. When you install NetBeans IDE, click Customize to install only some of the components. Deselect all of the listed runtimes, since you are using the Sun GlassFish Mobility Platform version of Sun GlassFish Enterprise Server.

1. **Start NetBeans IDE.**
2. **From the Tools menu, select Plugins.**
3. **In the Plugins dialog, click the Available Plugins tab, then type Maven in the Search field.**
4. **Select the checkbox for the Maven plugin, then click the Install button.**
5. **Follow the instructions in the NetBeans IDE Installer wizard.**
6. **After the wizard returns, click Close in the Plugins dialog.**
7. **Choose Windows→Other→Maven Repository Browser.**

8 **In the Maven Repository Browser tree, right-click the Java.net Repository node and select Update Index.**

▼ To Create a Maven Project for a JAX-RS Enterprise Connector

1 **In NetBeans IDE, click the Projects tab.**

2 **Choose File→New Project.**

3 **In the New Project dialog, select Maven from the Categories tree.**

4 **Select Maven Project from the Projects list, then click Next.**

5 **On the Maven Archetype page, expand Archetypes from the remote Maven Repositories node.**

6 **From the list, select MEP Connector Archetype (JAX-RS), then click Next.**

7 **On the Name and Location page, modify any default values you wish to, then click Finish.**

The new project appears in the Projects tab. The default project name is mavenproject1 (war).

8 **Under mavenproject1, expand the Source Packages node.**

9 **Expand the com.mycompany.mavenproject1 node.**

Two class files appear, named `MyBusinessObjectsResource.java` and `MyBusinessObjectResource.java`. These classes correspond to the container-item pattern familiar to JAX-RS programmers.

Next Steps To create the Enterprise Connector, you add code to these files and define your business object.

▼ To View and Build the MusicDB JAX-RS Connector Project

1 **In NetBeans IDE, select File → Open Project.**

2 **In the Open Project window, navigate to `sgmp-client-1_1_01-fcs-b02/samples/secure-musicdb/src/connector/jaxrs/musicdb-ws-3.1.39` and click Open Project.**

The `musicdb-ws` project appears in the Projects tab. The source files are under the Source Packages node.

If the project has a red exclamation point next to it, right click the project and choose Show and Resolve Problems. In the window that appears, click the Download Libraries button. After a short delay, thewindow disappears and the red exclamation point no longer appears next to the project.

3 **Right-click the project and choose Build to build the project.**

The first time you build a project using the Maven plugin, the build takes a long time while Maven downloads a large number of libraries.

Defining Operations on Business Objects

The resource classes in the `mavenproject1` project, `MyBusinessObjectsResource.java` and `MyBusinessObjectResource.java`, use the container-item design pattern typical of JAX-RS applications. The methods in these classes are invoked by the Gateway.

The methods in the `MyBusinessObjectsResource` class perform operations on business objects as follows:

- The `MyBusinessObjectsResource.getBusinessObjects` method implements the GET operation, retrieving a list of business objects.
- The `MyBusinessObjectsResource.getBusinessObject` method retrieves a single `MyBusinessObjectResource` object using a `@path` annotation. This method is invoked as part of the design pattern and is already implemented in the Maven archetype.
- The `MyBusinessObjectsResource.lifeCycle` method implements the POST operation, initiating and terminating a session with the JAX-RS Enterprise Connector bridge. It also returns the file extension used by all business objects returned by this Enterprise Connector.

The methods in the `MyBusinessObjectResource` class perform operations on business objects as follows:

- The `MyBusinessObjectResource.getBusinessObject` method is not currently invoked. The Maven archetype allows you to implement it, and the MusicDB JAX-RS Enterprise Connector implements it, but you are not required to implement it. The method implements the GET operation, retrieving a business object in serialized form.
- The `MyBusinessObjectResource.putBusinessObject` method implements the PUT operation, creating or updating the binary representation of a business object.
- The `MyBusinessObjectResource.deleteBusinessObject` method implements the DELETE operation, deleting the business object.

The following sections describe these methods more fully and provide examples of their implementation.

- "The `MyBusinessObjectsResource` Class" on page 27
- "The `MyBusinessObjectResource` Class" on page 31

The `MyBusinessObjectsResource` Class

The `MyBusinessObjectsResource` class exposes the following URLs, as the comments for the class indicate:

- `http://`*server*`:`*port*`/`*context-root*`/resources/businessObjects`
- `http://`*server*`:`*port*`/`*context-root*`/resources/businessObjects/{id}`, where `id` is the name of the business object.

The `@Path` annotation at the beginning of the class establishes the path to these URLs:

```
@Path("businessObjects")
public class MyBusinessObjectsResource {
```

The class begins with a definition of the file extension to be returned by the `lifeCycle` method, followed by a context declaration and class constructor.

```
    // UPDATE CODE: Set extension for these objects
    public static final String EXTENSION = ".obj";

    @Context
    private UriInfo context;

    public MyBusinessObjectsResource() {
    }
```

The `getBusinessObjects` method is mapped to the HTTP GET operation on this resource. The method template looks like this:

```
    @GET
    @Produces("text/xml")
    public BusinessObjects getBusinessObjects(
            @QueryParam("username") @DefaultValue("username") String user,
            @QueryParam("password") @DefaultValue("password") String password,
            @QueryParam("sessionId") @DefaultValue("") String sessionId)
    {
        try {
            BusinessObjects result = new BusinessObjects();
            List<BusinessObject> resultList = result.getBusinessObject();
```

```
            // INSERT CODE: populate list of business objects

            return result;
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

The `getBusinessObjects` method returns an instance of a JAXB object, `BusinessObjects`, serialized as an XML document. Each object is identified by a name and a binary representation that is a contract between the Enterprise Connector and the client. The `getBusinessObject` method called on the `result` object is not one of the methods exposed in the resource classes. Instead, it is a JAXB accessor method defined in the package that is imported at the beginning of the resource class file:

```
import com.sun.mep.connector.jaxrs.getbusinessobjects.*;
```

Some operations go beyond the simple CRUD operations. The `lifeCycle` method corresponds to the HTTP POST operation. It has the following signature:

```
    @POST
    @Produces("text/plain")
    public String lifeCycle(
            @QueryParam("username") @DefaultValue("username") String user,
            @QueryParam("password") @DefaultValue("password") String password,
            @QueryParam("sessionId") @DefaultValue("") String sessionId,
            @QueryParam("operation") String operation)
```

You can use this method to control the lifecycle of a synchronization session, identified by the `sessionId` parameter. Specifically, you can use the value of the `operation` argument to initialize a session by connecting to a database or EIS system, or to terminate the session:

```
        if (operation.equals("initialize")) {
            // INSERT CODE: session initialization
        }
        else if (operation.equals("terminate")) {
            // INSERT CODE: session termination
        }
        else {
            throw new RuntimeException("Lifecycle operation " + operation +
                    " not understood");
        }
        return EXTENSION;
```

Example: The `MusicAlbumsResource` Container Class

In the MusicDB example, the `MusicAlbumsResource` class is the container class. After importing needed packages, the resource class begins with a `@Path` annotation that establishes the URLs:

```
@Path("albums")
public class MusicAlbumsResource {
```

The HTTP operations will use the URLs `http://`*server*`:`*port*`/`*context-root*`/resources/albums` and `http://`*server*`:`*port*`/`*context-root*`/resources/albums/{id}`. For example, if you specify `musicdb-ws` as the context root and are running the Enterprise Server on your own system, the URL for retrieving the business objects would be `http://localhost:8080/musicdb-ws/resources/albums`.

The next few lines define the file extension to be used for these business objects, a JAX-RS context, and a class constructor. They also define a user name prefix specific to this Enterprise Connector, to be used by the `lifeCycle` method:

```
    private static final String EXTENSION = ".alb";
    private static final String DB_USER_NAME = "musicdbuser";

    @Context
    private UriInfo context;

    public MusicAlbumsResource() {
    }
```

Next, the `lifeCycle` method provides code to establish or terminate a connection to the database, using the `ConnectionPool` defined in `src/main/java/com/sun/mep/ws/musicdb/util/ConnectionPool.java`. Both the `createConnection` and `destroyConnection` methods take a session ID argument.

```
    @POST
    @Produces("text/plain")
    public String lifeCycle(
            @QueryParam("username") @DefaultValue("username") String user,
            @QueryParam("password") @DefaultValue("password") String password,
            @QueryParam("sessionId") @DefaultValue("") String sessionId,
            @QueryParam("operation") String operation)
    {
        if (operation.equals("initialize")) {
            if (!user.startsWith(DB_USER_NAME)) {
                throw new RuntimeException("A MusicDB user name must start "
                        + "with the '"+ DB_USER_NAME + "' prefix");
            }
            ConnectionPool.getInstance().createConnection(sessionId);
        }
```

```
        else if (operation.equals("terminate")) {
            ConnectionPool.getInstance().destroyConnection(sessionId);
        }
        else {
            throw new RuntimeException("Lifecycle operation " + operation +
                    " not understood");
        }
        return EXTENSION;
    }
```

For the MusicDB application, a connection can be established only if the user name begins with `musicdbuser`. At the end, the method returns the string `.alb`, the file extension for music albums.

The `getBusinessObjects` method begins by using another connection to create a JDBC statement:

```
    @GET
    @Produces("text/xml")
    public BusinessObjects getBusinessObjects(
            @QueryParam("username") @DefaultValue("username") String user,
            @QueryParam("password") @DefaultValue("password") String password,
            @QueryParam("sessionId") @DefaultValue("") String sessionId,
            @QueryParam("useTimestamps") @DefaultValue("false") boolean useTimestamps)
    {
        Statement stmt = null;

        try {
            // Create SQL statement from connection
            ConnectionPool pool = ConnectionPool.getInstance();
            stmt = pool.getConnection(sessionId).createStatement();
```

The `useTimestamps` argument of the method, which is specific to the MusicDB Enterprise Connector, provides the value of the `useTimestamps` flag. An administrator can set the value of the `useTimestamps` property when defining the Enterprise Connector in the Sun GlassFish Mobility Platform Administration Console. If the property is not set, the argument value is set to false.

It then executes the template code that instantiates the list of JAXB objects:

```
            // Get list of business objects to return
            BusinessObjects result = new BusinessObjects();
            List<BusinessObject> resultList = result.getBusinessObject();
```

Next, the method performs an SQL query to retrieve from the back-end database all the business objects that belong to the user who is logged in to the Enterprise Connector:

```
            ResultSet rs = stmt.executeQuery(
                    "SELECT * FROM album WHERE username = '" + user + "'");
```

The method iterates through the query result set, instantiating each business object (in this case, a `MusicAlbum`) and populating it with its retrieved properties (including the `lastModified` property if the `useTimestamps` flag is set), then serializing it to a JAXB object and adding it to the list of objects. Finally, the method closes the result set and returns the list of JAXB objects.

```
            while (rs.next()) {
                // Use temporary object for serialization purposes
                MusicAlbum album = new MusicAlbum(user, false);
                album.setName(rs.getString(1));
                album.setArtist(rs.getString(2));
                album.setDatePublished(rs.getString(3).replace("-", ""));
                album.setRating(rs.getInt(4));
                if (useTimestamps) {
                    album.setLastModified(rs.getTimestamp(6).getTime());
                }

                // Map MusicAlbum to JAXB bean
                BusinessObject bo = new BusinessObject();
                bo.setName(album.getName());
                bo.setValue(album.serialize());
                if (useTimestamps) {
                    bo.setLastModified(album.getLastModified());
                }

                // Add BusinessObject to result list
                resultList.add(bo);
            }
            rs.close();

            return result;
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
        finally {
            if (stmt != null) {
                try { stmt.close(); } catch (Exception e) { /* ignore !*/ }
            }
        }
    }
```

The `MyBusinessObjectResource` Class

The `MyBusinessObjectResource` class implements the CUD operations create, update, and delete, using the HTTP operations PUT and DELETE.

The `putBusinessObject` method either creates or updates a business object, returning the serialized object. The method looks like this:

```
@PUT
@Consumes("application/octet-stream")
public void putBusinessObject(
        @QueryParam("username") @DefaultValue("username") String user,
        @QueryParam("password") @DefaultValue("password") String password,
        @QueryParam("sessionId") @DefaultValue("") String sessionId,
        @QueryParam("lastModified") @DefaultValue("0") long lastModified,
        @PathParam("id") String id, byte[] object)
{
    // INSERT CODE: create/update object from client
}
```

The `deleteBusinessObject` method removes a business object from the back-end system, given its identifier and, optionally, the object contents. The method signature looks like this:

```
@DELETE
public void deleteBusinessObject(
        @QueryParam("username") @DefaultValue("username") String user,
        @QueryParam("password") @DefaultValue("password") String password,
        @QueryParam("sessionId") @DefaultValue("") String sessionId,
        @QueryParam("lastModified") @DefaultValue("0") long lastModified,
        @PathParam("id") String id, byte[] object)
```

The `mergeBusinessObjects` method, like the `MyBusinessObjectsResource.lifeCycle` method, goes beyond the simple CRUD operations to implement the HTTP POST operation:

```
@POST
@Produces("application/octet-stream")
public byte[] mergeBusinessObjects(
        @QueryParam("username") @DefaultValue("username") String user,
        @QueryParam("password") @DefaultValue("password") String password,
        @QueryParam("sessionId") @DefaultValue("") String sessionId,
        @PathParam("id") String id, BusinessObjects objects)
```

Two business objects are passed in the argument `objects`: the server object followed by the client object.

The Gateway can call this method when there is an update conflict. If an object is updated on both the client and the server, the Gateway asks the Enterprise Connector to provide a merged object containing updated information for both objects.

Because the `getBusinessObject` method is not currently used, it is not described here.

Example: The `MusicAlbumResource` Class

In the MusicDB example, the `MusicAlbumResource` class is the item class. The `MusicAlbumResource` class implements the CUD methods specified by the `MyBusinessObjectResource` template.

The `putBusinessObject` method begins by adding an additional boolean parameter, `useTimestamps`, to the method signature:

```
@PUT
@Consumes("application/octet-stream")
public void putBusinessObject(
        @QueryParam("username") @DefaultValue("username") String user,
        @QueryParam("password") @DefaultValue("password") String password,
        @QueryParam("sessionId") @DefaultValue("") String sessionId,
        @QueryParam("lastModified") @DefaultValue("0") long lastModified,
        @QueryParam("useTimestamps") @DefaultValue("false") boolean useTimestamps,
        @PathParam("id") String id, byte[] object)
```

The `useTimestamps` parameter allows the method to instantiate a `MusicAlbum` using its three-argument constructor. The method code begins by declaring two JDBC statements and creating a connection pool instance. It then instantiates the `MusicAlbum` object and deserializes its byte array argument into this object:

```
Statement stmt1 = null, stmt2 = null;
try {
        // Get connection pool
        ConnectionPool pool = ConnectionPool.getInstance();

        // Map to connector object to generate SQL statement
        MusicAlbum album = new MusicAlbum(id, user, useTimestamps);
        album.deserialize(object);
```

Next, the method determines whether the album already exists by executing the select statement for the `MusicAlbum` object.

```
        // Check if album already exists
        boolean isInsert = true;
        stmt1 = pool.getConnection(sessionId).createStatement();
        ResultSet rs = stmt1.executeQuery(album.getSelectString());
        if (rs.next()) {
            isInsert = false;
        }
        rs.close();
```

If the select statement returns a non-empty result set, the boolean value `isInsert` is set to false, indicating that the method will perform an update. Otherwise, the method will perform an

insert to create the object in the database. The following code executes either the insert statement or the update statement for the `MusicAlbum` object, then closes the two JDBC statements.

```
            // Run INSERT or UPDATE  against database
            stmt2 = pool.getConnection(sessionId).createStatement();
            stmt2.executeUpdate(isInsert ? album.getInsertString()
                    : album.getUpdateString());
        }
        catch (Exception ex) {
            throw new RuntimeException(ex);
        }
        finally {
            if (stmt1 != null) {
                try { stmt1.close(); } catch (Exception e) { /* ignore !*/ }
            }
            if (stmt2 != null) {
                try { stmt2.close(); } catch (Exception e) { /* ignore !*/ }
            }
        }
    }
```

The `deleteBusinessObject` method for `MusicAlbum`, like `putBusinessObject`, adds a `useTimestamps` parameter to its signature:

```
    @DELETE
    public void deleteBusinessObject(
            @QueryParam("username") @DefaultValue("username") String user,
            @QueryParam("password") @DefaultValue("password") String password,
            @QueryParam("sessionId") @DefaultValue("") String sessionId,
            @QueryParam("lastModified") @DefaultValue("0") long lastModified,
            @QueryParam("useTimestamps") @DefaultValue("false") boolean useTimestamps,
            @PathParam("id") String id, byte[] object)
```

The method declares a JDBC statement, then instantiates a `MusicAlbum` object using its three-argument constructor and deserializes its byte array argument into this object:

```
    {
        Statement stmt = null;
        try {
            // Map to connector object to generate SQL statement
            MusicAlbum album = new MusicAlbum(id, user, useTimestamps);
            album.deserialize(object);
```

The code creates a connection pool instance and a JDBC statement, then executes the delete statement for the `MusicAlbum` object to remove it from the database:

```
            ConnectionPool pool = ConnectionPool.getInstance();
            stmt = pool.getConnection(sessionId).createStatement();
            stmt.executeUpdate(album.getDeleteString());
        }
        catch (Exception ex) {
            throw new RuntimeException(ex);
        }
        finally {
            if (stmt != null) {
               try { stmt.close(); } catch (Exception e) { /* ignore !*/ }
            }
        }
    }
```

The `MusicAlbum` Enterprise Connector does not implement the `mergeBusinessObjects` method. This means that the object on the server always wins if there is a conflict.

The `getBusinessObject` method is implemented in the `MusicAlbumResource` class, but it is never invoked.

Building, Deploying, and Configuring a JAX-RS Enterprise Connector

Once you have used NetBeans IDE to create a JAX-RS Enterprise Connector, build it by right-clicking the project and choosing Build.

To deploy the Enterprise Connector, use the Sun GlassFish Enterprise Server Administration Console. You must also configure a connector connection pool and a connector resource for the Enterprise Connector.

By default, the MusicDB JAX-RS Enterprise Connector is deployed in an installed version of Sun GlassFish Mobility Platform. However, it is not configured. If you would like to use this connector, you must undeploy the deployed connector and deploy the version you opened and built in "To View and Build the MusicDB JAX-RS Connector Project" on page 25.

Deploying and configuring an Enterprise Connector involves the following tasks:

▼ To Start the Sun GlassFish Enterprise Server and Database

1. **In NetBeans IDE, click the Services tab.**
2. **Right-click the Servers node and choose Add Server.**
   The Add Server Instance dialog opens.
3. **On the Choose Server page, select GlassFish V2 and click Next.**
4. **On the Platform Folder Location page, click Browse and navigate to the location of the Enterprise Server, then click Choose.**
   The Enterprise Server location appears in the Platform Location field.
5. **Click Next.**
   The Domain Admin Login Info appears.
6. **Verify that the Admin Username is correct, and type the administrator password in the Admin Password field.**
   The default password is `adminpass`.
7. **Click Finish.**
   GlassFish V2 appears under the Servers node.
8. **If the Enterprise Server is not already running, right-click GlassFish V2 and choose Start.**
9. **Expand the Databases node.**
10. **If your database (either MySQL or Oracle) is not running, right-click the database and choose Start.**
    If the database does not start, you may have to edit the database properties to specify your password.

▼ To Deploy an Enterprise Connector

1. **Log in to the Enterprise Server Administration Console.**
   a. **In a browser, type the Administration Console URL (usually `http://localhost:4848/`).**
   b. **Type the administrator username and password (usually `admin` and `adminpass`).**

c. **Click the Login button.**

2 **In the navigation tree, expand the Applications node.**

3 **Under the Applications node, select the Web Applications node.**

The Web Applications page opens.

4 **(Optional) If you plan to deploy the MusicDB JAX-RS Enterprise Connector, select** `musicdb-ws` **and click the Undeploy button.**

5 **Click the Deploy button.**

The Deploy Enterprise Applications/Modules page opens.

6 **On the Deploy Enterprise Applications/Modules page, verify that Packaged File to be Uploaded to the Server is selected, then click Browse.**

7 **In the file chooser, navigate to the location of the WAR file and click Open.**

The WAR file is normally in the directory `NetBeansProjects/`*project-name*`/target`. On a Windows system, the `NetBeansProjects` directory is under My Documents.

The new `musicdb-ws` project is in the directory `sgmp-client-1_1_01-fcs-b02/samples/secure-musicdb/src/connector/jaxrs/musicdb-ws-3.1.39/target`.

8 **If necessary, change the Application Name and Context Root by removing the version part of the WAR file name.**

9 **Click OK.**

The deployed Enterprise Connector appears in the Web Applications table.

Next Steps Next, use the Enterprise Server Administration Console to configure the Enterprise Connector.

▼ To Create a Connector Connection Pool for the Enterprise Connector

1 **In the navigation tree, expand the Resources node.**

2 **Under the Resources node, expand the Connectors node.**

3 **Select Connector Connection Pools.**

The Connector Connection Pools page opens.

4. **Click the New button.**
   The New Connector Connection Pool (Step 1 of 2) page opens.
5. **In the Name field, type** `mep/`*connector-name*, **where** *connector-name* **is the name of your Enterprise Connector.**
   For MusicDB, the name should be `mep/musicdb-ws`.
6. **From the Resource Adapter drop-down list, select** `ds-jcr-connector-jaxrs`, **the JAX-RS connector bridge.**
   The connection definition, `javax.resource.spi.ManagedConnectionFactory`, is filled in automatically.
7. **Click Next.**
   The New Connector Connection Pool (Step 2 of 2) page opens.
8. **Click Finish to accept the default settings.**
   The new connector connection pool appears in the Connector Connection Pools table.

Next Steps Next, you must create a connector resource for the Enterprise Connector.

▼ To Create a Connector Resource for the Enterprise Connector

1. **Under the Connectors Node, select the Connector Resources node.**
   The Connector Resources page opens.
2. **Click New.**
   The New Connector Resource page opens.
3. **In the JNDI Name field, type** `mep/`*connector-name*, **where** *connector-name* **is the name of your Enterprise Connector.**
   For MusicDB, the name should be `mep/musicdb-ws`.
4. **From the Pool Name drop-down list, select** `mep/`*connector-name*, **the connection pool you created.**
   For MusicDB, the name should be `mep/musicdb-ws`.
5. **Click OK.**
   The new connector resource appears in the Connector Resources table.

▼ To Verify that an Enterprise Connector Is Deployed

1 **In your browser, open the URL for the business objects, using this syntax:**

`http://`*server*`:`*port*`/`*context-root*`/resources/`*businessObjects*

For example, for the MusicDB Enterprise Connector, you would specify the following for a locally running Enterprise Server:

```
http://localhost:8080/musicdb-ws/resources/albums
```

Specifying this URL is equivalent to invoking the `getBusinessObjects` method.

2 **Verify the content of the page that opens.**

If the Enterprise Connector is running, an XML document opens in your browser, containing a `businessObjects` element that encloses zero or more `businessObject` elements. Each `businessObject` element contains a name field and a binary representation.

▼ To Configure the Enterprise Connector on Sun GlassFish Mobility Platform

Add and configure the JAX-RS Enterprise Connector as a local Enterprise Connector using the Sun GlassFish Mobility Platform Administration Console.

You can also configure the Enterprise Connector as a remote Enterprise Connector if you deploy it on a remote system. For instructions, see "To Create and Activate a New Enterprise Connector" in *Sun GlassFish Mobility Platform 1.1 Administration Guide* and "To Configure a Remote Enterprise Connector" in *Sun GlassFish Mobility Platform 1.1 Administration Guide*.

1 **Log in to the Sun GlassFish Mobility Platform Administration Console.**

   a. **In a browser, type the Administration Console URL (usually `http://localhost:8080/sync/admin`).**

   b. **Type the username and password (the default values are `admin` and `syncpass`).**

   c. **Click the Login button.**

2 **Click the Connectors tab.**

3 **In the Name field of the Local Enterprise Connectors panel, type the name of the Enterprise Connector (for example, MusicDbRS).**

4. **Click Add.**

   The new Enterprise Connector appears in the list of repositories. The Active checkbox is selected by default, indicating that the Enterprise Connector has been activated.

   A new sub-tab for the Enterprise Connector appears. You can now configure the new Enterprise Connector.

5. **Click the sub-tab for the connector.**

6. **In the Local Enterprise Connector Settings panel, type the JNDI name of the connector resource you created in "To Create a Connector Resource for the Enterprise Connector" on page 38 (for example, `mep/musicdb-ws`) and click Save.**

7. **In the Local Enterprise Connector Properties panel, specify a value for the `uri` property.**

   For the MusicDB connector, this value is `http://localhost:8080/musicdb-ws/resources/albums`.

8. **(Optional) If you are configuring the MusicDB Enterprise Connector, in the same panel, specify a value of `true` for the `useTimestamps` property.**

   If you do not set the property, or if you set it to `false`, the MusicDB connector will use digests instead of timestamps to determine whether a business object has been updated since the last synchronization.

9. **Click Save in the Local Enterprise Connector Properties panel.**

Next Steps You can now perform synchronizations using the Enterprise Connector.

◆ ◆ ◆ CHAPTER 3

Creating an Enterprise Connector Using the Enterprise Connector Business Objects (ECBO) API

A Sun GlassFish Mobility Platform Enterprise Connector that uses the Enterprise Connector Business Object (ECBO) API typically extends five classes in the API and also includes an XML file that defines parameters used by the underlying implementation.

For details on the ECBO API classes and methods, see Chapter 5, "Classes and Methods in the Enterprise Connector Business Object API Package." The API documentation is also included in the Sun GlassFish Mobility Platform client bundle. In the directory where you unzipped the client bundle (see the *Sun GlassFish Mobility Platform 1.1 Installation Guide* for details), it is in the subdirectory `sgmp-client-1_1_01-fcs-b02/doc/ecbo/api`.

This chapter uses the MusicDB ECBO sample Enterprise Connector provided with Sun GlassFish Mobility Platform to demonstrate how to use the ECBO API. This Enterprise Connector acts as the intermediary between a client application on a mobile device and a back-end system. For this simple application, the back-end system is not a full-fledged EIS/EAI system but an ordinary database that is accessed using the Java Database Connectivity (JDBC) API.

By default, the MusicDB ECBO sample Enterprise Connector is deployed and configured in an installed version of Sun GlassFish Mobility Platform. You can view it in the Sun GlassFish Enterprise Server and Sun GlassFish Mobility Platform Administration Consoles as an example of how to deploy and configure an Enterprise Connector.

The source code for the MusicDB ECBO Enterprise Connector is included in the Sun GlassFish Mobility Platform client bundle. In the directory where you unzipped the client bundle, it is in a ZIP file in the subdirectory `sgmp-client-1_1_01-fcs-b02/samples/secure-musicdb/src/connector/ecbo/`. The ZIP file unzips as a Maven project.

This chapter covers the following topics:

Classes in the Enterprise Connector Business Object API

The Enterprise Connector Business Object (ECBO) API contains the following classes:

- `com.sun.mep.connector.api.BusinessObject` is the base class for all business objects. Business objects are the entities synchronized with client applications. The fields of the Enterprise Connector business object, and the serialization of those fields, must match those of the business object for the corresponding client application.
- `com.sun.mep.connector.api.BusinessObjectProvider` is a provider class for instances of `BusinessObject`.
- `com.sun.mep.connector.api.Command` is the base class for all business object commands.
- `com.sun.mep.connector.api.InsertCommand` is the base class for the command that inserts a business object into the database or EIS/EAI system.
- `com.sun.mep.connector.api.UpdateCommand` is the base class for the command that updates a business object in the database or EIS/EAI system.
- `com.sun.mep.connector.api.DeleteCommand` is the base class for the command that deletes a business object from the database or EIS/EAI system.
- `com.sun.mep.connector.api.SessionContext` is a helper class used by `BusinessObjectProvider`. It stores contextual information about the session in which a `BusinessObjectProvider` is instantiated.
- `com.sun.mep.connector.api.TransactionManager` is a helper class used by `BusinessObjectProvider`. It supports methods for starting, stopping, and aborting database transactions.

The MusicDB example implements its own versions of all of these classes except for `SessionContext`. It uses the default implementation of `SessionContext`.

See Chapter 5, "Classes and Methods in the Enterprise Connector Business Object API Package," for summaries of the classes and methods in the ECBO API packages.

To synchronize data with an EIS/EAI system such as Siebel or SAP, your `BusinessObjectProvider` implementation and the three command implementations will need to call methods that access the Sun JCA Adapter for that system. These methods are provided by the Sun Java Composite Application Platform Suite (Java CAPS). See Chapter 4, "Accessing a Sun JCA Adapter for an EIS/EAI System," for details.

Extending the BusinessObject Class

The BusinessObject class holds the data you need to synchronize. In addition to the required properties name and extension, you specify properties and define getter and setter methods for this data.

For details on this class, see "The BusinessObject Class" on page 75.

The name property is the most important BusinessObject property. This property defines the identity of the business object. The name you specify must be unique within your database or EIS/EAI system.

For the MusicDB example, the class that extends BusinessObject is MusicAlbum. The source file MusicAlbum.java begins by importing Java SE packages and the required ECBO API class that it extends:

```
package com.sun.mep.connector.jdbc.album;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import java.sql.Connection;
import java.util.Calendar;
import java.sql.Timestamp;

import com.sun.mep.connector.api.BusinessObject;
```

The class code itself begins by declaring a string value to be used by the serialize method:

```
public class MusicAlbum extends BusinessObject<MusicAlbumProvider> {

    private static final String DEFAULT_VALUE = "$$default$$";
```

This example then declares its data properties (there are only four in addition to the name property):

```
    /**
     * Album's artist.
     */
    String artist;

    /**
     * Date when the album was published.
     */
    Calendar datePublished;
```

```
/**
 * Album's rating from 1 to 5.
 */
int rating;

/**
 * Last modified timestamp.
 */
long lastModified;
```

The `lastModified` property is unique to the Enterprise Connector version of the business object; it is not used in the mobile client application. This property allows the Gateway to use timestamps to query the back-end system to determine whether a business object has been updated since the last synchronization. If the Enterprise Connector does not use timestamps, the Gateway computes a digest over the serialized business object, using a hash function, and compares it to the previous digest. Using timestamps is more efficient than using digests.

The code then declares a `StringBuilder` and defines the one-argument constructor, which takes the `MusicAlbumProvider` as its argument:

```
/**
 * String builder used to return SQL commands.
 */
StringBuilder stringBuilder = new StringBuilder();

public MusicAlbum(MusicAlbumProvider provider) {
    super(provider);
}
```

Now the class implements its getter and setter methods for the `artist`, `rating`, `datePublished`, and `lastModified` properties:

```
public String getArtist() {
    return artist;
}

public void setArtist(String artist) {
    this.artist = artist;
}

public int getRating() {
    return rating;
}

public void setRating(int rating) {
    this.rating = rating;
}
```

```
/**
 * Returns the date published as a string in the format
 * 'YYYYMMDD'.
 */
public String getDatePublished() {
     stringBuilder.setLength(0);
     stringBuilder.append(datePublished.get(Calendar.YEAR));
     int month = datePublished.get(Calendar.MONTH) + 1;
     if (month < 10) {
         stringBuilder.append('0');
     }
     stringBuilder.append(month);
     int day = datePublished.get(Calendar.DAY_OF_MONTH);
     if (day < 10) {
         stringBuilder.append('0');
     }
     stringBuilder.append(day);
     return stringBuilder.toString();
}

/**
 * Set the date published in the format 'YYYYMMDD'.
 */
public void setDatePublished(String date) {
    datePublished = Calendar.getInstance();
    datePublished.set(Calendar.YEAR,
            Integer.parseInt(date.substring(0, 4)));
    datePublished.set(Calendar.MONTH,
            Integer.parseInt(date.substring(4, 6)) - 1);
    datePublished.set(Calendar.DAY_OF_MONTH,
            Integer.parseInt(date.substring(6, 8)));
}
```

The code next implements the `getLastModified` and `setLastModified` methods:

```
/**
 * Returns last modified timestamp.
 */
@Override
public long getLastModified() {
    if (getBusinessObjectProvider().useTimestamps()) {
        return lastModified;
    }
    else {
        throw new UnsupportedOperationException(getClass().getName() +
                " does not support lastModified timestamp");
    }
```

```
}

/**
 * Sets last modified timestamp.
 */
@Override
public void setLastModified(long millis) {
    if (getBusinessObjectProvider().useTimestamps()) {
        lastModified = millis;
    }
    else {
        throw new UnsupportedOperationException(getClass().getName() +
                " does not support lastModified timestamp");
    }
}
```

If the Enterprise Connector does not use timestamps, the methods throw exceptions that cause the Gateway to use digests.

The class implements the `getExtension` method by specifying `.alb` as the file extension for `MusicAlbum` objects. This extension must match the extension used by the client.

```
@Override
public String getExtension() {
    return ".alb";
}
```

The class does not implement its own versions of the `getName` and `setName` methods; instead, it uses the versions defined in the `BusinessObject` class.

The class uses Java Serialization to implement the `serialize` and `deserialize` methods as follows. Note the calls to the `BusinessObject` versions of `getName` and `setName` in addition to the getter and setter methods defined by `MusicAlbum`. The format used in the `serialize` and `deserialize` methods is part of the contract between the client and the Enterprise Connector.

```
public byte[] serialize() throws IOException {
    ByteArrayOutputStream out = new ByteArrayOutputStream();
    DataOutputStream dOut = new DataOutputStream(out);

    dOut.writeUTF(getName());
    dOut.writeUTF(getArtist() != null ? getArtist() : DEFAULT_VALUE);
    dOut.writeUTF(getDatePublished() != null ? getDatePublished() : DEFAULT_VALUE);
    dOut.writeUTF(Integer.toString(getRating()));
    dOut.flush();
    return out.toByteArray();
}

public void deserialize(byte[] array) throws IOException {
```

```
    ByteArrayInputStream in = new ByteArrayInputStream(array);
    DataInputStream dIn = new DataInputStream(in);

    setName(dIn.readUTF());
    artist = dIn.readUTF();
    if (artist.equals(DEFAULT_VALUE)) {
        artist = null;
    }
    String date = dIn.readUTF();
    if (date.equals(DEFAULT_VALUE)) {
        datePublished = null;
    }
    else {
        setDatePublished(date);
    }
    rating = Integer.parseInt(dIn.readUTF());
}
```

The class implements the `getInsertCommand`, `getUpdateCommand`, and `getDeleteCommand` methods using the constructors specific to this business object:

```
/**
 * {@inheritDoc}
 */
@Override
public MusicAlbumInsertCommand getInsertCommand() {
    return new MusicAlbumInsertCommand(this, getSQLConnection(),
            getInsertString());
}

/**
 * {@inheritDoc}
 */
@Override
public MusicAlbumUpdateCommand getUpdateCommand() {
    return new MusicAlbumUpdateCommand(this, getSQLConnection(),
            getUpdateString());
}

/**
 * {@inheritDoc}
 */
@Override
public MusicAlbumDeleteCommand getDeleteCommand() {
    return new MusicAlbumDeleteCommand(this, getSQLConnection(),
            getDeleteString());
}
```

One of the constructor arguments for each command is the value returned by a helper method (`getInsertString`, `getUpdateString`, `getDeleteString`) that generates an SQL statement string. These methods are implemented as follows:

```
/**
 * Returns an SQL insert statement to add this instance
 * to the database.
 */
public String getInsertString() {
    stringBuilder.setLength(0);
    stringBuilder.append(hasLastModified() ?
        "INSERT INTO album" :
        "INSERT INTO album (name, artist, date_published, rating, username)")
        .append(" VALUES ('")
        .append(getName()).append("', '")
        .append(artist).append("', DATE '")
        .append(datePublished.get(Calendar.YEAR)).append("-")
        .append(datePublished.get(Calendar.MONTH) + 1).append("-")
        .append(datePublished.get(Calendar.DAY_OF_MONTH)).append("', ")
        .append(Integer.toString(rating)).append(", '")
        .append(getBusinessObjectProvider().getUsername())
        .append("'");

    if (hasLastModified()) {
        Timestamp timestamp = new Timestamp(getLastModified());
        stringBuilder.append(", TIMESTAMP '")
                .append(timestamp.toString())
                .append("'");
    }
    stringBuilder.append(')');

    return stringBuilder.toString();
}

/**
 * Returns an SQL update statement to modify this instance
 * in the database.
 */
public String getUpdateString() {
    stringBuilder.setLength(0);
    stringBuilder.append("UPDATE album SET artist='")
         .append(artist).append("', date_published=DATE '")
         .append(datePublished.get(Calendar.YEAR)).append("-")
         .append(datePublished.get(Calendar.MONTH) + 1).append("-")
         .append(datePublished.get(Calendar.DAY_OF_MONTH)).append("', rating=")
         .append(Integer.toString(rating));

    if (hasLastModified()) {
```

```
            Timestamp timestamp = new Timestamp(getLastModified());
            stringBuilder.append(", last_modified = TIMESTAMP '")
                .append(timestamp.toString()).append("'");
        }

        stringBuilder.append(" WHERE name = '").append(getName())
             .append("' AND username = '")
             .append(getBusinessObjectProvider().getUsername())
             .append("'");
        return stringBuilder.toString();
    }

    /**
     * Returns an SQL delete statement to remove this instance
     * from the database.
     */
    public String getDeleteString() {
        stringBuilder.setLength(0);
        stringBuilder.append("DELETE FROM album WHERE name = '")
                     .append(getName())
                     .append("' AND username = '"
                            + getBusinessObjectProvider().getUsername() + "'");
        return stringBuilder.toString();
    }
```

You may notice that the SQL statements show an additional column, `username`, in the database's `album` table in addition to columns for the `MusicAlbum` class's properties (`name`, `artist`, `datePublished`, and `rating`). The `username` column and the `name` column provide a composite primary key for the `album` table. The `username` column identifies the owner of an album and allows the `MusicAlbumProvider.getBusinessObjects` method to return only the albums for a particular user, so that multiple users can share the `album` table.

The `getInsertString` and `getUpdateString` methods also set the timestamp of the new or modified business object if the business object uses timestamps.

An additional method, `getSelectString`, is provided for testing purposes.

Another constructor argument for the commands is the value returned by another helper method, `getSQLConnection`, which returns a JDBC `Connection` object created by the `MusicAlbumProvider` class.

```
    /**
     * Returns a connection object that can be used to
     * execute SQL commands.
     */
    public Connection getSQLConnection() {
        return getBusinessObjectProvider().getSQLConnection();
    }
```

Extending the BusinessObjectProvider Class

The BusinessObjectProvider class serves several purposes:

- It allows you to retrieve all the business objects from a database by calling the getBusinessObjects() method.
- It allows you to create new business objects by calling the newBusinessObject() method.
- It provides access to a transaction manager and a session context.

For details on this class, see "The BusinessObjectProvider Class" on page 77.

For the MusicDB example, the class that extends BusinessObjectProvider is MusicAlbumProvider. Like the file for the MusicAlbum class, the MusicAlbumProvider.java source file begins by importing Java SE packages and the required ECBO API classes. It then begins by setting up a logger and declaring some string constants, a JDBC connection object, its implementation of the TransactionManager class, and a user name object:

```
public class MusicAlbumProvider extends BusinessObjectProvider<MusicAlbum> {

    static final Logger logger = BusinessObjectProvider.getLogger();

    public static final String REPOSITORY_NAME = "MusicDbRepository";
    public static final String MUSICDB_JNDI_DATASOURCE = "jdbc/musicdb";
    public static final String DB_USER_NAME = "musicdbuser";
    public static final String DB_USER_PASS = "musicdbpass";
    Connection sqlConnection = null;

    MusicAlbumTransactionManager transactionManager;

    String username;
```

The REPOSITORY_NAME value is identical to the repository name specified in the resource file for the Enterprise Connector. In the MusicDB sample, the resource file is named MusicDbRepository.xml and defines a repository named MusicDbRepository. The MUSICDB_JNDI_DATASOURCE value defines the name of the JDBC Datasource used to connect with the database.

The code implements two forms of the business object constructor: the no-argument constructor specified by the API and a one-argument form that takes a user name as argument for testing purposes.

Next, the code implements the two lifecycle methods for the BusinessObjectProvider class, initialize and terminate, which coincide with the start and end of a synchronization session.

The initialize method allocates resources required for a synchronization session or for database authentication. In this case, the code does the following:

- Looks up a JDBC datasource.
- Retrieves the username value from the SessionContext for use in updating the album table in the database.
- Obtains a JDBC connection using the credentials established for the MusicDB database when it was created.
- Instantiates a transaction manager.

```
/**
 * Creates a connection to the {@link #MUSICDB_JNDI_DATASOURCE}
 * database.
 */
@Override
public void initialize() {
    logger.fine("Initializing provider " + this);

    try {
        Context jndiContext = new InitialContext();
        DataSource ds = null;

        // If unable to get JNDI datasource, use local one for testing
        try {
            ds = (DataSource) jndiContext.lookup(MUSICDB_JNDI_DATASOURCE);
        } catch (NoInitialContextException e) {
            ds = new MusicDbDataSource();       // testing only!
        }

        // Get database credentials from provider's context
        SessionContext sessionContext = getSessionContext();
        username = sessionContext.getUsername();
        if (username == null || !username.startsWith(DB_USER_NAME)) {
            throw new RuntimeException("A MusicDB user name must start "
                    + "with the '"+ DB_USER_NAME + "' prefix");
        }

        // Get connection using default credentials
        sqlConnection = ds.getConnection(DB_USER_NAME, DB_USER_PASS);

        // Init transaction manager
        transactionManager = new MusicAlbumTransactionManager();
    }
    catch (SQLException ex) {
        throw new RuntimeException(ex);
    }
    catch (NamingException ex) {
```

```
            throw new RuntimeException(ex);
        }
    }
```

The implementation of the `terminate` method releases any resources allocated by the `initialize` method. In this case, it closes the JDBC connection.

```
    /**
     * Closes a connection to the {@link #MUSICDB_JNDI_DATASOURCE}
     * database.
     */
    @Override
    public void terminate() {
        logger.debug("Terminating provider " + this);

        try {
            if (sqlConnection != null) {
                sqlConnection.close();
            }
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

The implementation of the `getRepositoryName` method specifies the string value declared at the beginning of the class. The repository in question is the repository that is used for communication between the Gateway Engine and the Enterprise Connector and that is specified by the resource file.

```
    /**
     * {@inheritDoc}
     */
    @Override
    public String getRepositoryName() {
        return REPOSITORY_NAME;
    }
```

The implementation of the `getBusinessObjects` method uses a JDBC query to retrieve all the albums for the user `username` from the database, instantiates a `MusicAlbum` object for each retrieved album, and adds it to an `ArrayList` of albums.

```
    /**
     * {@inheritDoc}
     */
    @Override
    public List<MusicAlbum> getBusinessObjects() {
        logger.fine("Getting objects from provider " + this);
```

```
        Statement stmt = null;
        List<MusicAlbum> albums = null;

        try {
            stmt = sqlConnection.createStatement();

            // Read all music albums and store them in array
            albums = new ArrayList<MusicAlbum>();
            ResultSet rs = stmt.executeQuery(
                    "SELECT * FROM album WHERE username = '" + username + "'");
            while (rs.next()) {
                MusicAlbum album = new MusicAlbum(this);
                album.setName(rs.getString(1));
                album.setArtist(rs.getString(2));
                album.setDatePublished(rs.getString(3).replace("-", ""));
                album.setRating(rs.getInt(4));
                if (album.hasLastModified()) {
                    album.setLastModified(rs.getTimestamp(6).getTime());
                }
                albums.add(album);
            }
            rs.close();
        } catch (Exception ex) {
            throw new RuntimeException(ex);
        } finally {
            if (stmt != null) {
                try { stmt.close(); } catch (Exception e) { /* ignore !*/ }
            }
        }
        return albums;
    }
```

The implementation of the `newBusinessObject` method is much simpler: it calls the one-argument constructor for `MusicAlbum`.

```
    /**
     * {@inheritDoc}
     */
    @Override
    public MusicAlbum newBusinessObject() {
        return new MusicAlbum(this);
    }
```

The provider class also implements the helper method `getSQLConnection`, which returns the JDBC connection that was instantiated by the `initialize` method. This method is called by the `MusicAlbum` class.

```
/**
 * Returns a connection object that can be used to
 * execute SQL commands.
 */
public Connection getSQLConnection() {
    return sqlConnection;
}
```

The provider class also implements a `getUsername` method that is called by the `MusicAlbum` class's utility methods:

```
/**
 * Returns the user name logged into the session that
 * created this provider.
 */
public String getUsername() {
    return username;
}
```

The MusicDB provider also implements a `useTimestamps` method, which returns `true` if the provider is using timestamps to determine when an object was last modified. This method is not part of the ECBO API but is specific to this Enterprise Connector. Another connector may use another mechanism to implement the `lastModified` property.

```
/**
 * Returns the value of parameter 'useTimestamps'. This
 * parameter controls the use of timestamps over digests
 * by the gateway. Default value is false.
 */
public boolean useTimestamps() {
    SessionContext sessionContext = getSessionContext();
    String v = sessionContext.getParameters().get("useTimestamps");
    return (v == null) ? false : v.equalsIgnoreCase("true");
}
```

An administrator can set the value of the `useTimestamps` property when defining the Enterprise Connector in the Sun GlassFish Mobility Platform Administration Console. If the property is not set, the `useTimestamps` method sets the value to false.

The provider implements the `mergeBusinessObjects` method to resolve conflicts between the client and server:

```
/**
 * If timestamps are available then pick the object that
 * has been updated last. Otherwise, since we don't know
 * which object was updated last, we just compute the
 * average of the ratings and assume that all the other
 * fields have not been updated.
```

```
 */
@Override
public void mergeBusinessObjects(MusicAlbum serverObject,
        MusicAlbum clientObject)
{
    if (clientObject.hasLastModified() && serverObject.hasLastModified()) {
        if (clientObject.getLastModified() > serverObject.getLastModified()) {
            serverObject.setArtist(clientObject.getArtist());
            serverObject.setDatePublished(clientObject.getDatePublished());
            serverObject.setRating(clientObject.getRating());
        }
    }
    else {
        serverObject.setRating(
                (clientObject.getRating() + serverObject.getRating()) / 2);
    }
}
```

The `getTransactionManager` method retrieves the `MusicAlbumTransactionManager` that is declared at the beginning of the file, instantiated in the `initialize` method, and implemented within the `MusicAlbumProvider.java` file.

```
/**
 * Returns a transaction manager that uses JDBC to start,
 * stop and abort transactions.
 */
@Override
public MusicAlbumTransactionManager getTransactionManager() {
    return transactionManager;
}
```

Extending the `TransactionManager` Class

The `MusicAlbumProvider.java` file includes the implementation of the `TransactionManager` class. The `TransactionManager` class can be implemented in a separate file, but the relationships between the two classes mean that it is simpler to keep them together. It is also possible to use the default implementation of the `TransactionManager` class instead of implementing it yourself.

For details on this class, see "The `TransactionManager` Class" on page 80.

The MusicDB implementation of this class is called `MusicAlbumTransactionManager`. This class manages the database transactions for `MusicAlbum` objects using the JDBC API. It turns the database's auto-commit feature off if it has one and starts, stops, and aborts database transactions. The class definition begins with the constructor, which takes no arguments.

```
public class MusicAlbumTransactionManager extends
        TransactionManager<MusicAlbumProvider> {

    public MusicAlbumTransactionManager() {
        super(MusicAlbumProvider.this);

        assert (sqlConnection != null);
        try {
            sqlConnection.setAutoCommit(false);
        }
        catch (SQLException e) {
            // Ignore if not supported by DB
        }
    }
```

The abortTransaction method calls the JDBC Connection.rollback method:

```
    @Override
    public void abortTransaction() {
        logger.debug("Aborting transaction on SQL connection "
                + sqlConnection);
        try {
            sqlConnection.rollback();
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
```

The beginTransaction and endTransaction methods are closely linked. The endTransaction method commits the current transaction, an action that automatically starts the next transaction. The beginTransaction method simply calls endTransaction.

```
    @Override
    public void beginTransaction() {
        endTransaction();   // starts a new one
    }

    @Override
    public void endTransaction() {
        logger.debug("Starting/Committing transaction on SQL connection "
                + sqlConnection);
        try {
            sqlConnection.commit();
        }
        catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
```

Extending the InsertCommand, UpdateCommand, and DeleteCommand Classes

The InsertCommand, UpdateCommand, and DeleteCommand classes all extend the Command class.

For details on these classes, see "The InsertCommand Class" on page 79, "The UpdateCommand Class" on page 80, "The DeleteCommand Class" on page 78, and "The Command Class" on page 78.

For the MusicDB example, the implementations of the three classes are almost identical. The source files for the implementations are MusicAlbumInsertCommand.java, MusicAlbumUpdateCommand.java, and MusicAlbumDeleteCommand.java.

Each source file begins by importing Java SE packages and the required ECBO API classes. It then begins by setting up a logger and declaring two objects that are used by the constructor method and the execute method. For example, MusicAlbumInsertCommand.java begins as follows:

```
public class MusicAlbumInsertCommand extends InsertCommand<MusicAlbum> {

    static final Logger logger = BusinessObjectProvider.getLogger();

    private String sqlStatement;

    private Connection sqlConnection;
```

The code then extends the class constructor. While the constructor for the base class takes one argument, the constructor for each of the implementation classes takes three arguments: the MusicAlbum, a string that represents a SQL statement, and a JDBC connection. For example, the constructor for MusicAlbumUpdateCommand looks like this:

```
    public MusicAlbumUpdateCommand(MusicAlbum album,
            Connection sqlConnection, String sqlStatement)
    {
        super(album);
        this.sqlConnection = sqlConnection;
        this.sqlStatement = sqlStatement;
        logger.debug("Creating instance " + this + ": " + sqlStatement);
    }
```

Finally, the code for each command implements the execute method in exactly the same way. First, it uses the instantiated connection to create a JDBC Statement object using the instantiated string. Then it executes the statement.

```
    public void execute() {
        Statement stmt = null;
```

```
        try {
            logger.fine("Executing instance " + this + ": " + sqlStatement);
            stmt = sqlConnection.createStatement();
            stmt.executeUpdate(sqlStatement);
        }
        catch (Exception ex) {
            throw new RuntimeException(ex);
        }
        finally {
            if (stmt != null) {
                try { stmt.close(); } catch (Exception e) { /* ignore !*/ }
            }
        }
    }
```

Creating the Resource File for an Enterprise Connector

The resource file that you need to package with an Enterprise Connector (as described in "Developing Connectors Using the Enterprise Connector Business Object (ECBO) API" on page 12) is an XML file. The name of the file typically refers to the database or EIS/EAI system. For example, the resource file for MusicDB is named `MusicDbRepository.xml`. The file begins and ends as follows:

```
<jeceira xmlns:nt="http://www.jcp.org/jcr/nt/1.0"
    xmlns:jcr="http://www.jcp.org/jcr/1.0"
    xmlns:sync="http://www.synchronica.com/jcr/types"
    xmlns:aprzv="http://www.aparzev.com/jrc/aprzv"
    xmlns:udc="http://www.synchronica.com/udc/types/1.0">
    <repositories>
        <repository name="MusicDbRepository">

            <workspaces>
                <workspace name="MusicDbWorkspace" />
            </workspaces>
            ....
        </repository>
    </repositories>
</jeceira>
```

All resource files are identical except for two values:

- The `name` attribute of the `repository` element, in this case `MusicDbRepository`
- The `name` attribute of the `workspace` element, in this case `MusicDbWorkspace`

To create your own resource file, you can copy the resource file from the sample Enterprise Connector source directory, rename it, and modify these two values. In the unzipped client

bundle, you can find the file in `sgmp-client-1_1_01-fcs-b02/samples/secure-musicdb/src/connector/ecbo/MusicDbRepository.xml`.

You use these values when you configure the Enterprise Connector in the Sun GlassFish Mobility Platform Administration Console. See "Using the Connectors Tab" in *Sun GlassFish Mobility Platform 1.1 Administration Guide* for details.

Using NetBeans IDE To Create an ECBO Enterprise Connector

The simplest way to create an ECBO Enterprise Connector is to use NetBeans IDE 6.5 with the Maven plugin.

▼ To Create a Maven Project for an ECBO Enterprise Connector

Before You Begin Follow the instructions in "To Install the Maven Plugin" on page 24 to set up NetBeans IDE.

1. **In NetBeans IDE, click the Projects tab.**
2. **Choose File→New Project.**
3. **In the New Project dialog, select Maven from the Categories tree.**
4. **Select Maven Project from the Projects list, then click Next.**
5. **On the Maven Archetype page, expand Archetypes from the remote Maven Repositories node.**
6. **From the list, select MEP Connector Archetype (RAR), then click Next.**
7. **On the Name and Location page, accept all the default values, then click Finish.**
   The new project, mavenproject2 (rar), appears in the Projects tab.
8. **Under mavenproject2, expand the Source Packages node.**
9. **Expand the com.mycompany.mavenproject2 node.**
   The following class files appear:
   - `MyBusinessObject.java`
   - `MyBusinessObjectProvider.java`
   - `MyBusinessObjectDeleteCommand.java`
   - `MyBusinessObjectInsertCommand.java`

- `MyBusinessObjectUpdateCommand.java`

These files provide the required classes for an ECBO Enterprise Connector.

10 Expand the com.mycompany.mavenproject2.rar node.

The following class files appear:

- `Adapter.java`
- `InitializationJob.java`
- `ManagedConnection.java`
- `ManagedConnectionFactory.java`
- `Metadata.java`

These classes implement the resource adapter. The `ManagedConnectionFactory.java` code contains a reference to the `MyBusinessObjectProvider` class.

11 Rename the classes so that they are specific to your Enterprise Connector.

12 In the `ManagedConnectionFactory.java` file, provide a unique ID for the `serialVersionUID` field to replace the value in the file.

This step is not required, but it is recommended.

13 Add code to the ECBO Enterprise Connector files.

No changes to the resource adapter files are needed other than the name and unique ID changes.

14 Build the project.

When you build the project, all the class files are bundled together in a RAR file.

Deploying and Configuring an ECBO Enterprise Connector

The tasks involved in deploying and configuring an ECBO Enterprise Connector are very similar to the tasks for a JAX-RS Enterprise Connector.

By default, the MusicDB ECBO Enterprise Connector is both deployed and configured in an installed version of Sun GlassFish Mobility Platform. There is no need to redeploy it.

The steps to deploy and configure an ECBO Enterprise Connector are very similar to those in "Building, Deploying, and Configuring a JAX-RS Enterprise Connector" on page 35, with a few exceptions:

"To Deploy an Enterprise Connector" on page 36
You deploy the Enterprise Connector under the Connector Modules node of the Administration Console, not the Web Applications node. An ECBO Enterprise Connector is a resource adapter.

"To Create a Connector Connection Pool for the Enterprise Connector" on page 37
The resource adapter for the connection pool is the Enterprise Connector itself, so select the deployed ECBO Enterprise Connector from the Resource Adapter drop-down list.

"To Verify that an Enterprise Connector Is Deployed" on page 39
Do not perform this task.

"To Configure the Enterprise Connector on Sun GlassFish Mobility Platform" on page 39
In the Local Enterprise Connector Settings panel, the Workspace Name should be the same as the name specified for the `workspace` element in the resource file for the Enterprise Connector.

In the Local Enterprise Connector Properties panel, do not specify a value for the `uri` property. Instead, you may specify a value for the `business-object-provider` property (see the configured MusicDB ECBO Enterprise Connector for an example), but this is not required if you use the Maven plugin to create the Enterprise Connector.

◆ ◆ ◆ CHAPTER 4

Accessing a Sun JCA Adapter for an EIS/EAI System

If you are designing your Enterprise Connector to access an EIS/EAI system instead of a database, the Enterprise Connector can access the Sun JCA Adapter for that system instead of making JDBC calls.

You must first create an Object Type Definition (OTD) that maps your business object properties to data on the EIS/EAI system. To create the OTD, you need to use NetBeans IDE with plugins that are provided with Sun GlassFish Mobility Platform.

After you create the OTD, use the NetBeans IDE code completion feature to call methods on the classes generated by the OTD wizard.

This section covers the following topics:

Creating an Object Type Definition (OTD)

For information on working with Sun JCA Adapters, see the Designing section of the Java CAPS documentation (`http://developers.sun.com/docs/javacaps/designing/`). Specific sections you will need to look at include the following:

- Technical Overview for Sun JCA Adapters (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.cnfg_tech-ovrvw_t.html`)
- Object Type Definition Wizards (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.ggrbu.html`)

  The two middle sections on inbound and outbound JCA Resource Adapter client code are not relevant to Enterprise Connectors.

To obtain the NetBeans IDE plugins needed to create an OTD, follow the instructions in Installation of Netbeans Modules (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.inst_jca-adapter_t.html`). The `NetBeansModules` referred to in these instructions are part of your Sun GlassFish Mobility Platform installation. In the location where you unzipped the installation bundle `sgmp-1_1-`*operating-system*`.zip`, you will find them in the directory `sgmp1.1-3.1.39/NetBeansModules`. (The Java CAPS documentation states that they are in the directory `AdapterPack/NetBeansModules/CommonLib`, but for Sun GlassFish Mobility Platform they are in the directory `NetBeansModules/commonlib`.)

To develop an OTD for your application, follow the instructions appropriate to your EIS/EAI system in Developing OTDs for Application Adapters (`http://developers.sun.com/docs/javacaps/designing/dotdappadptr.dotdappadptr_intro.html`).

The instructions for adapters supported by Sun GlassFish Mobility Platform are in the following sections:

- Creating SAP BAPI OTDs (`http://developers.sun.com/docs/javacaps/designing/dotdappadptr.dsgn_sap-bapi-otd_t.html`)
- Creating Siebel EAI OTDs (`http://developers.sun.com/docs/javacaps/designing/dotdappadptr.dsgn_siebel-eai-otd_t.html`)
- Using the Oracle Applications Wizard and JCA Adapter Tooling with an EJB Project (`http://developers.sun.com/docs/javacaps/designing/jcapssunjcaad.cnfg_oracleapps-wiz_t.html`)

An Enterprise Connector is a Sun JCA Adapter client application that is not an Enterprise JavaBeans (EJB) component. You may find that you need to create the OTD and develop the Enterprise Connector inside an EJB project. However, you should then remove the Enterprise Connector and OTD from the EJB JAR file and place them in an ordinary JAR file before you place the JAR file in the `domains/mep/lib` directory for the Enterprise Server. The OTD is generated in a separate JAR file, so it is easy to copy it to another project.

An Enterprise Connector is a Sun JCA Adapter client application that is not an Enterprise JavaBeans (EJB) component. You may find that you need to create the OTD and develop the Enterprise Connector inside an EJB project. However, you should then remove the Enterprise Connector and OTD from the EJB JAR file, repackage the Enterprise Connector in a RAR or WAR (depending on whether it uses the ECBO or JAX-RS API), package the OTD as an ordinary JAR file, and then place the OTD JAR file in the RAR or WAR file.

Writing Code to Access a Sun JCA Adapter

To access a Sun JCA Adapter, your code needs to use Java CAPS APIs, which are documented at `http://developers.sun.com/docs/javacaps/reference/javadocs/index.jsp`.

This section describes how to extend the ECBO classes to access a Sun JCA Adapter.

The default implementation of the `TransactionManager` class may be sufficient for your application.

Note – This section describes how to make these changes for an ECBO Enterprise Connector. If you are developing a JAX-RS Enterprise Connector, you should be able to adapt these instructions to your needs.

Extending the `BusinessObjectProvider` Class to Access a Sun JCA Adapter

To allow your Enterprise Connector to work with a Sun JCA Adapter, your `BusinessObjectProvider` implementation needs to create a connection to the Adapter in its `initialize` method, close that connection in its `terminate` method, and retrieve objects through the Adapter in its `getBusinessObject` method.

Note – For a JAX-RS Enterprise Connector, make make similar changes in your implementation of the `MyBusinessObjectsResource.lifeCycle` method.

For a SAP BAPI application, for example, you import the following packages:

```
import com.stc.connector.appconn.common.ApplicationConnectionFactory;
import com.stc.connector.appconn.common.ApplicationConnection;
import com.stc.connector.sapbapiadapter.appconn.SAPApplicationConnection;
import com.stc.util.OtdObjectFactory;
```

When you create a provider for a `Customer` business object, you declare objects like the following. The `customer.Customer` class is generated by the OTD wizard.

```
public class CustomerProvider extends BusinessObjectProvider<Customer> {
    ...
    public static final String SAP_JNDI_DATASOURCE = "jcaps/sap";
```

```
public static final String REPOSITORY_NAME = "SAPRepository";

private ApplicationConnectionFactory mJCAsapadapter = null;
private ApplicationConnection mJCAsapadapterConnection = null;
private customer.Customer mJCAsapcustomerCommObj = null;
```

The provider's `initialize` method then allocates these resources. It obtains an `ApplicationConnectionFactory` object by means of a JNDI lookup, then uses the factory to create the `ApplicationConnection` object. These method calls are the same no matter which EIS/EAI system you are using:

```
@Override
public void initialize() {
    logger.debug("Initializing provider " + this);

    try {
        InitialContext ic = new InitialContext();
        // First get ApplicationConnectionFactory through JNDI lookup
        mJCAsapadapter =
            (ApplicationConnectionFactory) ic.lookup(SAP_JNDI_DATASOURCE);

        /* Then create ApplicationConnection. One AppConn can be dynamically
         * allocated to a physical connection defined in connection pool;
         * this results in connection reuse according to JCA and Appserver
         * contract
         */
        mJCAsapadapterConnection = mJCAsapadapter.getConnection();
```

The `initialize` method then uses the `OtdObjectFactory` to create an instance of a SAP customer communication object. Methods called on this object are specific to the SAP OTD. The code casts the generic `ApplicationConnection` object `mJCAsapadapterConnection` to another application connection specific to SAP:

```
        /* Create Customer communication object
         */
        mJCAsapcustomerCommObj =
            (customer.Customer) OtdObjectFactory.createInstance(null,
                "customer.Customer");

        /* Set ApplicationConnection on Customer communication object
         */
        mJCAsapcustomerCommObj.setAppConn(
            (SAPApplicationConnection) mJCAsapadapterConnection);
```

The `initialize` method next uses the ECBO API `SessionContext` object to retrieve the user name and password. It then uses these values to create user credentials specific to SAP, and finally connects to the Sun JCA Adapter for SAP.

```
            // Get backend credentials from provider's context
            SessionContext sessionContext = getSessionContext();
            String param = sessionContext.getUsername();
            if (param != null) {
                mJCAsapcustomerCommObj.getSAPConnectionParams().setUserid(param);
            }
            param = sessionContext.getPassword();
            if (param != null) {
                mJCAsapcustomerCommObj.getSAPConnectionParams().setPassword(param);
            }
            mJCAsapcustomerCommObj.connectWithNewParams();

            ic.close();
        }
        catch (Exception ex) {
            logger.debug("Initializing provider exception" + ex.getMessage());
            throw new RuntimeException(ex);
        }
    }
```

The `terminate` method closes the connection created by the `initialize` method:

```
    @Override
    public void terminate() {
        logger.debug("Terminating provider " + this);

        try {
            if (mJCAsapadapterConnection != null) {
                mJCAsapadapterConnection.close();
                logger.info("terminate provider close connection"
                        + mJCAsapadapterConnection.toString());
            }
        }
        catch (Exception e) {
            logger.debug("terminating provider exception" + e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

The provider code also implements a utility method, `getSAPCustomerClient`, which retrieves the customer communication object:

```
    /**
     * @return SAPCustomerClient object that can be used to
     * operate on a Customer BAPI.
     */
    public customer.Customer getSAPCustomerClient() {
        return mJCAsapcustomerCommObj;
    }
```

The `getBusinessObjects` method uses the `getSAPCustomerClient` method to retrieve the SAP customer data and store it in the Enterprise Connector's `Customer` object. It again calls methods on the communication object generated by the OTD wizard.

Note – For a JAX-RS Enterprise Connector, make similar changes in your implementation of the `MyBusinessObjectsResource.getBusinessObjects` method.

```
@Override
//Retrieve all IDocs and map here between Customer and VendorAccount Object
public List<Customer> getBusinessObjects() {
    logger.debug("Getting objects from provider " + this);

    HashMap<String, Customer> customerMap = new HashMap<String, Customer>();

    try {
        // Getting customer list
        getSAPCustomerClient().getGetList().getIDRANGE(0).setOPTION("CP");
        getSAPCustomerClient().getGetList().getIDRANGE(0).setLOW("*");
        logger.info("Executing Customer with the following values Option " +
            "[" + getSAPCustomerClient().getGetList().getIDRANGE(0).getLOW()
            + "] Option ["
            + getSAPCustomerClient().getGetList().getIDRANGE(0).getOPTION() + "]");
        getSAPCustomerClient().getGetList().execute();

        // Process returned data and populate customer list
        customer.Customer.GetList.ExportParams.RETURN ret =
            getSAPCustomerClient().getGetList().getExportParams().getRETURN();
        logger.info("Retrieved ["
            + getSAPCustomerClient().getGetList().countADDRESSDATA()
            + "] customers");

        customer.Customer.GetList.ADDRESSDATA[] addressList =
            getSAPCustomerClient().getGetList().getADDRESSDATA();
        for (int i = 0; i < addressList.length; i++) {
            customer.Customer.GetList.ADDRESSDATA addr = addressList[i];

            // Ignore companies whose names start with "DELETED" -- hack
            if (!addr.getNAME().startsWith("DELETED")) {
                // Ignore customers whose names are repeated
                if (customerMap.containsKey(addr.getNAME())) {
                    continue;
                }

                // Create a new Customer instance
                Customer comp = new Customer(this);
```

```
// Set unique name for business object
comp.setName(addr.getNAME());

// Set customer number and name
comp.setCustomerNumber(addr.getCUSTOMER());
comp.setCustomerName(addr.getNAME());

// Get sales area
getSAPCustomerClient().getGetSalesAreas().getImportParams()
    .setCUSTOMERNO(comp.getCustomerNumber());
getSAPCustomerClient().getGetSalesAreas().execute();
String retNo = getSAPCustomerClient().getGetSalesAreas()
    .getExportParams().getRETURN().getMESSAGE();
String retMsg = getSAPCustomerClient().getGetSalesAreas()
    .getExportParams().getRETURN().getCODE();
logger.info("Return Number [" + retNo + "] retMsg ["
    + retMsg + "].");
if (retNo.length() > 0) {
    throw new RuntimeException(retMsg);
}

// Set sales related fields
comp.setSalesOrg(getSAPCustomerClient().getGetSalesAreas()
    .getSALESAREAS(0).getSALESORG());
comp.setDistChannel(getSAPCustomerClient().getGetSalesAreas()
    .getSALESAREAS(0).getDISTRCHN());
comp.setDivision(getSAPCustomerClient().getGetSalesAreas()
    .getSALESAREAS(0).getDIVISION());

// Get detail on customer
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setCUSTOMERNO(comp.getCustomerNumber());
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setPI_SALESORG(comp.getSalesOrg());
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setPI_DISTR_CHAN(comp.getDistChannel());
getSAPCustomerClient().getGetDetail1().getImportParams()
    .setPI_DIVISION(comp.getDivision());
getSAPCustomerClient().getGetDetail1().execute();
retNo =
    getSAPCustomerClient().getGetDetail1().getExportParams()
        .getRETURN().getMESSAGE();
retMsg =
    getSAPCustomerClient().getGetDetail1().getExportParams()
        .getRETURN().getNUMBER();
logger.info("Return Number [" + retNo + "] retMsg ["
    + retMsg + "].");
if (retNo.length() > 0) {
```

```
                    throw new RuntimeException(retMsg);
                }

                // Populate customer object data
                customer.Customer.GetDetail1.ExportParams.PE_COMPANYDATA
                    currAddr = getSAPCustomerClient().getGetDetail1()
                        .getExportParams().getPE_COMPANYDATA();
                comp.setCity(currAddr.getCITY());
                comp.setPostalCode(currAddr.getPOSTL_COD1());
                comp.setStreet(currAddr.getSTREET());
                comp.setCountryKey(currAddr.getCOUNTRY());
                comp.setLanguageKey(currAddr.getLANGU_ISO());
                comp.setRegion(currAddr.getREGION());
                comp.setTelephone(currAddr.getTEL1_NUMBR());
                comp.setFaxNumber(currAddr.getFAX_NUMBER());
                comp.setCurrencyKey(currAddr.getCURRENCY());

                customerMap.put(comp.getCustomerName(), comp);
            }
        }
        return new ArrayList<Customer>(customerMap.values());
    }
    catch (Exception ex) {
        throw new RuntimeException(ex);
    }
}
```

The `getRepositoryName` and `newBusinessObject` methods have implementations very similar to those in the `MusicAlbumProvider` class:

```
@Override
public String getRepositoryName() {
    return REPOSITORY_NAME;
}

@Override
public Customer newBusinessObject() {
    return new Customer(this);
}
```

The other methods in the provider class use the default `BusinessObjectProvider` implementation: `getSessionContext`, `setSessionContect`, and `getTransactionManager`.

Extending the `BusinessObject` Class to Access a Sun JCA Adapter

The `BusinessObject` class for an Enterprise Connector that accesses a Sun JCA Adapter may have straightforward implementations of the `BusinessObject` methods, but it may also require some additional utility methods. A SAP BAPI `Customer` object, for example, implements a large number of getter and setter methods for its properties. Its `serialize` and `deserialize` methods can be relatively simple.

The `Customer` object implementations of the `getInsertCommand`, `getUpdateCommand`, and `getDeleteCommand` methods call the constructors for the command classes, as expected. However, here the constructors take two arguments, and the second argument is the value returned by a utility method.

```
/**
 * {@inheritDoc}
 */
@Override
public CustomerInsertCommand getInsertCommand() {
    return new CustomerInsertCommand(this, getInsertCustomer());
}

/**
 * {@inheritDoc}
 */
@Override
public CustomerUpdateCommand getUpdateCommand() {
    return new CustomerUpdateCommand(this, getUpdateCustomer());
}

/**
 * {@inheritDoc}
 */
@Override
public CustomerDeleteCommand getDeleteCommand() {
    return new CustomerDeleteCommand(this, getDeleteCustomer());
}
```

The utility methods use the provider class's `getSAPCustomerClient` method to retrieve first the customer communication object, and then the `CreateFromData1` object. For example, the `getInsertCustomer` method begins as follows:

```
/**
 * Returns a Customer CreateFromData1 object to be used for insert.
 */
public customer.Customer.CreateFromData1 getInsertCustomer() {
```

```
        customer.Customer.CreateFromData1 cfd = getBusinessObjectProvider()
                .getSAPCustomerClient().getCreateFromData1();
```

The rest of the `getInsertCustomer` method uses the `CreateFromData1` object to assign the `Customer` properties to the SAP BAPI customer object. Finally, it returns the `CreateFromData1` object.

```
       // Set import parameters
        cfd.getImportParams().getPI_COPYREFERENCE().setREF_CUSTMR(
            CustomerProvider.REF_CUSTOMER);
        cfd.getImportParams().getPI_COPYREFERENCE().setSALESORG(getSalesOrg());
        cfd.getImportParams().getPI_COPYREFERENCE().setDISTR_CHAN(getDistChannel());
        cfd.getImportParams().getPI_COPYREFERENCE().setDIVISION(getDivision());

        // Required import parameters
        cfd.getImportParams().getPI_COMPANYDATA().setNAME(getCustomerName());
        cfd.getImportParams().getPI_COMPANYDATA().setLANGU_ISO(getLanguageKey());
        cfd.getImportParams().getPI_COMPANYDATA().setCURRENCY(getCurrencyKey());
        cfd.getImportParams().getPI_COMPANYDATA().setCOUNTRY(getCountryKey());
        cfd.getImportParams().getPI_COMPANYDATA().setPOSTL_COD1(getPostalCode());
        cfd.getImportParams().getPI_COMPANYDATA().setCITY(getCity());

        // Additional import parameters
        cfd.getImportParams().getPI_COMPANYDATA().setSTREET(getStreet());
        cfd.getImportParams().getPI_COMPANYDATA().setREGION(getRegion());
        cfd.getImportParams().getPI_COMPANYDATA().setTEL1_NUMBR(getTelephone());
        cfd.getImportParams().getPI_COMPANYDATA().setFAX_NUMBER(getFaxNumber());

        return cfd;
    }
```

The `getUpdateCustomer` method is almost identical to the `getInsertCustomer` method except that it also marks the fields as having changed:

```
        // Mark fields to be changed
        String ex = "X";
        cfd.getImportParams().getPI_COMPANYDATAX().setNAME(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setLANGU_ISO(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setCURRENCY(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setCOUNTRY(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setPOSTL_COD1(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setCITY(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setSTREET(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setREGION(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setTEL1_NUMBR(ex);
        cfd.getImportParams().getPI_COMPANYDATAX().setFAX_NUMBER(ex);
```

Similarly, the `getDeleteCustomer` method informs SAP to delete a record by changing its name to begin with the string `DELETED`:

```
// Mark customer as deleted by prepending "DELETE" to the name
setCustomerName("DELETED " + getCustomerName());
logger.fine("Changing NAME field to [" + getCustomerName() + "].");
cfd.getImportParams().getPI_COMPANYDATA().setNAME(getCustomerName());
cfd.getImportParams().getPI_COMPANYDATAX().setNAME("X");
cfd.getImportParams().getPI_COMPANYDATA().setLANGU_ISO(getLanguageKey());
cfd.getImportParams().getPI_COMPANYDATA().setCURRENCY(getCurrencyKey());
cfd.getImportParams().getPI_COMPANYDATA().setCOUNTRY(getCountryKey());
cfd.getImportParams().getPI_COMPANYDATA().setPOSTL_COD1(getPostalCode());
cfd.getImportParams().getPI_COMPANYDATA().setCITY(getCity());
cfd.getImportParams().setCUSTOMERNO(getCustomerNumber());
cfd.getImportParams().setPI_SALESORG(getSalesOrg());
cfd.getImportParams().setPI_DISTR_CHAN(getDistChannel());
cfd.getImportParams().setPI_DIVISION(getDivision());
```

Extending the `InsertCommand`, `UpdateCommand`, and `DeleteCommand` Classes to Access a Sun JCA Adapter

Note – For a JAX-RS Enterprise Connector, make similar changes in your implementation of the `MyBusinessObjectResource.putBusinessObject` and `MyBusinessObjectResource.deleteBusinessObject` methods.

For the three command classes, you need to use generated classes and methods from the OTD. For a SAP BAPI application, for example, you need to import the following packages for the `CustomerInsertCommand` implementation:

```
import customer.Customer.CreateFromData1;
import customer.Customer.CreateFromData1.ExportParams.RETURN;
```

You then use the first of the imported classes in the class constructor, which takes two arguments instead of the single argument of the default implementation:

```
public CustomerInsertCommand(Customer bobject, CreateFromData1 cfd) {
    super(bobject);
    mCreateFromData = cfd;
    logger.debug("Creating instance " + this);
}
```

The `CreateFromData1` object passed to the constructor is the returned value from the `Customer` class's `getInsertCustomer` method.

You implement the `execute` command by calling the class's own `execute` method and retrieving any return value through the second imported class:

```
@Override
public void execute() {
    try {
        mCreateFromData.execute();
        RETURN ret = mCreateFromData.getExportParams().getRETURN();
        String retMsg = ret.getMESSAGE();
        String message = "SAP (" + ret.getNUMBER() + "): " + retMsg;
        logger.info(message);
        if (retMsg.length() > 0) {
            throw new RuntimeException(message);
        }
    }
    catch (RuntimeException e) {
        throw e;
    }
    catch (Exception e) {
        logger.severe(e.getMessage());
        throw new RuntimeException(e.getMessage(), e);
    }
}
```

For both the `CustomerDeleteCommand` and the `CustomerUpdateCommand` implementations, you import the following:

```
import customer.Customer.ChangeFromData1;
import customer.Customer.ChangeFromData1.ExportParams.RETURN;
```

The constructors and the `execute` methods for these classes use the `ChangeFromData1` object returned from the `Customer.getUpdateCustomer` and `Customer.getDeleteCustomer` methods. Otherwise the class implementations are identical to those of the `CustomerInsertCommand` implementation.

◆ ◆ ◆ CHAPTER 5

Classes and Methods in the Enterprise Connector Business Object API Package

The Enterprise Connector Business Object (ECBO) API contains one package, `com.sun.mep.connector.api`, that developers must use. This chapter summarizes the classes contained within this package:

The API documentation is included in the Sun GlassFish Mobility Platform client bundle. In the directory where you unzipped the client bundle (see the *Sun GlassFish Mobility Platform 1.1 Installation Guide* for details), it is in the directory `sgmp-client-1_1_01-fcs-b02/doc/ecbo/api`.

The `BusinessObject` Class

Table 5–1 lists the constructor and methods belonging to the `BusinessObject` class, the base class for all business objects. Business objects are the entities synchronized with client applications. Each business object instance is identified by a name, which is also used to name the file that holds the object on the mobile client. Business objects are created by business object providers; see "The `BusinessObjectProvider` Class" on page 77 for details.

TABLE 5–1 Class com.sun.mep.connector.api.BusinessObject

Method	Description
`BusinessObject(T bobjectProvider)`	Constructor that takes the `BusinessObjectProvider` for the business object as its argument.
`public abstract void deserialize(byte[] data)`	Deserializes a business object from a byte array.
`public T getBusinessObjectProvider()`	Returns a reference to the business object provider associated with this object.
`public abstract <T extends BusinessObject> DeleteCommand<T> getDeleteCommand()`	Returns a command that when executed can delete this business object from a back end.
`public abstract java.lang.String getExtension()`	Returns the default extension for business objects of this type. Extensions are used by the files holding these objects and must be part of the contract with clients. That is, clients and connectors must use the same extension for the same type of business object. Concrete subclasses should redefine this method.
`public abstract <T extends BusinessObject> InsertCommand<T> getInsertCommand()`	Returns a command that when executed can insert this business object into a back end.
`public long getLastModified()`	Returns a timestamp indicating the last time this business object was modified. The timestamp represents the number of milliseconds since "the epoch," namely January 1, 1970, 00:00:00 GMT.
`public java.lang.String getName()`	Returns the name of this business object. Names must be unique identifiers.
`public abstract <T extends BusinessObject> UpdateCommand<T> getUpdateCommand()`	Returns a command that when executed can update this business object in a back end.
`public boolean hasLastModified()`	Returns a flag indicating if the `lastModified` timestamp is supported by this connector. The Gateway will always prefer timestamps over digests to detect updates, if available.
`public abstract byte[] serialize()`	Serializes a business object into a byte array.
`public void setLastModified(long millis)`	Sets a timestamp indicating the last time this business object was modified. The timestamp must represent the number of milliseconds since "the epoch," namely January 1, 1970, 00:00:00 GMT.
`public void setName(java.lang.String name)`	Sets the name of this business object. Names must be unique identifiers.

The BusinessObjectProvider Class

Table 5–2 lists the constructor and methods belonging to the BusinessObjectProvider class. This provider class for instances of BusinessObject serves multiple purposes:

- It can be used to retrieve all the business objects from a back end by calling the getBusinessObjects() method.
- It can be used to create new business objects by calling the newBusinessObject() method.
- It provides access to a transaction manager and a session context. See "The TransactionManager Class" on page 80 and "The SessionContext Class" on page 79 for more information.

An instance of this class is created for each synchronization session. Use the methods initialize() and terminate() to allocate and free session resources.

TABLE 5–2 Class com.sun.mep.connector.api.BusinessObjectProvider

Method	Description
public BusinessObjectProvider()	No-argument constructor.
public abstract java.util.List<T> getBusinessObjects()	Returns a complete list of the business objects available in the back end associated with this provider. This is in essence a query method for all the instances in the back end.
public abstract java.lang.String getRepositoryName()	Returns the name of the repository holding these objects.
public SessionContext getSessionContext()	Returns the session context associated with this provider.
public <U extends BusinessObjectProvider> TransactionManager<U> getTransactionManager()	Returns the transaction manager associated with this provider, or null if no transaction manager has been set. If null is returned, transactions are not supported by this provider.
public void initialize()	This method is called right after an instance of this class is created. You can use it to allocate resources for the duration of a synchronization session. Other uses of this method include back-end authentication. Credentials needed for authentication are available from the SessionContext, which you can access by calling getSessionContext().

TABLE 5–2 Class com.sun.mep.connector.api.BusinessObjectProvider *(Continued)*

Method	Description
`public void mergeBusinessObjects(T serverObject, T clientObject)`	This method is called to resolve an update conflict. A connector is responsible for merging `serverObject` and `clientObject` and returning the result in `serverObject`. Calling `getLastModified()` may be helpful during the implementation of this method.
`public abstract T newBusinessObject()`	Returns a fresh instance of a business object.
`public void setSessionContext(SessionContext context)`	Sets the session context for this provider.
`public void terminate()`	This method is called when a synchronization session is about to be terminated. Use this method to free any resources allocated by this object.

The `Command` Class

Table 5–3 lists the constructor and methods belonging to the `Command`class. This class is the base class for all business object commands. The classes that extend this class are described in "The `DeleteCommand` Class" on page 78, "The `InsertCommand` Class" on page 79, and "The `UpdateCommand` Class" on page 80.

TABLE 5–3 Class com.sun.mep.connector.api.Command

Method	Description
`public Command(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.
`public T getBusinessObject()`	Returns the business object on which this command is executed.

The `DeleteCommand` Class

Table 5–4 lists the constructor and method belonging to the `DeleteCommand` class. This class is the base class for delete business object commands. It deletes a business object from a back end.

TABLE 5–4 Class com.sun.mep.connector.api.DeleteCommand

Method	Description
`public DeleteCommand(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.

The `InsertCommand` Class

Table 5–5 lists the constructor and method belonging to the `InsertCommand` class. This class is the base class for insert business object commands. It inserts a business object into a back-end.

TABLE 5–5 Class com.sun.mep.connector.api.InsertCommand

Method	Description
`public InsertCommand(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.

The `SessionContext` Class

Table 5–6 lists the constructor and methods belonging to the `SessionContext` class. This class stores contextual information about the session in which a `BusinessObjectProvider` is instantiated. This information includes credentials for logging into an EIS/EAI system or a database as well as well as properties associated with an Enterprise Connector.

TABLE 5–6 Class com.sun.mep.connector.api.SessionContext

Method	Description
`public SessionContext(java.lang.String username, java.lang.String password)`	Two-argument constructor that accepts username and password credentials.
`public java.util.Map<java.lang.String, java.lang.String> getParameters()`	Returns all properties specified in the Admin Console's definition for this Enterprise Connector.
`public java.lang.String getPassword()`	Returns the password used to log into the Enterprise Connector.

TABLE 5–6 Class com.sun.mep.connector.api.SessionContext *(Continued)*

Method	Description
`public java.lang.String getUsername()`	Returns the user name used to log into the Enterprise Connector.
`public void setParameter(java.lang.String name, java.lang.String value)`	Stores a name and value pair in the internal map.

The TransactionManager Class

Table 5–7 lists the constructor and methods belonging to the `TransactionManager` class. This class provides the transaction manager for a business object provider class. It supports methods for starting, stopping, and aborting back-end transactions.

TABLE 5–7 Class com.sun.mep.connector.api.TransactionManager

Method	Description
`public TransactionManager(T bobjectProvider)`	Constructor that creates a new `TransactionManager` for the specified business object provider.
`public void abortTransaction()`	Rolls back the current transaction on the back-end system. By default, this operation is a no-op.
`public void beginTransaction()`	Starts a new transaction on the back-end system. All business object updates, deletes and inserts will be executed in a transaction. By default, this operation is a no-op.
`public void endTransaction()`	Ends the current transaction on the back-end system. All business object updates, deletes and inserts will be executed in a transaction. By default, this operation is a no-op.
`public T getBusinessObjectProvider()`	Returns the business object manager associated with this transaction manager.

The UpdateCommand Class

Table 5–8 lists the constructor and method belonging to the `UpdateCommand` class. This class is the base class for update business object commands. It updates a business object in a back end.

TABLE 5–8 Class com.sun.mep.connector.api.UpdateCommand

Method	Description
`public UpdateCommand(T bobject)`	Constructor that takes a business object argument.
`public abstract void execute()`	Executes this command against a back end. Unchecked exceptions, such as `java.lang.RuntimeException`, can be used to report errors.

Index

G

I

J

L

M

N