Sun Java System Mobile Enterprise Platform 1.0 Developer's Guide for Client Applications

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

Part No: 820–3753–10
July 2008

081205@21288

Contents

Preface

This guide explains how to develop mobile client applications for Sun Java System Mobile Enterprise Platform 1.0 (MEP).

MEP is a comprehensive mobility solution that enables offline data access, data synchronization, and secure access to EIS/EAI applications such as Siebel and SAP.

MEP 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, formerly known as SyncML. 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 applications for Java Platform, Micro Edition (Java ME).

Before You Read This Book

Before reading this guide, you should be familiar with Java Platform, Micro Edition (Java ME).

How This Book Is Organized

This book contains the following chapters:

Mobile Enterprise Platform Documentation Set

The Mobile Enterprise Platform documentation set is available at `http://docs.sun.com/coll/1780.1`. To learn about Mobile Enterprise Platform, refer to the books listed in the following table.

TABLE P–1 Books in the Mobile Enterprise Platform Documentation Set

Book Title	Description
Sun Java System Mobile Enterprise Platform 1.0 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 Java System Mobile Enterprise Platform 1.0 Architectural Overview	Introduction to the architecture of Mobile Enterprise Platform.
Sun Java System Mobile Enterprise Platform 1.0 Installation Guide	Installing the software and its components, and running a simple application to verify that installation succeeded.
Sun Java System Mobile Enterprise Platform 1.0 Deployment Guide	Deployment of applications and application components to Mobile Enterprise Platform.
Sun Java System Mobile Enterprise Platform 1.0 Developer's Guide for Client Applications	Creating and implementing Java Platform, Mobile Edition (Java ME platform) applications for Mobile Enterprise Platform that run on mobile devices.
Sun Java System Mobile Enterprise Platform 1.0 Developer's Guide for Enterprise Connectors	Creating and implementing Enterprise Connectors for Mobile Enterprise Platform intended to run on Sun Java System Application Server.
Sun Java System Mobile Enterprise Platform 1.0 Administration Guide	System administration for Mobile Enterprise Platform, focusing on the use of the MEP Administration Console.

Application Server Documentation Set

When you install MEP, it is deployed to Sun Java System Application Server 9.1 Update 2.

The Application Server documentation set describes deployment planning and system installation. The Uniform Resource Locator (URL) for Application Server documentation is `http://docs.sun.com/coll/1343.5`. For an introduction to Application Server, refer to the books in the order in which they are listed in the following table.

TABLE P–2 Books in the Application Server Documentation Set

Book Title	Description
Documentation Center	Application Server documentation topics organized by task and subject.

TABLE P–2 Books in the Application Server Documentation Set *(Continued)*

Book Title	Description
Release Notes	Late-breaking information about the software and the documentation. Includes a comprehensive, table-based summary of the supported hardware, operating system, Java Development Kit (JDK), and database drivers.
Quick Start Guide	How to get started with the Application Server product.
Installation Guide	Installing the software and its components.
Deployment Planning Guide	Evaluating your system needs and enterprise to ensure that you deploy the Application Server in a manner that best suits your site. General issues and concerns that you must be aware of when deploying the server are also discussed.
Application Deployment Guide	Deployment of applications and application components to the Application Server. Includes information about deployment descriptors.
Developer's Guide	Creating and implementing Java Platform, Enterprise Edition (Java EE platform) applications intended to run on the Application Server that follow the open Java standards model for Java EE components and APIs. Includes information about developer tools, security, debugging, and creating lifecycle modules.
Java EE 5 Tutorial	Using Java EE 5 platform technologies and APIs to develop Java EE applications.
WSIT Tutorial	Developing web applications using the Web Service Interoperability Technologies (WSIT). Describes how, when, and why to use the WSIT technologies and the features and options that each technology supports.
Administration Guide	System administration for the Application Server, including configuration, monitoring, security, resource management, and web services management.
High Availability Administration Guide	Post-installation configuration and administration instructions for the high-availability database.
Administration Reference	Editing the Application Server configuration file, `domain.xml`.
Upgrade and Migration Guide	Upgrading from an older version of Application Server or migrating Java EE applications from competitive application servers. This guide also describes differences between adjacent product releases and configuration options that can result in incompatibility with the product specifications.
Performance Tuning Guide	Tuning the Application Server to improve performance.
Troubleshooting Guide	Solving Application Server problems.
Error Message Reference	Solving Application Server error messages.
Reference Manual	Utility commands available with the Application Server; written in man page style. Includes the `asadmin` command line interface.

Typographic Conventions

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

TABLE P–3 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	A placeholder to be replaced 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 (note that some emphasized items appear bold online)	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file.

Symbol Conventions

The following table explains symbols that might be used in this book.

TABLE P–4 Symbol Conventions

Symbol	Description	Example	Meaning
`[ ]`	Contains optional arguments and command options.	`ls [-l]`	The `-l` option is not required.
`{ | }`	Contains a set of choices for a required command option.	`-d {y|n}`	The `-d` option requires that you use either the `y` argument or the `n` argument.
`${ }`	Indicates a variable reference.	`${com.sun.javaRoot}`	References the value of the `com.sun.javaRoot` variable.
-	Joins simultaneous multiple keystrokes.	Control-A	Press the Control key while you press the A key.
+	Joins consecutive multiple keystrokes.	Ctrl+A+N	Press the Control key, release it, and then press the subsequent keys.

TABLE P–4 Symbol Conventions *(Continued)*

Symbol	Description	Example	Meaning
→	Indicates menu item selection in a graphical user interface.	File → New → Templates	From the File menu, choose New. From the New submenu, choose Templates.

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/`)

Searching Sun Product Documentation

Besides searching Sun product documentation from the docs.sun.com℠ web site, you can use a search engine by typing the following syntax in the search field:

search-term `site:docs.sun.com`

For example, to search for "broker," type the following:

```
broker site:docs.sun.com
```

To include other Sun web sites in your search (for example, java.sun.com, www.sun.com, and developers.sun.com), use `sun.com` in place of `docs.sun.com` in the search field.

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.

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 Send Comments. In the online form, provide the full document title and part number. The part number is a 7-digit or 9-digit number that can be found on the book's title page or in the document's URL. For example, the part number of this book is 820-3753.

◆ ◆ ◆ CHAPTER 1

Introduction to the Mobile Client Business Object (MCBO) API

Sun Java System Mobile Enterprise Platform (MEP) provides a Java Platform, Mobile Edition (Java ME) data synchronization library, called the Mobile Client Business Object (MCBO) API , that enables development of client applications for MEP. This library, in conjunction with the Enterprise Connector Business Object (ECBO) API, provides a complete solution that allows you to synchronize arbitrary enterprise data. Although the MCBO and ECBO APIs are based on Open Mobile Alliance Data Synchronization (OMA DS), you do not need to know specifics of OMA DS in order to use the APIs.

You must be connected to a server in order to synchronize data with the server. However, you can use the MCBO library in disconnected mode; that is, you can add, delete, and modify client data without being connected to a server.

This chapter covers the following topics:

About the Mobile Client Business Object (MCBO) API

The Mobile Client Business Object (MCBO) API provides a simple set of APIs to build Java ME client applications that can synchronize business data with databases or ERP/EAI systems. To provide this synchronization capability, the APIs use an implementation of the Open Mobile Alliance Data Synchronization (OMA DS) specifications, formerly known as SyncML. Even if you have no knowledge of OMA DS, you can use the MCBO API to build Java ME client applications with OMA DS capabilities on Java ME devices. These client applications can synchronize their local data with a Sun Java System Mobile Enterprise Platform (MEP) server, which in turn communicates with a database or ERP/EAI system. The MCBO API can establish connections to any server that conforms to the OMA DS standard.

The MCBO API provides the ability to synchronize business objects in the form of arbitrary user-defined data types.

The MCBO API allows you to synchronize any objects that can be represented as a byte array, including arbitrary data types and data collections. Examples include:

- Databases
- Binary data, such as images
- Nodes belonging to hierarchical/tree data structures (for example, registry entries)

The MCBO API offers the following benefits:

- It provides a very simple framework for data synchronization of business objects
- It keeps the resulting application jar files small
- Client device memory requirements are low (the maximum size of the heap is kept low)

Synchronization Types

The MCBO API supports the following types of client-initiated synchronizations:

- Both from server to client and from client to server:
  - **Two-way sync (fast sync)**

    Two-way sync, also called fast sync, is the normal synchronization mode, in which the client and the server exchange modifications to the data that they have stored. An initial slow sync is used to populate the data on the client.

    For details, see "Two-way Sync (Fast Sync)" in *Sun Java System Mobile Enterprise Platform 1.0 Architectural Overview*.
  - **Slow sync**

    The slow sync is similar to two-way sync, except that all the items in the client databases are compared with all the items in the server databases, on a field-by-field basis. A slow sync can be requested if the client and server data is mismatched or if the client or server loses its information.

    For details, see "Slow Sync" in *Sun Java System Mobile Enterprise Platform 1.0 Architectural Overview*
- From client to server only:
  - **One-way sync from client**

    This is one half of a two-way sync. In this mode, the client sends modifications of its data store to the server. The server updates its data store appropriately but does not send modifications of its data store to the client.
  - **Refresh sync from client**

In this mode, the client exports all its data to the server. The server is expected to replace all its data with the data sent by the client.

Note – Use this synchronization type with caution.

- From server to client only:
  - **One-way sync from server**

    This is the other half of a two-way sync. In this mode, the server sends modifications of its data store to the client. The client updates its data store appropriately but does not send modifications of its data store to the server.
  - **Refresh sync from server**

    In this mode, the server exports all its data from a database to the client. The client is expected to replace all its data with the data sent by the server.

Server-initiated synchronization is also possible. The server can initiate syncs by sending SMS messages to the client device.

Both the client and the server store information about changes to their respective data stores since the last successful synchronization. When the next synchronization is performed, the client and server negotiate how the changes are resolved and propagated according to the type of synchronization being performed.

Client Device Requirements

The client device must be a Java and GPRS/UMTS enabled device that supports the following specifications:

- Mobile Information Device Profile (MIDP) 2.0

  MIDP is a specification published for the use of Java on embedded devices such as cell phones and PDAs. MIDP is part of the Java ME framework. MIDP 2.0 was developed under the Java Community Process as JSR-118.
- Either Connected Limited Device Configuration (CLDC) 1.1 or Connected Device Configuration (CDC) 1.1.2

  CLDC is a specification of a framework for Java ME applications targeted at devices with very limited resources, such as pagers and mobile phones. CLDC 1.1 was developed under the Java Community Process as JSR-139.

  CDC is a specification of a framework for Java ME applications targeted at devices with less limited resources, such as PDAs. CDC 1.1.2 was developed under the Java Community Process as JSR-218.
- JSR-75, PDA Optional Packages for the Java ME Platform, for accessing mobile device filesystems

The size of the obfuscated jar file for the MCBO API is approximately 190 KB (it can vary depending upon the size of the application), so the burden of supporting this API is low even on devices that have limited memory. Client applications are bundled with a copy of the API as a single jar file.

Server-side Requirements

On the server side, the MCBO API needs a standard-conforming OMA DS server. The MEP Gateway Engine installed on Application Server is such a server.

◆ ◆ ◆ CHAPTER 2

Building a Client Application

A Mobile Enterprise Platform (MEP) client application typically includes the following:

- A Java Platform, Micro Edition (Java ME) MIDlet that uses the Mobile Client Business Object (MCBO) API in addition to Java ME APIs
- An extension of the `com.sun.mep.client.api.BusinessObject` class

This chapter describes the basics of building a MEP client application. It contains the following sections:

This chapter does not explain how to develop a MIDlet, because this process varies depending upon what development tools you use.

Packages in the Mobile Client Business Object API

The Mobile Client Business Object (MCBO) API consists of the following Java classes:

- `com.sun.mep.client.api.BusinessObject`, which defines your data model and the serialized form used to store the data on the client device
- `com.sun.mep.client.api.BusinessObjectStorage`, which manages the storage and retrieval of `BusinessObject` instances on the client device
- `com.sun.mep.client.api.DefaultSecurityManager`, which provides a basic implementation of `com.sun.mep.client.api.SecurityManager`
- `com.sun.mep.client.api.SecurityManager`, which manages all of the client-side security features

- `com.sun.mep.client.api.SyncManager`, which controls synchronization with the MEP Gateway Engine
- `com.sun.mep.client.api.SyncResults`, which provides coarse-grained statistics after synchronizations
- `com.sun.mep.client.api.SMSMessageHandler`, which is a callback handler for SMS push notification messages sent from the Gateway Engine
- `com.sun.mep.client.api.SyncType`, which enumerates the six synchronization types
- `com.sun.mep.client.api.EncodingType`, which enumerates two encoding types
- `com.sun.mep.client.api.SyncException`, which provides exception-handling methods
- `com.sun.mep.client.api.SMSMessageHandler`, a callback handler for SMS push messages sent from the Gateway Engine
- `com.sun.mep.client.api.SMSInstructionType`, which enumerates SMS push instruction types

See Chapter 4, "Classes and Methods in the Mobile Client Business Object API Package," for summaries of the classes, fields, and methods in these packages. The API documentation is also included in the MEP client bundle. In the directory where you unzipped the client bundle (see the *Sun Java System Mobile Enterprise Platform 1.0 Installation Guide* for details), it is in the directory `sjsmep-client-1_0_02-fcs/doc/mcbo/api`.

The MCBO API packages provide a simple interface on top of a set of more complex packages, the `com.synchronica` APIs. At times an application may find it useful to call some of these APIs.

This chapter uses the Secure MusicDB sample application provided with MEP to demonstrate how to use the MCBO API. The client in this application communicates with an Enterprise Connector deployed in the Gateway Engine, which in turn communicates with a database using the Java Database Connectivity (JDBC) API.

The source code for the Secure MusicDB sample application is included in the MEP client bundle. In the directory where you unzipped the client bundle, it is in the subdirectory `sjsmep-client-1_0_02-fcs/samples/mcbo/secure-musicdb/`.

Use of security features in a MEP application is recommended, but it is not required. If you implement security, you can provide your own implementation of `com.sun.mep.client.api.SecurityManager` to replace `com.sun.mep.client.api.DefaultSecurityManager`.

Extending the BusinessObject Class

To create a client application using the MCBO API, you need to create your own class that extends the `com.sun.mep.client.api.BusinessObject` class.

Typically, you begin by importing the packages the class needs. In the Secure MusicDB sample code, the `Album` class, defined in the file `Album.java`, begins by importing the following packages:

```
import com.sun.mep.client.api.BusinessObject;
import com.synchronica.commons.date.DateStringParser;
import com.synchronica.commons.date.Iso8601Converter;
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.util.Date;
import java.util.TimeZone;
```

You must implement bean properties for your data model. The only required getter and setter methods are `setName` and `getName`, which specify and retrieve the name of the object, and `getExtension`, which returns the file extension for your object.

In addition, you must implement the `serialize` and `deserialize` methods.

The `Album` class extends the `BusinessObject` class:

```
public class Album extends BusinessObject {
```

The code first declares a string constant and the bean properties:

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

String albumName;
/**
 * Album's artist.
 */
String artist;
/**
 * Date in which the album was published.
 */
Date datePublished;
/**
 * Album's rating from 1 to 5.
 */
int rating;
```

The Album class has two constructor methods, a no-argument version and one that takes the name of the album as an argument:

```
public Album() {
    super();
}

public Album(String name) {
    super(name);
}
```

The Album class does not implement its own versions of getName and setName, instead inheriting the versions in BusinessObject. It implements getExtension by specifying the suffix .alb as the file extension for Album objects:

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

In addition, the class implements getter and setter methods for the String property artist, the java.util.Date property datePublished, and the int property rating:

```
public String getArtist() {
    return artist;
}
public Date getDatePublished() {
    return datePublished;
}
public int getRating() {
    return rating;
}
public void setArtist(String artist) {
    this.artist = artist;
}
public void setDatePublished(Date datePublished) {
    this.datePublished = datePublished;
}
public void setRating(int rating) {
    this.rating = rating;
}
```

The Album class implements the serialize method by creating a java.io.DataOutputStream from a java.io.ByteArrayOutputStream, writing the album data to the java.io.DataOutputStream, then returning the java.io.ByteArrayOutputStream converted to a ByteArray.

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

```
    DataOutputStream dOut = new DataOutputStream(out);

    dOut.writeUTF(getName());
    dOut.writeUTF(artist != null ? artist : DEFAULT_VALUE);
    dOut.writeUTF(
        datePublished != null ? getSimplifiedDate(datePublished) : DEFAULT_VALUE);
    dOut.writeUTF(Integer.toString(rating));
    dOut.flush();

    System.err.println("Serializing album:");
    System.err.println("    Name: "  + getName());
    System.err.println("  Artist: "  + artist != null ? artist : DEFAULT_VALUE);
    System.err.println("    Date: "  +
        datePublished != null ? getSimplifiedDate(datePublished) : DEFAULT_VALUE);
    System.err.println("  Rating: "  + Integer.toString(rating));

    return out.toByteArray();
}
```

The class implements the `deserialize` method by creating a `java.io.DataInputStream` from a `java.io.ByteArrayInputStream` passed as an argument, then reading the album data from the `java.io.DataInputStream`. It uses some utility methods from the `com.synchronica` API to handle date information.

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

    albumName = dIn.readUTF();
    artist = dIn.readUTF();
    if (artist.equals(DEFAULT_VALUE)) {
        artist = null;
    }

    DateStringParser dateParser = new Iso8601Converter();
    String date = dIn.readUTF();
    Calendar c = dateParser.stringToCalendar(date, TimeZone.getDefault());
    datePublished = date.equals(DEFAULT_VALUE) ? null : c.getTime();

    rating = Integer.parseInt(dIn.readUTF());
}
```

The `Album` class also contains a utility method, `getSimplifiedDate`, that converts the `java.util.Date` value to a `String`.

Using the Mobile Client Business Object API in a Java ME Application

A client application on a mobile device consists primarily of a Java ME MIDlet. The MIDlet implements the Graphical User Interface (GUI) for the application and also calls Mobile Client Business Object (MCBO) API methods to synchronize the `BusinessObject` data.

This section describes how to use the MCBO API in a MIDlet. It assumes that you are already familiar with the Java ME API. It does not describe how to create a Graphical User Interface (GUI) to display business objects and allow users to create or modify them. A good tool for creating a MIDlet is the NetBeans IDE with the Mobility Pack; for details, visit the NetBeans web site (`http://www.netbeans.org/`).

In the Secure MusicDB sample code, the `SecureJdbcMIDlet.java` file contains Java ME code and MCBO API code. The Java ME code creates the user interface that allows users to create, edit, and delete business objects. The MCBO API code stores and retrieves business object data on the client device and synchronizes client-side modifications with the data on the back end.

Typically, a MIDlet begins by importing the MCBO API package in addition to the Java ME packages:

```
import com.sun.mep.client.api.*;
```

A MIDlet uses the API to perform the following tasks:

Creating `DefaultSecurityManager`, `SyncManager` and `BusinessObjectStorage` Objects

The first task for the MEP client code is to create the objects needed for synchronization and data manipulation:

- A `SyncManager` object
- A `BusinessObjectStorage` object
- Optionally, a `DefaultSecurityManager` object or another extension of the `SecurityManager` class

You may also want to enable logging for debugging purposes by calling the `SyncManager.enableLogging` method. If logging is enabled, logging messages for the client code are written both to standard output and to a file on the device named `meplog.txt`.

You commonly perform these operations within a thread, as follows:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        securityMgr = new DefaultSecurityManager("musicdb");
        securityMgr.setMaxValidationAttempts(3);
        syncMgr = new SyncManager(".alb", securityMgr);
        syncMgr.enableLogging(true);
        boStorage = syncMgr.getBusinessObjectStorage();
    }
});
t.run();
```

This code first instantiates a security manager and sets the maximum allowed number of validation attempts. If this maximum is exceeded, all MEP records on the device are erased. See "Data Destruction" on page 40 for details.

The code then uses the form of the `SyncManager` constructor that takes two arguments, the file extension used for the business object and the security manager. In this case, the extension is the string `".alb"`, as specified by the `getExtension` method of the `Album` class. The code also turns on logging.

Once you enable security by specifying an implementation of `SecurityManager` in the `SyncManager` constructor, all of the data stored locally on the device will be encrypted and decrypted automatically. There are no further requirements on the client application to explicitly perform encryption or decryption of the data.

The code then calls the `SyncManager.getBusinessObjectStorage` factory method to instantiate the `BusinessObjectStorage` object. This object provides storage for `Album` objects on the mobile device's file system.

Establishing Login Credentials

To provide application-level authentication, a secure client application must use the security manager to create login credentials for the user. The MIDlet code provides an initial login screen that requires the user to create both a secret and a Personal Identification Number (PIN). Users do not need to remember the secret, but they must remember the PIN.

The MIDlet code calls the security manager's `computeKey` and `setKey` methods to create a key from the PIN entered by the user. It then calls the security manager's `storeCredentials` method to create credentials based on the secret.

```
byte[] key = securityMgr.computeKey(getInitialPinField().getString());
securityMgr.setKey(key);
securityMgr.storeCredentials(getSecretField().getString());
```

The `getInitialPinField` and `getSecretField` methods are UI methods that obtain the needed string values.

The secret and PIN provide security in addition to the username and password credentials required by the Gateway Engine in order to perform synchronization (as described in "Setting User Credentials" on page 23).

Working with Business Objects on the File System

The MIDlet code typically allows users to create new objects and to edit or delete existing objects in disconnected mode, using the mobile device's file system without being connected to a server. The code commonly uses a combination of `BusinessObject` and `BusinessObjectStorage` methods to perform the following tasks:

- "Retrieving Objects for Editing" on page 22
- "Deleting Objects" on page 22
- "Saving Objects" on page 23

Typically, a user on a client device performs a number of operations on the client device in disconnected mode, then logs in to the server and synchronizes the data.

Retrieving Objects for Editing

To allow users to view existing objects, the code commonly displays a list of names returned by the `BusinessObjectStorage.listBusinessObjectNames` method. This method retrieves a list of the names of all the business objects that have the file extension specified by the `SyncManager` constructor method. For example, the `SecureJdbcMIDlet` code calls the following method before populating a form with a list of albums:

```
Vector v = boStorage.listBusinessObjectNames();
```

To display a selected album, the `SecureJdbcMIDlet` code instantiates an `Album` object, using a name argument that represents the filename stripped of its `".alb"` extension. The code then calls the `BusinessObjectStorage.readBusinessObject` method to read the data for the selected album from the file system into the `Album` object.

```
Album a = new Album(selectedAlbum.substring(0, selectedAlbum.length()-4));
boStorage.readBusinessObject(a);
```

The `SecureJdbcMIDlet` code then calls the getter methods for `Album` objects to retrieve the selected album's property values and display them in a form for editing.

Deleting Objects

To allow users to delete a selected album, the `SecureJdbcMIDlet` code calls the `BusinessObjectStorage.deleteBusinessObject` method with a `String` argument, the name of the album:

```
boStorage.deleteBusinessObject(selectedAlbum);
```

Saving Objects

To save a newly created or edited album, the `SecureJdbcMIDlet` code calls its `saveAlbum` method. This method instantiates an `Album` object and then calls the methods that set the album's properties, using Java ME GUI code to retrieve the values. Finally, the `saveAlbum` method calls the `BusinessObjectStorage.writeBusinessObject` method to save the album to the file system:

```
Album a = new Album();
...
boStorage.writeBusinessObject(a);
```

Synchronizing Data with the Server

Once users have created or modified objects on the client using `BusinessObjectStorage` methods, they can use `SyncManager` methods to synchronize the modified data with the server. Synchronization includes the following tasks:

- "Setting User Credentials" on page 23
- "Performing Synchronization" on page 23
- "Retrieving Synchronization Results" on page 24

Setting User Credentials

The Gateway Engine requires username/password authentication for secure access. Before performing a synchronization, the MIDlet must call the `SyncManager.setCredentials` method, which takes three arguments: the username, the password, and the HTTP/S URL of the Gateway Engine. In `SecureJdbcMIDlet.java`, the arguments are supplied by three GUI methods, as follows:

```
syncMgr.setCredentials(
        getUserName().getString(),
        getPassword().getString(),
        getSyncServer().getString());
```

These methods obtain input from the user and return `TextField` values.

The initial creation of users is a MEP administrative task, described in the *Sun Java System Mobile Enterprise Platform 1.0 Administration Guide*

Performing Synchronization

Once user credentials are established, synchronization can take place. The `SecureJdbcMIDlet` code calls the `SyncManager.sync` method, which takes a `SyncType` argument. In this case, the code calls a method that returns a `SyncType` value:

```
syncMgr.sync(getSelectedSyncType());
```

The `getSelectedSyncType` method in turn uses the value returned by a GUI method, `getSyncType`.

Retrieving Synchronization Results

After a successful synchronization, you can retrieve and display information about the synchronization results. The `SecureJdbcMIDlet` code retrieves the results using the `SyncManager.getSyncResults` method, which returns a `SyncResults` value:

```
SyncResults results = syncMgr.getSyncResults();
```

It then displays the results in a GUI form by calling `SyncResults` methods.

Developing Client Applications for the BlackBerry Using NetBeans IDE

This guide cannot describe how to develop client applications for every possible device using every possible development tool. This section, however, describes how to develop a client application for one of the most commonly used devices, the BlackBerry, using one of the most commonly used development tools, NetBeans IDE. It contains the following sections:

Prerequisites

Before you can develop a client application for the Blackberry using NetBeans IDE, you must install the following software on a Microsoft Windows system:

- The Java Development Kit (JDK), version 5 or 6. Set your `JAVA_HOME` and `PATH` environment variables to point to your installation of JDK 5 or JDK 6.
- BlackBerry Java Development Environment (BlackBerry JDE) v4.2.1

  Go to `http://na.blackberry.com/eng/developers/javaappdev/javadevenv.jsp`, then download and install BlackBerry JDE v4.2.1. This is the only version that is compatible with current versions of NetBeans IDE. BlackBerry JDE runs only on Windows systems.
- BlackBerry Email and MDS Services Simulator Package v4.1.4

Go to
http://na.blackberry.com/eng/developers/browserdev/devtoolsdownloads.jsp,
then download and install the BlackBerry Email and MDS Services Simulator Package v4.1.4.

- NetBeans IDE 6.1

  Go to http://www.netbeans.org/, then download and install NetBeans IDE 6.1. BlackBerry client application development has been thoroughly tested only with this version of NetBeans IDE.

  On the NetBeans download page, select All as the version to download. When you install NetBeans IDE, click Customize to install only some of the components. Select the following:

  - Base IDE
  - Java SE
  - Web and Java EE
  - Mobility

  Do not select a runtime, since you are using the MEP version of Application Server.

Note – You must have a BlackBerry Developer Community account in order to download the BlackBerry software. If you do not have an account, follow the instructions on the website to obtain one.

▼ To Configure BlackBerry JDE v4.2.1

1. **Click Start→All Programs→Research In Motion→BlackBerry JDE 4.2.1→JDE.**
2. **Click Edit→Preferences.**
3. **Click the Simulator tab and perform these steps:**
   a. **Select the 8800-JDE Profile from the pull-down menu.**
   b. **Select the Launch simulator checkbox.**
   c. **Select the Launch Mobile Data Service (MDS) with Simulator checkbox.**
4. **Click the MDS Simulator Tab. Make sure that the MDS Simulator directory location is pointing to the v4.1.4 MDS directory you installed. For example:**

```
C:\Program Files\Research In Motion\BlackBerry Email and MDS Services Simulators 4.1.4\MDS
```

5. **Click OK.**

   You can leave the JDE running, because you may need it later on.

▼ To Configure NetBeans IDE for BlackBerry Application Development

1 Start a text editor and copy the following text into an empty file.

Note – If you installed BlackBerry JDE in a non-default location (for example, not on the `C:\` drive), edit the contents of the `home` property setting for the `platform` element.

Make sure that the contents of the `preverifycmd` property setting for the platform element all appear on one line of the file. The contents are broken up here for readability only.

```
<?xml version='1.0'?>
<!DOCTYPE platform PUBLIC '-//NetBeans//DTD J2ME PlatformDefinition 1.0//EN'
          'http://www.netbeans.org/dtds/j2me-platformdefinition-1_0.dtd'>
<platform name="BlackBerry_JDE_421"
          home="C:\Program Files\Research In Motion\BlackBerry JDE 4.2.1"
          type="CUSTOM"
          displayname="BlackBerry JDE 421"
          srcpath=""
          docpath="${platform.home}/docs/api,"
          preverifycmd="&quot;{platformhome}{/}bin{/}preverify&quot;
                        {classpath|-classpath &quot;{classpath}&quot;}
                        -d &quot;{destdir}&quot; &quot;{srcdir}&quot;"
          runcmd="cmd /C &quot;cd /D {platformhome}{/}simulator&amp;{device}&quot;"
          debugcmd="cmd /C &quot;cd /D {platformhome}{/}bin&amp;jdwp&quot;">
    <device name="8800" description="8800">
        <optional name="JSR75" version="1.0"
                  displayname="File Connection and PIM Optional Packages"
                  classpath="${platform.home}/lib/net_rim_api.jar"
                  dependencies="" default="true"/>
        <optional name="MMAPI" version="1.0"
                  displayname="Mobile Media API"
                  classpath="${platform.home}/lib/net_rim_api.jar"
                  dependencies="" default="true"/>
        <configuration name="CLDC" version="1.1"
                       displayname="Connected Limited Device Configuration"
                       classpath="${platform.home}/lib/net_rim_api.jar"
                       dependencies="" default="true"/>
        <optional name="OBEX" version="1.0"
                  displayname="Object Exchange APIs"
                  classpath="${platform.home}/lib/net_rim_api.jar"
                  dependencies="" default="true"/>
        <optional name="JSR82" version="1.0"
                  displayname="Java APIs for Bluetooth Wireless Technology"
                  classpath="${platform.home}/lib/net_rim_api.jar"
```

```
                dependencies="" default="true"/>
        <optional name="WMA" version="1.1"
                displayname="Wireless Messaging API"
                classpath="${platform.home}/lib/net_rim_api.jar"
                dependencies="" default="true"/>
        <optional name="JSR179" version="1.0"
                displayname="Location Based APIs"
                classpath="${platform.home}/lib/net_rim_api.jar"
                dependencies="" default="true"/>
        <optional name="JSR177" version="1.0"
                displayname="Security and Trust Services APIs"
                classpath="${platform.home}/lib/net_rim_api.jar"
                dependencies="" default="true"/>
        <profile name="MIDP" version="2.0"
                displayname="Mobile Information Device Profile"
                classpath="${platform.home}/lib/net_rim_api.jar"
                dependencies="" default="true"/>
    </device>
</platform>
```

2. **Save the file, giving it the name** `BlackBerry_JDE_421.xml`**.**

3. **Copy the file to the following location in your home directory under** `C:\Documents and Settings`**:**

   `.netbeans\6.1\config\Services\Platforms\org-netbeans-api-java-Platform`

4. **If NetBeans IDE is running, stop it.**

   You will be prompted to start (or restart) NetBeans IDE in the next task, "To Import the SecureMusicDB Sources into NetBeans IDE as a BlackBerry Project" on page 27.

Next Steps After you start NetBeans IDE, The Blackberry JDE will appear in the list of platforms when you choose Java Platforms from the Tools menu.

▼ To Import the SecureMusicDB Sources into NetBeans IDE as a BlackBerry Project

To build and run a SecureMusicDB project for the BlackBerry from sources in NetBeans IDE, follow these steps.

1. **To obtain the MEP client library bundle, go to the following URL:** `http://www.sun.com/software/products/mep/get.jsp`**.**

2. **Click Download, provide the requested information, then click Log In and Continue.**

3. **Download the `sjsmep-client-1_0_02-fcs.zip` bundle.**

4. **Unzip the bundle in a location of your choosing (for example, under `C:\`).**

5. **Start NetBeans IDE.**
   The first time you start NetBeans IDE, you are prompted to install some updates. Install them.

6. **In NetBeans IDE, follow these steps to create a Mobility Project and import the `secure-musicdb` sources.**

   a. **From the File menu, select New Project.**
      The Choose Project screen appears.

   b. **Click Mobility, then click Mobile Project from Existing MIDP Sources.**

   c. **Click Next.**
      The Specify MIDP Sources Screen appears.

   d. **In the Sources Location field, specify the location of the `secure-musicdb` sources in the unzipped bundle. For example, if you unzipped the bundle to the `C:\` directory, specify the following:**
      ```
      C:\sjsmep-client-1_0_02-fcs\samples\mcbo\secure-musicdb\src
      ```

   e. **Click Next.**
      The Name and Location Screen appears.

   f. **Type a name for the Project or keep the default name.**

   g. **Click Next.**
      The Default Platform Selection Screen appears.

   h. **Set the Emulator Platform to "BlackBerry JDE 421" and verify that the Device is 8800.**

   i. **Click Finish.**
      The project appears in the Projects pane.

7. **To add the file `mep_client_api.jar` to the supported Libraries & Resources, follow these steps.**

   a. **Right-click the project and select Properties.**

   b. **Click Libraries & Resources.**

   c. **Click Add Jar/Zip.**

d. **Browse to the location of the unzipped bundle above the `lib` directory to add `mep_client_api.jar`.**

   For example, if you unzipped the bundle to the `C:\` directory, the file name would be `C:\sjsmep-client-1_0_02-fcs\lib\BlackBerry\mep_client_api.jar`.

e. **Click OK.**

f. **Click the Files tab (next to the Projects tab) and open the `project.properties` file under the `nbproject` directory.**

g. **Edit the `file.reference.mep_client_api.jar` property to contain the fully qualified path name of the `mep_client_api.jar` file.**

   For a BlackBerry project, the pathname must be absolute, not relative.

   For example, if you unzipped the bundle to the `C:\` directory, edit the property definition to look like this:

```
file.reference.mep_client_api.jar=C:/sjsmep-client-1_0_02-fcs/lib/BlackBerry/mep_client_api.jar
```

   Use forward slashes (`/`) instead of the usual Windows file separator.

8 **Click the Files tab and open the project's `build.xml` file.**

9 **Add the following code fragment immediately before the `</project>` tag at the end of the file:**

```
<target name="do-preprocess">
    <fail unless="libs.ant-contrib.classpath">
 Classpath to Ant Contrib library (libs.ant-contrib.classpath property) is not set.
    </fail>
    <taskdef resource="net/sf/antcontrib/antlib.xml">
        <classpath>
            <pathelement path="${libs.ant-contrib.classpath}"/>
        </classpath>
    </taskdef>
    <available file="${platform.home}/bin/rapc.exe" property="do.rapc"/>
    <if>
        <isset property="do.rapc"/>
        <then>
            <property name="jpda.port" value="8000"/>
            <path id="antlib.classpath">
                <fileset dir="${user.dir}/mobility8/modules/ext/"
                         includes="ant-contrib-1.0b3.jar"/>
            </path>
            <mkdir dir="${dist.dir}"/>
            <path id="src-files">
                <fileset dir="${src.dir}" includes="**/*.*"/>
            </path>
            <property name="srcs" value="${toString:src-files}"/>
```

```
                <for list="${srcs}" param="file" delimiter=";" trim="true">
                    <sequential>
                        <echo message="@{file}${line.separator}"
                              file="${src.dir}/${name}_build.files" append="true"/>
                    </sequential>
                </for>
                <touch file="${dist.dir}/${dist.jar}"/>
                <nb-overrideproperty name="buildsystem.baton"
                                     value="${preprocessed.dir}"/>
            </then>
            <else>
                <nb-overrideproperty name="buildsystem.baton" value="${src.dir}"/>
                <antcall target="${name}-impl.do-preprocess"/>
            </else>
        </if>
    </target>
    <target name="do-compile">
        <if>
            <isset property="do.rapc"/>
            <then>
                <antcall target="create-jad"/>
                <antcall target="update-jad"/>
                <copy file="${dist.dir}/${dist.jad}" toDir="${src.dir}"/>
                <exec dir="${src.dir}"
                      executable="${platform.home}/bin/rapc.exe" failonerror="true">
                    <arg value="-quiet"/>
                    <arg value="import=${platform.bootclasspath};${libs.classpath}"/>
                    <arg value="codename=${name}"/>
                    <arg value="-midlet"/>
                    <arg value="jad=${dist.jad}"/>
                    <arg value="@${name}_build.files"/>
                </exec>
                <delete file="${basedir}/${src.dir}/${name}_build.files"/>
                <copy file="${name}.alx" todir="${dist.dir}"/>
                <nb-overrideproperty name="buildsystem.baton"
                                     value="${build.classes.dir}"/>
            </then>
            <else>
                <nb-overrideproperty name="buildsystem.baton"
                                     value="${preprocessed.dir}"/>
                <antcall target="${name}-impl.do-compile"/>
            </else>
        </if>
    </target>
    <target name="pre-obfuscate">
        <nb-overrideproperty name="buildsystem.baton" value="${build.classes.dir}"/>
    </target>
    <target name="post-jar" if="do.rapc">
```

```
        <move todir="${dist.dir}">
            <fileset dir="${src.dir}">
                <include name="**/${name}*.*"/>
            </fileset>
        </move>
        <copy todir="${platform.home}/simulator" verbose="true">
            <fileset dir="${dist.dir}">
                <include name="**/${name}*.*"/>
            </fileset>
        </copy>
    </target>
    <target name="post-clean">
        <delete failonerror="false" includeemptydirs="true">
            <fileset dir="${platform.home}/simulator">
                <include name="**/${name}*.*"/>
            </fileset>
            <fileset dir="${dist.dir}">
                <include name="**/*.*"/>
            </fileset>
            <fileset dir="${src.dir}">
                <include name="**/${name}*.*"/>
            </fileset>
        </delete>
    </target>
```

10 Create an `.alx` file for this project.

a. Click the Files tab.

b. Right-click your project and select New→Other.

c. In the Choose File Type screen, click Other, then click Empty File.

d. Click Next.

e. In the Name and Location screen, give the file the same name as your project, with the extension `.alx`.

For example, if `bb-secure-musicdb` is the project name, name the file `bb-secure-musicdb.alx`.

f. Click Finish.

The empty file opens.

g. Copy and paste the following text into the file, replacing `myProject` with your project name, and including any vendor and copyright information needed for your application.

```
<loader version="1.0">
    <application id="myProject">
        <name>
            myProject
        </name>
        <description/>
        <version>
            1.0
        </version>
        <vendor>
        </vendor>
        <copyright>
        </copyright>
        <fileset Java="1.3">
            <directory/>
            <files>
                myProject.cod
            </files>
        </fileset>
        <application id="mep_client_api">
            <name/>
            <description/>
            <version>
                1.0
            </version>
            <vendor>
                Sun Microsystems Inc.
            </vendor>
            <copyright>
                Copyright (c) 2008 Sun Microsystems Inc.
            </copyright>
            <fileset Java="1.3">
                <directory/>
                <files>
                    mep_client_api.cod
                </files>
            </fileset>
        </application>
    </application>
</loader>
```

h. Save and close the file.

11 **Copy the file** `mep_client_api.cod` **from the directory** `C:\sjsmep-client-1_0_02-fcs\lib\BlackBerry` **to the** `simulator` **directory of the BlackBerry JDE (for example,** `C:\Program Files\Research In Motion\BlackBerry JDE 4.2.1\simulator`**).**

12 **Click the Projects tab, then right-click your** `secure-musicdb` **project and select Clean & Build.**

13 **Right-click your** `secure-musicdb` **project and select Run.**

The BlackBerry Device Simulator appears.

Note – When you select Run, NetBeans IDE automatically loads the application on the Simulator using the `.jad` and `.jar` files (not the `.cod` file). To load the application from the `.cod` file created, use the Load Java Program option in the Simulator.

---

14 **Launch the** `MepSecureJdbcMIDlet` **application and perform a Sync.**

Note – The MDS must be running for the client to perform syncs. If you started the JDE, MDS should get launched automatically. Otherwise, start MDS manually as follows: From the Windows Start menu, choose All Programs→Research in Motion→BlackBerry Email and MDS Services Simulators 4.1.4→MDS.

Next Steps To remove the application from the Simulator, delete the `.jad`, `.jar`, and `.cod` files from the Simulator directory within the JDE and execute the three erase options in the JDE under File→Erase Simulator File.

▼ To Create a New BlackBerry Project to Use the MCBO API

To create a new NetBeans IDE project that uses the MCBO API, follow these steps.

1 **In NetBeans IDE, choose New Project from the File menu.**

2 **Choose Project Screen.**

3 **Click Mobility→MIDP Application.**

4 **Click Next.**

5 **In the Name and Location screen:**

a. **Type a name for the project or keep the default name.**

b. **Select Set as Main Project.**

c. **Select Create Hello MIDlet.**

d. **Click Next.**

6 **In the Default Platform Selection Screen:**

a. **Specify BlackBerryJDE421 as the Emulator Platform.**

b. **Specify 8800 as the Device.**

c. **Click Finish.**

7 **Add the** `mep_client_api.jar` **file to your Libraries & Resources for this project in order to call and have access to the MCBO API.**

a. **If you have not done so before, go to** `http://www.sun.com/software/products/mep/get.jsp` **and download the** `sjsmep-client-1_0_02-fcs.zip` **bundle.**

b. **Unzip the** `sjsmep-client-1_0_02-fcs.zip` **bundle (for example, under** `C:\`**).**

c. **In NetBeans IDE, right-click Project→Properties.**

d. **Click Libraries & Resources.**

e. **Click Add Jar/Zip.**

f. **Browse to the location of the unzipped bundle above the** `lib` **directory to add** `mep_client_api.jar`**.**

For example, if you unzipped the bundle to the `C:\` directory, the file name would be `C:\sjsmep-client-1_0_02-fcs\lib\BlackBerry\mep_client_api.jar`.

g. **Click the Files Tab (next to the Projects tab) and open the** `project.properties` **file under the** `nbproject` **directory.**

h. **Edit the** `file.reference.mep_client_api.jar` **property to contain the fully qualified path name of the** `mep_client_api.jar` **file.**

For a BlackBerry project, the pathname must be absolute, not relative.

For example, if you unzipped the bundle to the `C:\` directory, edit the property setting to look like this:

```
file.reference.mep_client_api.jar=C:/sjsmep-client-1_0_02-fcs/lib/BlackBerry/mep_client_api.jar
```

Use forward slashes (/) instead of the usual Windows file separator.

i. **Click the Files Tab and open the project's `build.xml` file. Immediately before the `</project>` tag at the end of the file, add the same code fragment you added in Step 9 of "To Import the SecureMusicDB Sources into NetBeans IDE as a BlackBerry Project" on page 27.**

8 **Create an `.alx` file for this project:**

a. **Click the Files tab.**

b. **Right-click your project and select New→Other.**

c. **In the Choose File Type screen, click Other, then click Empty File.**

d. **Click Next.**

e. **Give the file the same name as your project name, with the `.alx` extension.**

f. **Click Finish.**

g. **Copy and paste into the file the content from Step g under Step 10 of "To Import the SecureMusicDB Sources into NetBeans IDE as a BlackBerry Project" on page 27, replacing `myProject` with your project name.**

h. **Save and close the file.**

9 **Copy the file `mep_client_api.cod` from the directory `C:\sjsmep-client-1_0_02-fcs\lib\BlackBerry` to the `simulator` directory of the BlackBerry JDE (for example, `C:\Program Files\Research In Motion\BlackBerry JDE 4.2.1\simulator`).**

10 **Click the Projects tab, then right-click your project and select Clean & Build.**

11 **Right-click your project and select Run.**

The BlackBerry Device Simulator appears.

Note – When you select Run, NetBeans IDE automatically loads the application on the Simulator using the `.jad` and `.jar` files (not the `.cod` file). To load the application from the `.cod` file created, use the Load Java Program option in the Simulator.

---

12 **Launch the MIDlet application.**

Next Steps At this point, you have a boilerplate application that says Hello World. You can now add code to implement and call the MCBO API classes and methods, and you can edit the MIDlet code to provide a user interface and to perform synchronizations.

◆ ◆ ◆ CHAPTER 3

Client Security Architecture

This chapter contains an overview of the Sun Java System Mobile Enterprise Platform (MEP) client security features and describes how the Secure MusicDB application implements these features.

This chapter covers the following topics:

Client security must perform the following tasks:

- Provide a simple PIN-based form of authentication (see "Authentication on the Client Device" on page 38)
- Provide a means to secure data at rest on the mobile device (see "Data Encryption" on page 39)
- Provide a means to securely synchronize with the Gateway Engine on the server (see "Transport-layer Security" on page 40)
- Provide a mechanism to destroy business data (see "Data Destruction" on page 40)
- Provide a means to prevent the client device from synchronizing (see "Lockout" on page 40)
- Provide a means to remotely destroy all of the data on the device (see "Poison Pill" on page 40)

- Provide a means to notify the application that a certain quiet period has elapsed (see "Data Fading" on page 40)
- Provide an API that allows developers to replace the MEP default implementation with their own (see "The `DefaultSecurityManager` Class" on page 45 and "The `SecurityManager` Class" on page 47)

Client Security Limitations

This is not a perfectly secure system, so some best-practice recommendations are needed. Since MEP provides a library, it is not possible to control how sensitive information (that is, credentials) is managed. The application programmer is responsible for managing this information.

Best Practices for Secure Client Applications

Developers of secure client applications should observe the following rules:

- Do not hard-code values for the Gateway credentials into the application
- Do not store or cache form data on the device
- Require or encourage end users to use the native security services of the device
- You must use HTTPS to provide transport-layer security

Authentication on the Client Device

There are two forms of authentication on the client device:

- **User Authentication**: the end user authenticates with the device through an alphanumeric Personal Identification Number (PIN)
- **Gateway Authentication**: the end user authenticates with the Gateway Engine through a username and password

The MEP client library provides an API to validate an arbitrary length alphanumeric PIN against a PIN derivative stored on the device. The library also maintains a count of validation attempts (even across restarts of the application). If a threshold of failed attempts is exceeded (specified by the client application), data destruction and device lockout can occur.

Storing the PIN derivative and a count of validation attempts on the device is an obvious weakness in the security architecture, as this data could be easily subverted. Therefore, it is recommended that users follow the best practices outlined above to improve the overall security of the system.

Supplying the correct PIN allows users to access the application and perform local operations, but users will not be able to synchronize with the Gateway Engine unless they supply the proper username/password credentials for the Gateway Engine.

Authentication Implementation

Let:

```
S = alphanumeric secret (random key sequence entered exactly once by user)
S' = md5sum( S )

P = alphanumeric PIN (entered by user every time)
P' = md5sum( P )

cipherText = encrypt( S, P' )

persist { S', cipherText } on the device
```

Upon subsequent logins:

```
P = PIN
P' = md5sum( P )
plainText = decrypt( cipherText, P' )

if ( md5sum( plainText ) == S' )
   success
else
   failure
```

Data Encryption

Data at rest on the mobile device is encrypted by using a digest of the PIN as the encryption key. There are four locations in the MEP client library where encryption and decryption must occur. In these locations, the MEP library will invoke encrypt/decrypt callback methods that perform the tasks.

Transport-layer Security

It is assumed that transport-layer security is provided by HTTPS. Data streaming in the OMA DS protocol is always encoded simply in base64.

Data Destruction

The MEP client library keeps track of how many times client applications attempt to validate a PIN against the PIN derivative stored on the device (even across application restarts). If the application exceeds the threshold specified by the application developer, the MEP library will erase all of the MEP records on the mobile device and prevent any further attempts to validate the PIN.

Lockout

The MEP client library keeps track of how much time has lapsed since the last synchronization attempt with the Gateway Engine. At the beginning of each synchronization, the MEP library calculates how much time has elapsed. If the time since the last synchronization exceeds the threshold specified by the application developer, then all MEP records can be erased from the device.

The library also maintains a count of validation attempts (even across restarts of the application). If a threshold of failed attempts is exceeded, both data destruction and lockout can occur.

Poison Pill

A MEP administrator can remotely trigger the destruction (wiping) of all the data on a particular device.

Data Fading

The client security implementation keeps track of how much time has elapsed since the last successful synchronization. The client application may specify a maximum quiet period after which the application may decide to activate the data destruction feature.

Secure MusicDB Java ME Application

The Secure MusicDB application demonstrates most of the security features described in this document:

- **Authentication on the Client Device:** The first time you launch the MIDlet, you are prompted to set a security PIN. The PIN can be any alphanumeric string. You are also asked to enter a long random sequence of key-presses on the device in the "secret" field. The PIN and secret are used to compute the derivatives described above, which are stored on the mobile device's RMS record store. Upon subsequent launches of the MIDlet, you are prompted to enter the PIN. If the PIN does not correctly reverse the computation of the derivatives stored in RMS, then an error message appears, and you are prompted to enter the PIN again. The MIDlet also clearly indicates how many attempts you have left before it performs data destruction.
- **Data Destruction and Lockout:** After you fail to enter the PIN 3 times, the MIDlet destroys all MusicDB data on the device, and you are prevented from using the application.
- **Recovering from Lockout:** If you are locked out of the application, you must remove and reinstall the Secure MusicDB application on your device. This should reset the security information stored in RMS, and you will see the initial screen asking for a secret and PIN.
- **Encryption:** The PIN you enter is used to encrypt and decrypt all data at rest on the device.
- **Transport-layer Security:** The Gateway Engine is configured to allow mobile clients to communicate using HTTPS in order to provide transport-layer security.

◆ ◆ ◆ CHAPTER 4

Classes and Methods in the Mobile Client Business Object API Package

The Mobile Client Business Object (MCBO) API contains one package, `com.sun.mep.client.api`, that developers must use. This chapter summarizes the classes and methods contained within this package.

The API documentation is included in the MEP client bundle. In the directory where you unzipped the client bundle (see the *Sun Java System Mobile Enterprise Platform 1.0 Installation Guide* for details), it is in the directory `sjsmep_client_library-1_0_02-fcs/doc/mcbo/api`.

The `BusinessObject` Class

Table 4–1 lists the constructors and methods belonging to the `BusinessObject` class. This class is the base type for objects that will be synchronized with the Gateway Engine. Each business object instance is identified by a name, which is also used to name the file holding the serialized form of the object on the device's filesystem. You can create a `BusinessObject` from scratch by instantiating a concrete subclass, or you can create an empty `BusinessObject` and deserialize the contents of a byte array into it. Use the latter technique when implementing the `deserialize` method.

TABLE 4–1 Class com.sun.mep.client.api.BusinessObject

Method	Description
`BusinessObject()`	No-argument constructor.
`BusinessObject(java.lang.String name)`	Constructor that takes the name of the object as an argument.
`public abstract void deserialize(byte[] data)`	Deserializes the supplied byte array into this empty `BusinessObject`.
`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 the Enterprise Connectors. That is, clients and Enterprise Connectors must use the same extension for the same type of business object. Concrete subclasses should redefine this method.
`public java.lang.String getName()`	Returns the name of this `BusinessObject`.
`public abstract byte[] serialize()`	Serializes this `BusinessObject` into a byte array.
`public void setName(java.lang.String name)`	Sets the name of this `BusinessObject`. This method may only be called once, to initialize a new `BusinessObject` being deserialized. You may not change the name once it has been set. The name must be a unique identifier that can legally be used as a file name.

The `BusinessObjectStorage` Class

Table 4–2 lists the methods belonging to the `BusinessObjectStorage` class. This class manages the storage and retrieval of `BusinessObject` instances in their serialized form on the device's filesystem. The factory method used to get an instance of this class is `SyncManager.getBusinessObjectStorage`.

TABLE 4–2 Class com.sun.mep.client.api.BusinessObjectStorage

Method	Description
`public boolean deleteAllBusinessObjects()`	Deletes all of the business objects from the device's file system. Returns true if the operation succeeded, false otherwise.
`public void deleteBusinessObject(java.lang.String name)`	Deletes the serialized form of the `BusinessObject` with the specified file name.

TABLE 4–2 Class com.sun.mep.client.api.BusinessObjectStorage *(Continued)*

Method	Description
`public void deleteBusinessObject(BusinessObject obj)`	Convenience method equivalent to `deleteBusinessObject(obj.getName() + extension)`.
`public java.util.Vector listBusinessObjectNames()`	Returns a list of file names for all of the serialized `BusinessObject` instances that match the extension.
`public void readBusinessObject(BusinessObject result)`	Reads a serialized `BusinessObject` from the device's filesystem and returns the result by reference in the `result` parameter.
`public void writeBusinessObject(BusinessObject obj)`	Writes the serialized form of the specified `BusinessObject` to the device's filesystem, possibly replacing any existing data.

The DefaultSecurityManager Class

Table 4–3 lists the methods belonging to the `DefaultSecurityManager`class. This class provides a basic implementation of `SecurityManager` (see Table 4–5).

128–bit security keys are generated from the pin using MD5 digest. The key is used to reverse a basic pin derivatives algorithm for client authentication. It is also used as the symmetric key for Triple-DES encryption of data at rest on the device.

TABLE 4–3 Class com.sun.mep.client.api.DefaultSecurityManager

Method	Description
`public DefaultSecurityManager()`	No-argument constructor.
`public final byte[] computeKey(java.lang.String pin)`	Takes an arbitrary-length clear-text pin entered by the user and creates a fixed-length digest suitable for use by the encrypt and decrypt methods.
`public static final byte[] computeMD5Digest(byte[] dataBytes)`	Computes and returns an MD5 hash of the specified `byte[]`.
`public static final byte[] computeMD5Digest(java.lang.String data)`	Computes and returns an MD5 hash of the specified string.
`public final byte[] decrypt(byte[] cipherText)`	Callback handler to perform decryption of data on device. The MEP runtime will invoke this method whenever it is necessary to decrypt data.

TABLE 4–3 Class com.sun.mep.client.api.DefaultSecurityManager *(Continued)*

Method	Description
`public final byte[] encrypt(byte[] plainText)`	Callback handler to perform encryption of data on device. The MEP runtime will invoke this method whenever it is necessary to encrypt data.
`public final byte[] getKey()`	Returns the encryption key computed by `computeKey(String)`.
`public final int getValidationAttempts()`	Returns the number of validation attempts.
`public final boolean isPinSet()`	Returns true if the user has never logged into the application. Use this method to determine when the credentials need to be stored on the device.
`public void setKey(byte[] key)`	Sets the key on the `SecurityManager` so it can be used during callbacks to encrypt or decrypt data on the device.
`public final void storeCredentials(java.lang.String secret)`	Persists derivatives of the pin/key and the supplied secret on the device. These derivatives are used upon subsequent logins to validate the pin. The secret can be any non-null, non-zero length alphanumeric string. Typically, the application developer would prompt the user to enter a random sequence of key presses on the device and pass that value into this method. This is a single-use value, so the user does not need to remember it. This method should only be called once, when the user needs to set their pin number (if `isFirstLogin()` returns true).
`public final boolean validatePin(java.lang.String pin)`	Determines if the pin is able to recompute the derivatives stored on the device in `storeCredentials(byte[], String)`. If so, it returns true; otherwise, it returns false. This method also keeps track of how many times it has been invoked. If it exceeds the maximum number of allowed attempts (`getMaxValidationAttempts()`), a `SecurityException` is thrown.

The EncodingType Class

Table 4–4 lists the fields and the one method belonging to the EncodingType class. This class provides a simple enum for OMA DS encoding types. These values can be used to specify either simple XML encoding or WBXML encoding during synchronizations with the Gateway Engine.

The WBXML encoding is a binary WAP encoding standard developed by the Open Mobile Alliance (OMA) for compressing the OMA DS payloads in order to reduce bandwidth usage on mobile devices.

TABLE 4–4 Class com.sun.mep.client.api.EncodingType

Field/Method	Description
`public static final EncodingType WBXML`	WBXML encoding.
`public static final EncodingType XML`	Simple XML encoding.
`public java.lang.String getValue()`	Returns the value of the current encoding type.

The SecurityManager Class

Table 4–5 lists the constructors and methods belonging to the SecurityManager class. This class manages all of the client-side security features. A default implementation is provided (DefaultSecurityManager), but you are free to supply your own implementation.

Security features include the following:

Authentication — A simple form of pin-based authentication is provided. The `isPinSet()` method determines if the user of the application has set a pin yet. If not, you will need to prompt the user to enter a "secret", which is just a long random alphanumeric string, and a pin, which is also an alphanumeric string. You must compute a security key from the pin using `computeKey(String)` and then set the key on the SecurityManager with `setKey(byte[])`. You then persist the user credentials using the `storeCredentials(String)` method. This method computes a hash of the unencrypted secret and stores this value along with the secret encrypted with the key into the device's RMS record store.

If the user has already set their pin, use the `validatePin(String)` method to determine if the pin is valid. This method computes an encryption key from the supplied pin and reverses the encryption and hash calculation on the encrypted secret stored in RMS. If the hash calculations match, the pin is valid.

	The SecurityManager also keeps track of pin validation attempts (that is, how many times the MIDlet calls validatePin(String)). If the MIDlet exceeds the maximum number of attempts (getMaxValidationAttempts()), a SecurityException is thrown. The number of pin validation attempts is stored in RMS so they can be tracked across restarts of the MIDlet.
Encryption	The encrypt(byte[]) and decrypt(byte[]) methods are callbacks used by the MEP synchronization library to perform encryption and decryption of the business objects as they are read from and written to the device file system. If security is enabled, the data is never written to the device in clear text.
Data Fading	You can specify a maximum "quiet period" between successful synchronizations by using setMaxQuietPeriod(long). Using the getRemainingQuietTime(SyncManager) method, you can determine how much time is left before the quiet period expires. Based on this information, you could choose to destroy all of the business objects on the device.
Data Destruction	In certain circumstances, you may choose to destroy all of the business objects stored on the device. For example, if the MIDlet has failed pin validation too many times, or if the quiet period has expired since the last sync. To destroy the data, call the destroyBusinessObjects(SyncManager) method for each type of business object you wish to destroy.

TABLE 4–5 Class com.sun.mep.client.api.SecurityManager

Method	Description
public SecurityManager()	No-argument constructor.
public abstract byte[] computeKey(java.lang.String pin)	Computes an encryption key from the pin. There are no restrictions on the length of the pin or the key. It is up to the implementing class to produce the appropriate length key for the encryption algorithm being used. Calling this method will also set the key value so it can be retrieved using getKey().
public abstract byte[] decrypt(byte[] cipherText)	Callback handler to perform decryption of data on device. The MEP runtime will invoke this method whenever it is necessary to decrypt data.
public final boolean destroyBusinessObjects(SyncManager mgr)	Activates data destruction on business objects managed by the specified SyncManager. All business objects managed by the specified SyncManager will be deleted from the device.

TABLE 4–5 Class com.sun.mep.client.api.SecurityManager *(Continued)*

Method	Description
`public abstract byte[] encrypt(byte[] plainText)`	Callback handler to perform encryption of data on device. The MEP runtime will invoke this method whenever it is necessary to encrypt data.
`public abstract byte[] getKey()`	Returns the value of the key computed by `computeKey(String)`.
`public final long getMaxQuietPeriod()`	Returns the maximum allowable quiet time.
`public final int getMaxValidationAttempts()`	Returns the maximum allowable validation attempts.
`public final long getRemainingQuietTime(SyncManager mgr)`	Returns the time remaining (in milliseconds) before the quiet period expires. If there is no specified quiet period (that is, if the value returned by `getMaxQuietPeriod()` is less than zero), this method returns a negative value. Otherwise, it returns the time remaining in milliseconds, or zero if the quiet period has been exceeded. If a sync has not been attempted yet, it returns `getMaxQuietPeriod()`.
`public abstract int getValidationAttempts()`	Returns the number of validation attempts.
`public abstract boolean isPinSet()`	Returns true if the user has never logged into the application. Use this method to determine when the credentials need to be stored on the device.
`public abstract void setKey(byte[] key)`	Sets the key on the `SecurityManager` so it can be used during callbacks to encrypt or decrypt data on the device.
`public final void setMaxQuietPeriod(long period)`	Sets the maximum allowable quiet time (in milliseconds) between sync requests. If the device has been quiet for too long, the application will be prevented from performing the sync. Additionally, the application can decide to perform data destruction by calling `destroyBusinessObjects(SyncManager)`. Passing a negative value indicates that there should be no timeout.
`public final void setMaxValidationAttempts(int attempts)`	Sets the maximum allowable validation attempts. When the number of pin validation attempts exceeds this value, the application can decide to lock out the user and perform data destruction by calling `destroyBusinessObjects(SyncManager)`. The default value is 10 validation attempts. An argument of 0 (zero) indicates that there is no maximum number of validation attempts.

TABLE 4–5 Class com.sun.mep.client.api.SecurityManager *(Continued)*

Method	Description
`public abstract void storeCredentials(java.lang.String secret)`	Persists derivatives of the pin/key and the supplied secret on the device. These derivatives are used upon subsequent logins to validate the pin. The secret can be any non-null, non-zero length alphanumeric string. Typically, the application developer would prompt the user to enter a random sequence of key presses on the device and pass that value into this method. This is a single-use value, so the user does not need to remember it. This method should only be called once when the user needs to set their pin number (if `isFirstLogin()` returns true).
`public abstract boolean validatePin(java.lang.String pin)`	Determines if the pin is able to recompute the derivatives stored on the device in `storeCredentials(byte[], String)`. If so, it returns true; otherwise, it returns false. This method also keeps track of how many times it has been invoked. If it exceeds the maximum number of allowed attempts (`getMaxValidationAttempts()`), then a `SecurityException` is thrown.

The SMSMessageHandler Class

Table 4–6 lists the constructor and methods belonging to the `SMSMessageHandler` class. This class is a callback handler for SMS push notification messages sent from the Gateway Engine.

TABLE 4–6 Class com.sun.mep.client.api.SMSMessageHandler

Method	Description
`public SMSMessageHandler()`	No-argument constructor.
`public abstract void handlePoisonPill()`	Callback to allow client applications to respond to an SMS push poison pill request. Applications should immediately perform data destruction upon receiving this instruction.
`public abstract void handleSync(SyncType type)`	Callback to allow client applications to respond to an SMS push sync request. Applications should initiate a synchronization session of the requested type.

TABLE 4–6 Class com.sun.mep.client.api.SMSMessageHandler *(Continued)*

Method	Description
`public void processSMSMessagePayload(byte[] rawPayload, java.lang.String pin)`	Process the payload of the incoming SMS message and dispatch the instruction to the proper callback method (`handlePoisonPill()` or `handleSync(com.sun.mep.client.api.SyncType)`). The application's implementation of `javax.messaging.MessageListener` must call this method in order to decrypt and dispatch the instruction sent from the Gateway Engine. Make sure that this method is not invoked within the same thread that is delivering the SMS message. This method only supports processing SMS messages with text payloads. There is no support for binary SMS messages. If a pin is supplied, this method processes the payload as if it were encrypted. Otherwise, it is processed as clear text.

The SyncException Class

Table 4–7 lists the constructors and methods belonging to the SyncException class. This class is a checked exception type that indicates an error during synchronization.

TABLE 4–7 Class com.sun.mep.client.api.SyncException

Method	Description
`public SyncException(java.lang.String pMessage)`	Constructor that takes a `String` argument.
`public SyncException(java.lang.Throwable pLinkedException)`	Constructor that takes a linked exception argument.
`public SyncException(java.lang.String pMessage, java.lang.Throwable pLinkedException)`	Constructor that takes both a `String` and a linked exception as arguments.
`public java.lang.Throwable getLinkedException()`	Retrieves the linked exception for this exception.
`public void printStackTrace()`	Prints a stack trace for this exception to the standard error stream.

TABLE 4–7 Class com.sun.mep.client.api.SyncException *(Continued)*

Method	Description
`public void setLinkedException(java.lang.Throwable pLinkedException)`	Sets the linked exception for this exception.

The SyncManager Class

Table 4–8 lists the constructors and methods belonging to the `SyncManager` class. This class is responsible for managing synchronizations with the Gateway Engine for a particular kind of `BusinessObject` (classified by a particular extension). There is a one-to-one mapping between `SyncManager` and kinds of `BusinessObject` instances. If you have an application that deals with two different `BusinessObject` types, then you would have an instance of `SyncManager` for each type.

You must specify a unique `extension` for your `BusinessObject` when you construct this class. The `name` field in the `BusinessObject`, combined with the `extension`, will determine the entire name of the file on the device's filesystem. For example, you might pass the string `".act"` for account or `".ord"` for orders, but it can be anything legally allowed in a file name.

You may optionally specify a simple alphanumeric pin which will be used to encrypt any data stored locally on the device.

When you invoke the `sync(SyncType)` method, this will initiate a synchronization session with the Gateway Engine. During this session, only `BusinessObject` instances whose `extension` fields match will be synchronized.

TABLE 4–8 Class com.sun.mep.client.api.SyncManager

Method	Description
`public SyncManager(java.lang.String extension)`	Constructor that creates a new `SyncManager` for the specified business object type (`extension`) with no `SecurityManager`. This method is the equivalent of calling `new SyncManager(extension, null)`.
`public SyncManager(java.lang.String extension, SecurityManager sm)`	Constructor that creates a new `SyncManager` for the specified business object type (`extension`) and uses the specified `SecurityManager`.
`public void enableLogging(boolean value)`	Enables or disables debug logging in the MEP client APIs. If enabled, the MEP implementation library writes logging information to `stdout` and also to a log file (named `meplog.txt`) on the device. Logging is disabled by default.

TABLE 4–8 Class com.sun.mep.client.api.SyncManager *(Continued)*

Method	Description
`public BusinessObjectStorage getBusinessObjectStorage()`	Returns the `BusinessObjectStorage` manager associated with this `SyncManager`. There is a strict 1:1 relationship between `SyncManager` and `BusinessObjectStorage`.
`public EncodingType getEncoding()`	Returns the current transport encoding type.
`public java.lang.String getExtension()`	Returns the extension type associated with this `SyncManager`.
`public static final SecurityManager getSecurityManager()`	Returns a reference to the security manager, or null if one has not been set.
`public SyncResults getSyncResults()`	Returns the sync results for the latest successful sync.
`public long getTimeOfLastSync()`	Returns a time stamp of the last successful sync. This time stamp is recorded by calling `System.currentTimeMillis()`, so it is an offset from January 1, 1970 UTC. If a sync has not been performed yet, the method returns a negative value.
`public void setCredentials(java.lang.String username, java.lang.String password, java.lang.String url)`	Sets the credentials used during the synchronization process.
`public void setEncoding(EncodingType encoding)`	Sets the transport encoding type.
`public void sync(SyncType syncType)`	Performs the specified type of synchronization.

The SyncResults Class

Table 4–9 lists the methods belonging to the `SyncResults` class. This class contains coarse-grained statistics about the most recent synchronization (additions, deletions, and modifications).

TABLE 4–9 Class com.sun.mep.client.api.SyncResults

Method	Description
`public int getClientAdditions()`	Returns the number of records added on the client and sent to the server.
`public int getClientDeletions()`	Returns the number of records deleted on the client and sent to the server.

TABLE 4–9 Class com.sun.mep.client.api.SyncResults *(Continued)*

Method	Description
`public int getClientUpdates()`	Returns the number of records updated on the client and sent to the server.
`public int getServerAdditions()`	Returns the number of records added on the server and sent to the client.
`public int getServerDeletions()`	Returns the number of records deleted on the server and sent to the client.
`public int getServerUpdates()`	Returns the number of records updated on the server and sent to the client.

The SyncType Class

Table 4–10 lists the fields and methods belonging to the SyncType class. This class provides a simple enum for the available synchronization types.

TABLE 4–10 Class com.sun.mep.client.api.SyncType

Field/Method	Description
`public static final SyncType BACKUP_SYNC`	Refresh sync from client. All of the server data is replaced with the client data. Use with caution.
`public static final int BACKUP_SYNC_VALUE`	Constant enum value for BACKUP_SYNC.
`public static final SyncType FAST_SYNC`	Two-way fast sync. Client modifications since the last synchronization are sent to the server for reconciliation, and server modifications since the last synchronization are sent back to the client.
`public static final int FAST_SYNC_VALUE`	Constant enum value for FAST_SYNC.
`public static final SyncType ONE_WAY_CLIENT_SYNC`	One-way sync from client. Client modifications are sent to the server, but server modifications are not sent back to the client.
`public static final int ONE_WAY_CLIENT_SYNC_VALUE`	Constant enum value for ONE_WAY_CLIENT_SYNC.
`public static final SyncType ONE_WAY_SERVER_SYNC`	One-way sync from server. Server modifications are sent to the client, but client modifications are not sent to the server.
`public static final int ONE_WAY_SERVER_SYNC_VALUE`	Constant enum value for ONE_WAY_SERVER_SYNC.

TABLE 4–10 Class com.sun.mep.client.api.SyncType *(Continued)*

Field/Method	Description
`public static final SyncType RESTORE_SYNC`	Refresh sync from server. All of the client data is replaced with the server data.
`public static final int RESTORE_SYNC_VALUE`	Constant enum value for `RESTORE_SYNC`.
`public static final SyncType SLOW_SYNC`	Two-way slow sync. This is the same as a fast sync except that ALL client data (including unmodified records) is sent to the server for reconciliation.
`public static final int SLOW_SYNC_VALUE`	Constant enum value for `SLOW_SYNC`.
`public java.lang.String getDescription()`	Returns a description of the current `SyncType`.
`public int getValue()`	Returns the value of the current `SyncType`.