Accessing DB2 Everyplace using J2ME devices, part 2

Skill Level: Introductory

Naveen Balani (naveenbalani@rediffmail.com)
Developer

22 Apr 2004

This two-part tutorial shows developers how to build DB2 Everyplace mobile applications using J2ME DB2Everyplace MIDP ISync APIs.

Section 1. Introduction

Overview

This tutorial shows developers how to build DB2 Everyplace mobile applications using J2ME DB2 Everyplace MIDP ISync APIs and how to deploy the application in the J2ME environment using the J2ME Toolkit.

Part one of this two-part tutorial uses the DB2Everyplace J2ME MIDP ISync Client APIs to create a sample address book database on a J2ME Emulator/Device from a remote DB2 database. The address book database on the J2ME Emulator/Device is stored using the J2ME Record Store Management System.

Part two builds an address book application to access an address book record store, perform updates to it and then synchronize it with the remote DB2 address database. We use DB2 Everyplace MIDP APIs to access the address book database and perform updates to it.

Prerequisites

The developers must have knowledge and an understanding of DB2 Everyplace and J2ME.

This article requires that you have already installed DB2 Everyplace Enterprise Edition 8.1.4, DB2 Everyplace Software Development Kit 8.1.4, and J2ME ToolKit.

Software requirements:

The software required for the tutorials is:

- Windows 2000 with Service Pack 3 or above
- DB2 Everyplace 8.1.4 Enterprise Edition from DB2 Everyplace Enterprise Edition
- DB2 Everyplace 8.1.4 Software Development Kit from DB2 Everyplace Software Development Kit
- J2ME ToolKit 2.1 from J2ME Wireless Toolkit 2.1

Section 2. Getting started

Overview of record storage management using J2ME RMS

This section describes how DB2 Everyplace stores and updates records using the J2ME Record Store Management system.

The DB2 Everyplace record structure persisted on the J2ME mobile device using J2ME RMS has the following structure.

Row Data Format
Every row has an extra byte at its beginning to indicate if the application modified the row; this byte is intially zero. The application designer is required to update this byte if they DELETE, UPDATE or INSERT a row into RMS. This indicates that on a future SYNCHRONIZE request the rows should be sent to DB2 Everyplace to be included in the source database.

Failure to set this byte to ISync.ROW_ADDED, ISync.ROW_CHANGED, ISync.ROW_DELETED will cause your changes to never be replicated.

Within each row, for each column, if it is nullable, we need to call DataInputStream.readBoolean to get the null indicator. If null, no data for the column

follows.

If not null (indicator is false) or if the column is not nullable, the following table indicates which DataInputStream/DataOutputStream method to call.

The following table shows the Data Type Mapping from the source database to java.sql.Types to MIDP Row Data.

SQL Data Type on Data Source	java.sql.Types on MIDP Client	Java Data Type on MIDP Client
Datalink	// unsupported	N/A
Time	INTEGER	int
Date	INTEGER	int
Timestamp	LONG	long
Blob	LONGVARCHAR	String, or byte[]
Char	CHAR	String
Varchar	VARCHAR	String
Clob	LONGVARBINARY	String, or byte[]
Graphic	// unsupported	N/A
Vargraphic	// unsupported	N/A
Dbclob	// unsupported	N/A
Real	REAL	String
double precision	DOUBLE	String
Bit	VARCHAR	String
//unsupported	TINYINT	N/A
Smallint	SMALLINT	short
Int	INTEGER	int
bigint, int8	BIGINT	long
Numeric	VARCHAR	String

The following table shows the DataInputStream and DataOutputStream methods to use based on the MIDP row format.

SQL Type	JAVA Type	DataOut/InputStream method
INT8, BIGINT	Long	writeLong, readLong
CHAR	String	writeUTF, readUTF
VARCHAR	String	writeUTF, readUTF
DATE, TIME	Int (see below)	writeInt, readInt
TIMESTAMP	Long (see below)	writeLong, readLong
DECIMAL, NUMERIC	String	writeUTF, readUTF
INTEGER	Int	writeInt, readInt
SMALLINT	Short	writeShort, readShort

Analyzing the record store with the address database

This tutorial explains the DB2 Everyplace J2ME MIDP record store by revisiting the address database example that we used in part one of this tutorial series. Our address database is represented as follows:

```
   CREATE TABLE ADDRESS(ADDRESS_ID Char(30) not null primary  key,FirstName Char(30),
LastName Char(30),StreetAddress char(50),PhoneNumber char(15))
```

If we insert a row in our address table in DB2,

```
   INSERT INTO ADDRESS VALUES ('0000000001','ALEX','STEWART','1500 DEC ROAD, CAF CITY , CA
94357','510-999-7898')
```

Then, DB2 Everyplace row data on the MIDP Emulator/Device would be interpreted as follows:

```
   DataOutputStream dout = new DataOutputStream(byteArrayOutStrm);
   dout.writeByte(0); // dirty byte, flags byte at start of row
   dout.writeUTF("0000000001");//address_id is non nullable, no null indicator needs to be
set.

   dout.writeBoolean(false); // firstname is nullable, but not null hence we need to set
null indicator to false.
   dout.writeUTF("ALEX");//firstname data

   dout.writeBoolean(false); // lastname is nullable, but not null hence we need to set
null indicator to false.
   dout.writeUTF("STEWART");//lastname data
```

```
   dout.writeBoolean(false); // address is nullable, but not null hence we need to set null
indicator to
   false.
   dout.writeUTF("1500 DEC ROAD, CAF CITY , CA 94357");//address data

   dout.writeBoolean(false); // phone number  is nullable, but not null hence we need to
set null
   indicator to false.
   dout.writeUTF("510-999-7898");//phone number data
```

For example, if we insert the following address data in DB2:

```
   INSERT INTO ADDRESS VALUES ('0000000001',null,'STEWART','1500 DEC ROAD, CAF CITY , CA
94357','510-999-7898')
```

Then, DB2 Everyplace row data on the MIDP Emulator/Device would be interpreted as follows:

```
   DataOutputStream dout = new DataOutputStream(byteArrayOutStrm);
   dout.writeByte(0); // dirty byte, flags byte at start of row
   dout.writeUTF("0000000001");//address_id is non nullable, no null indicator needs to be
set.

   dout.writeBoolean(true); // firstname is nullable, and data is null hence we need to set
null indicator to true.

   dout.writeBoolean(false); // lastname is nullable, but not null hence we need to set
null indicator to false.
   dout.writeUTF("STEWART");//lastname data

   dout.writeBoolean(false); // address is nullable, but not null hence we need to set null
indicator to
   false.
   dout.writeUTF("1500 DEC ROAD, CAF CITY , CA 94357");//address data

   dout.writeBoolean(false); // phone number  is nullable, but not null hence we need to
set null
   indicator to false.
   dout.writeUTF("510-999-7898");//phone number data
```

Next, we look at the DB2 Everyplace MIDP APIs, which allow us easily to manipulate and access the required underlying data.

Section 3. Overview of MIDP DB2 Everyplace APIs

FastRecordStore and FastRecordEnumeration API overview

In this next section, we list the functionality of DB2 Everyplace MIDP APIs that we previously used for our address book application.

FastRecordStore class is build on top of the basic javax.microedition.rms.RecordStore class to provide performance improvements for writes (by providing a write buffer) as well as update, delete and find by primary key methods.

FastRecordEnumeration class provides provides a similar interface to javax.microedition.rms.FastRecordStore. It provides methods for enumerating over a FastRecordStore object.

Next we look at some of the important methods to access the Record Store.

- public static FastRecordStore openRecordStore(java.lang.String recordStoreName, boolean createIfNecessary)
  This method returns an instance of FastRecordStore. The recordStoreName represents the underlying record store name, which corresponds to remote database name if a synchronization has been performed; in our case its value is "ADDRESS."
- public static FastRecordStore openRecordStore(java.lang.String recordStoreName, boolean createIfNecessary)
  This method returns an instance of FastRecordStore. The recordStoreName represents the underlying record store name, which corresponds to remote database name if a synchronization has been performed; in our case its value is "ADDRESS."
- public FastRecordEnumeration enumerateRecords(javax.microedition.rms.RecordFilter filter,javax.microedition.rms.RecordComparator comparator,boolean keepUpdated)
  This method returns a new enumeration over the current FastRecordStore instance. Thus using the enumeration object we can loop through our records in our record store. Records can be visualized as corresponding to each rows in a table.
- public short getRecord(int recordId)
  This method returns a copy of the data stored in targetRecord, creating a new byte array each time to hold the record. We can get the record by enumerating through the FastRecordEnumeration class and using the method nextRecordId() of the FastRecordEnumeration class. The nextRecordId() returns the recordId of the next record in this enumeration.

The Index and TableMetaData class API overview

The **Index** class provides the developer with the ability to search, update and delete based on the primary key of a row. In order to keep the primary key index up to date, the developer must perform all "modification" operations via an instance of the Index class.

Primary Key operations provided by the Index class are findByPrimaryKey, deleteByPrimaryKey and updateByPrimaryKey. Methods are provided to ensure that the primary key is kept up to date and which must be used in place of the modification methods of RecordStore (addRecord, deleteRecord and setRecord) are insertRecord, deleteRecord and updateRecord.

Finally, findRecord(int), findRecord(int, byte[]) and findRecordSize(int) are also provided as convenience functions that wrap the functionality of getRecord(int), getRecord(int, byte[]) and getRecordSize(int), respectively.

Next we look at some of the important methods to create, insert and update the Record Store.

- Index(FastRecordStore store, TableMetaData table)
  This constructor creates a new Index Object using the FastRecordStore instance and the TableMetaData instance. All modifications to the data in record store can only be performed via the Index instance.
- updateRecord(int recordId, byte[] data, int numBytes)
  This method updates a record to the underlying FastRecordStore and modifies the primary key index. The recordId represents the record ID of the row in the Record Store to update. Data represents the new data for the row; even if only part of the row has changed, the entire row is replaced and numBytes represents the length of data to be written.
- insertRecord(byte[] data, int numBytes)
  This methods inserts a record to the underlying FastRecordStore and modifies the primary key index.
- close()
  The close method writes the changes in the Primary Key index to non-volatile storage and closes the underlying RecordStore. It is essential that for each instance of Index the index.close() method is called when the developer has finished operations on the Index. This ensures that changes to the Index structure are written to non-volatile memory and the underlying RMS table is closed.

TableMetaData Class: This class holds table information within an ISyncSubscriptionSet, consisting of the name of the table, the number of columns in

the table, the type of each column, if it is nullable, and if it is a primary key. The application (MIDLET) designer is responsible for reading and writing the raw data in the byte arrays read from and written to FastRecordStore and Index.

Next we move on to analyze the address book application code.

Section 4. Building the sample address book application

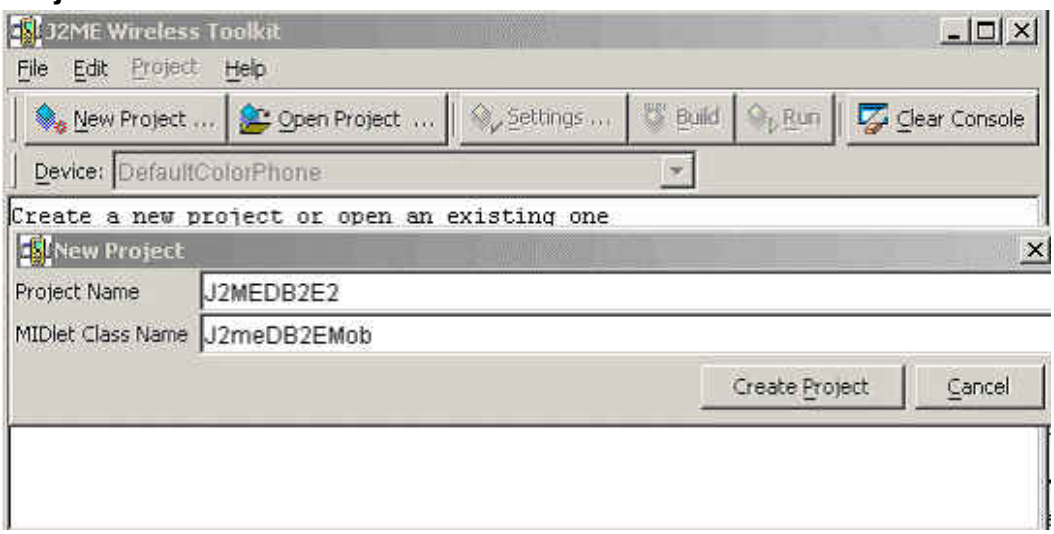
Importing the sample address book MIDlet application

In this section we import our existing sample address book MIDlet application. Extract the Sample-J2MEDB2E2.zip into any directory, say C:\. A folder J2MEDB2E will be created. Our address book MIDlet application is built upon the MIDlet application discussed in part 1 of this tutorial series.

First, create a new MIDP project in the J2ME toolkit. Follow these steps:

1. Navigate to **Start => Programs => J2ME Wireless Toolkit 2.1 => Ktoolbar.** The toolkit window will open.

2. Create a new project using Ktoolbar. Enter **`J2MEDB2E2`** as the project name and **`J2meDB2EMob`** as the MIDlet class name. Click on **Create Project.**



3. The next window will give you the opportunity to specify the project settings. Click **OK** to accept the deafults.

4. Copy the IsyncMidp.jar file from DB2EveryplaceSDKinst\Clients\MIDP\lib to c:\wtk21\apps\J2MEDB2E2\lib. The IsyncMidp.jar file contains the necessary DB2E MIDP synchronization classes and DB2-specific RMS classes.

5. Copy the source file, `J2meDB2EMob.java`, from `c:\J2MEDB2E` to `c:\wtk21\apps\J2MEDB2E2\src`. (Remember, the J2ME Wireless Toolkit is installed in the `C:\wtk21` path.)
J2meDB2EMob.java is a MIDlet application that uses the DB2 MIDP Sync APIs for synchronizing with our address database application. When synronization is complete, the DB2 MIDP Sync APIs automatically persists our address book data to J2ME devices by using the available Java persistence API, this being the Record Management System (RMS) for MIDP. After data is persisted we use DB2 Everyplace MIDP APIs discussed earlier to access the address record store through a Search User Interface. If the results are retrieved from search criteria, the user is given a menu to update the corresponding address record, which uses the Index class that was discussed earlier to update the corresponding record. Once done, we synchronize the data, so updates are reflected in the remote address database.

6. Click the **Build** button on the Ktoolbar. You will receive the following message:

```
Project settings saved
Building "J2MEDB2E2"
Build complete
```

Congratulations. You have successfully built your `J2meDB2EMob.java` MIDlet.

Section 5. Our MIDlet application: Code review

Adding command and User Interface elements

We will consider only the extensions that we have made to our Part 1 MIDlet

application.

```
Line 1: : //UI Elements
   TextField searchField = new TextField ("Search By Name", "", 5, TextField.ANY);
   TextField resultFName = new TextField ("FirstName", "", 5, TextField.ANY);
   TextField resultLName = new TextField ("LastName", "", 40, TextField.ANY);
   TextField resultAdd = new TextField ("Address", "", 100, TextField.ANY);
   TextField resultPhone = new TextField ("Phone", "", 20, TextField.ANY);


Line 2: //Global fields
   int currentRecordId = 0;


Line 3: : //Command buttons
   private Command searchContactButton;
   private Command searchResultsButton;
   private Command updateButton;
   private Command syncButton;
```

Line 1 defines the UI elements for for address book application. Line 2 defines the currentRecordId we are working on.

Line 3 defines the various command interfaces for our address book application.

Executing the commands

```
Line 4: public void commandAction(Command c, Displayable s) {

Line 5: if (c == searchContactButton)
      {
         searchForm = new Form("Search Address By First Name");
          //Add the search criteria field
         searchForm.append(searchField);

         //Add the command buttons
         searchForm.addCommand(searchResultsButton);
         searchForm.setCommandListener(this);
```

```
          display.setCurrent(searchForm);

      }


Line 6: if(c == searchResultsButton){
          //Call ReadRecords and get the result sets of serach criteria in vector
          Vector contactResults = readRecords(storeName,true);
          if(contactResults !=null){

          Form searchResultsForm = new Form("Search Results");
          //Add the results from contactResults vector to UI elements

          searchResultsForm.addCommand(updateButton);
               display.setCurrent(searchResultsForm);

   }

Line 7 : if(c == updateButton){

          updateRecord();

      }
```

Line 5 performs a check if the user has clicked the searchContact button; if yes the Search Address By First Name form is displayed.

Line 6 performs a check if the user has clicked the searchResultsButton button; if yes the Search Results form is displayed, which displays the address infomration. The user can perform edits to this address information.

Line 7 performs a check if the user has clicked the updateButton button; if yes the updateRecord() method is executed.

Read Record method

The readRecord() method is used to read records from the address record store. We analyze some of the important steps of this method.

```
Line 8: Vector readRecords(String storeName,boolean isSearch) {
```

```
Line 9 : FastRecordStore rms = FastRecordStore.openRecordStore(storeName, false);

Line 10: FastRecordEnumeration enum = rms.enumerateRecords(null, null, false);

Line 11: while(enum.hasNextElement() && contactResults == null){
    currentRecordId = enum.nextRecordId()
    contactResults = readAdressRecords(din,isSearch);
 }
```

The readRecords() method takes two parameters, storeName, which represents the record store name (i.e., which is "ADDERESS" for our application) and the isSearch parameter, which is a boolean value, denoting if all records need to be read in case no search criteria is specified.

Line 9 gets an instance of FastRecordStore using openRecordStore method as discussed earlier. Line 10 gets an instance of FastRecordEnumeration class for enumerating through our record store.

Line 11 iterates through the enumeration, gets the recordID and calls our readAdressRecords() method, which reads each address record and compares it with user-entered firstname. if a match is found, we populate the results back in the vector and return the vector. For simplicity we return only the first record, which satisfies the user-entered criteria.

Update Record method

The updateRecord() method is used to update records in the address record store. We analyze some of the important steps of this method.

```
Line 12 : ByteArrayOutputStream bout = new ByteArrayOutputStream();
     DataOutputStream dout = new DataOutputStream(bout);

Line 13: dout.writeByte(ISync.ROW_CHANGED);

Line 14: dout.writeUTF(currentAddressId);

Line 15: //2nd column
         dout.writeBoolean(false);
    dout.writeUTF(resultFName.getString());
         // Similar for remaning columns
```

```
Line 16: FastRecordStore rms = FastRecordStore.openRecordStore(storeName, false);

Line 17: Index  index = new
Index(rms,((MIDPISyncProvider)isync).getTableMetaDataByName(storeName));


Line 18: index.updateRecord(currentRecordId, bout.toByteArray(),bout.size());


Line 19: index.close();
```

At line 12 we create ByteArrayOutputStream and DataOutputStream object to hold our address record that needs to be updated.

At line 13 we write the byte ISync.ROW_CHANGED, denoting that the row has been updated. The record store format was discussed earlier in Getting Started section.

At line 14 we write out the first column, which is our address ID. The address ID is our primary key for our remote DB2 address table. Since it is not null value we don't write the boolean indicator flag for our address ID.

At line 15 we write the boolean indicator flag to false denoting that the value is not null, followed by the data from the resultFName TextField UI element entered by the user.

Line 16 gets an instance of FastRecordStore using openRecordStore method as discussed earlier in Section 3. Line 17 gets an instance of Index Class using the new Index construtor method as discussed earlier in Section 3.

At line 18 we update the record using the Index update method and finally at Line 18 close the index object to make the changes persistent.

Next we see our code in action.

Section 6. Running the address book application

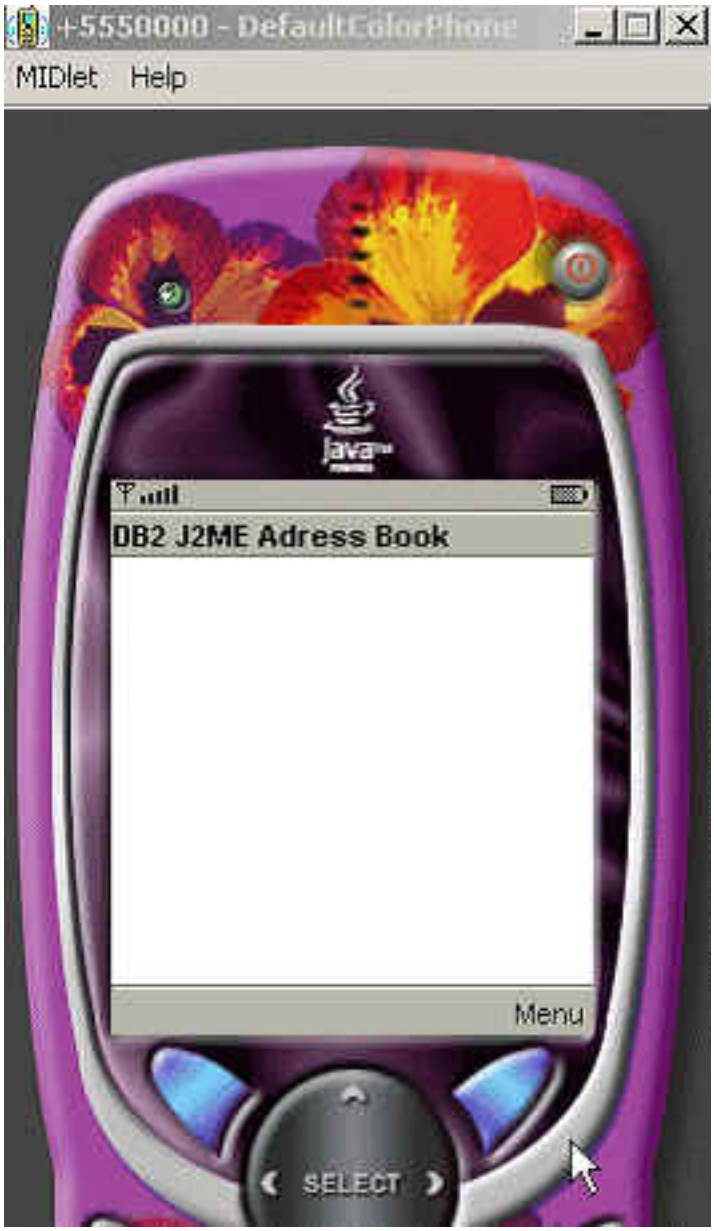
Running the application

Before running the application, start the Sync Server by navigating to **Program Files => IBM DB2 Everyplace => Start Servlet for SyncServer**.

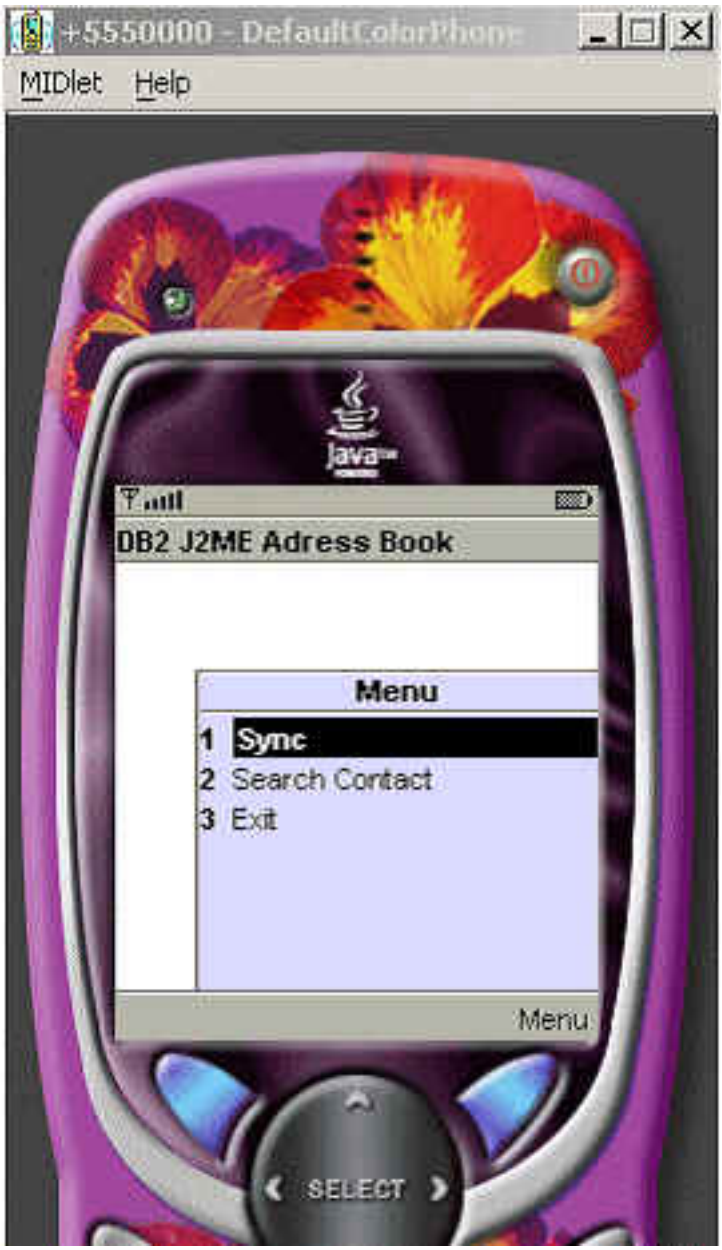
To run the sample J2MEDB2E2 application, click the **Run** button on the Ktoolbar.

The default emulator shown in the following figure should appear.

On launching the application, you should see the following screen:



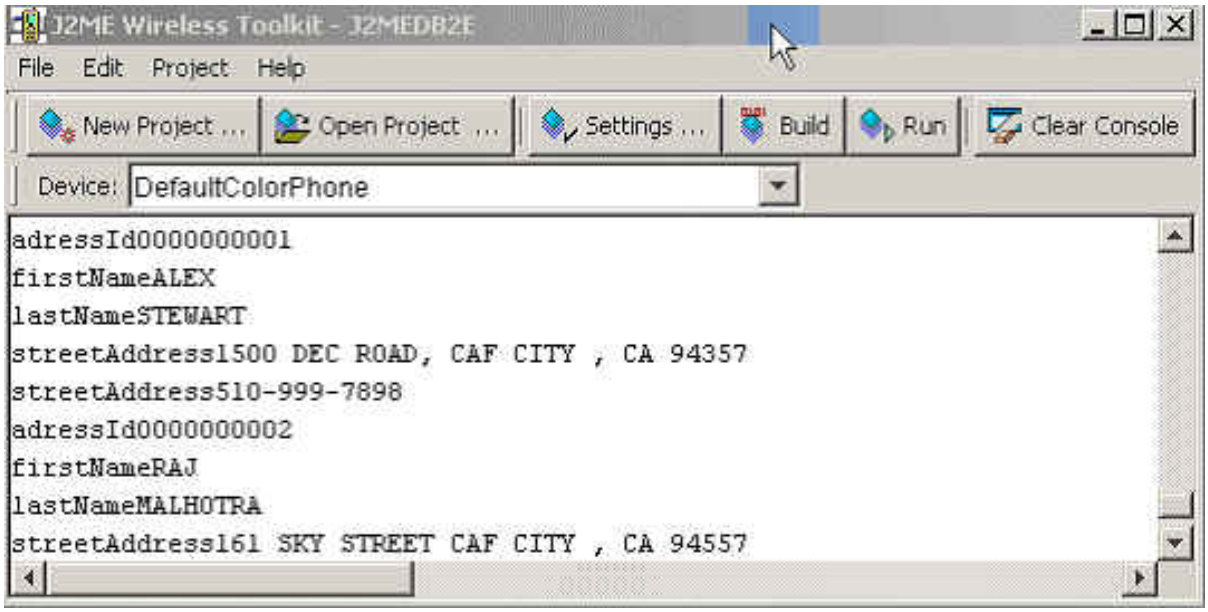
Click on the **Menu** and the following drop-down list will appear. Choose the **Sync** option to perform synchronization.



Building the application

On clicking the Sync button, the synronization process will start and J2ME Emulator will try to connect to the DB2 Synchronization server. You will receive a warning message during connecting as shown below. Click **OK** to proceed with a network connection to the DB2 Synchronization server in order to carry out the synchronization process.

When synronization is complete, the DB2 MIDP Sync APIs automatically persists our address book data to J2ME devices using the available Java persistence API, this being the Record Management System (RMS) for MIDP. The readRecords method is called, which reads all our address book records created on the J2ME Emulator/Device and displays it to the output console. At the Ktoolbar console we can view all the records read. Given below is the output of Ktoolbar console.



When synchronization is complete, click on **Menu** and select **Search Contact**, as shown in the figure below.

After clicking the Search Contact option, the following screen is displayed. Enter **RAJ** in the Search by Name field. Click on **Menu** and select **Search By First Name**.

When you choose the Search By First Name option, the following screen is displayed, which shows the corresponding address information.

Updating the application

To perform updates to the data, click on **Menu** and select the **Update** option to update the corresponding record.

Click on **Menu** and the sync option to synchronize the updates with the remote DB2 address tables.

To be sure the data is properly synchronized, open up the DB2 Command prompt for the DB2 source database and enter:

DB2 Connect to Address
DB2 Select * from Address

Since we have set the replication timeout parameter to 60 secs in MDAC, we need to wait for 60 secs to see the changes being reflected to the source database. You should see the same data that you entered in the address book application.

We have successfully created our address book application, accessed it and performed updates to the address record store using DB2 Everyplace MIDP APIs. We also synchronized our changes with our remote DB2 address database.

Section 7. Summary

Wrap up

In this tutorial we successfully developed and deployed an address book application on J2ME emulator using the J2ME Toolkit. We also performed updates to the address record store using the DB2 JM2E MIDP APIs and synchronized the updates with our remote DB2 Address database.

About the author

Naveen Balani
Naveen Balani spends most of his time designing and developing J2EE-based products. He has written various articles for IBM in the past covering topics from JMS, SOA, SOAP, Web services, Architecture Patterns, MQSeries, WebSphere Studio, XML, JAVA Wireless Devices and DB2 Everyplace for Palm, Java-Nokia and Wireless Data Synchronization.