Record Management System: Persistent Memory

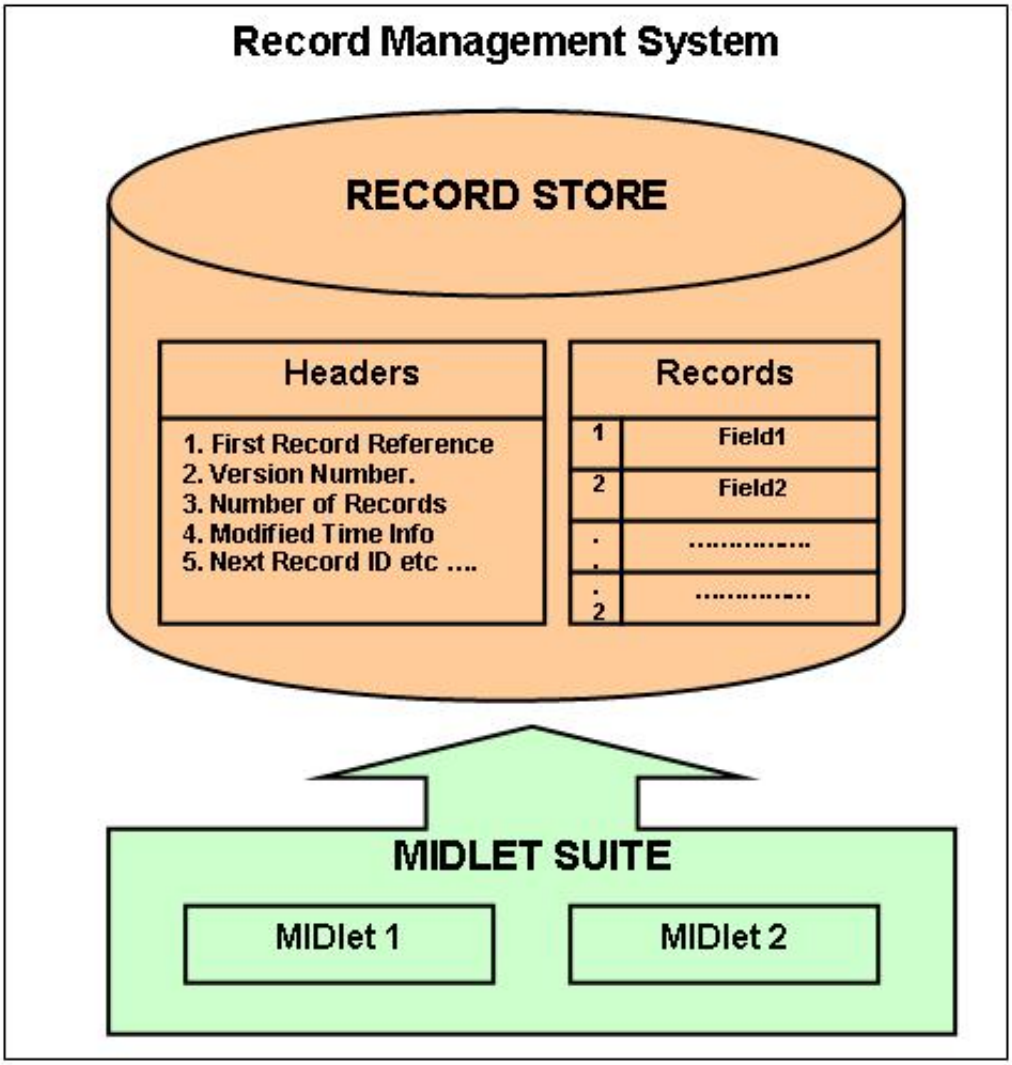
Back | Next

The Mobile Information Device Profile provides persistent storage of data through a simple a simple record-oriented database called the Record management system (RMS).

Data is stored over multiple sessions in nonvolatile memory, However in Palm OS devices the data is stored in volatile memory. In case of volatile memory the data will be lost if the battery of the device is removed for a couple of minutes.

J2ME Record Management System (RMS) Architecture

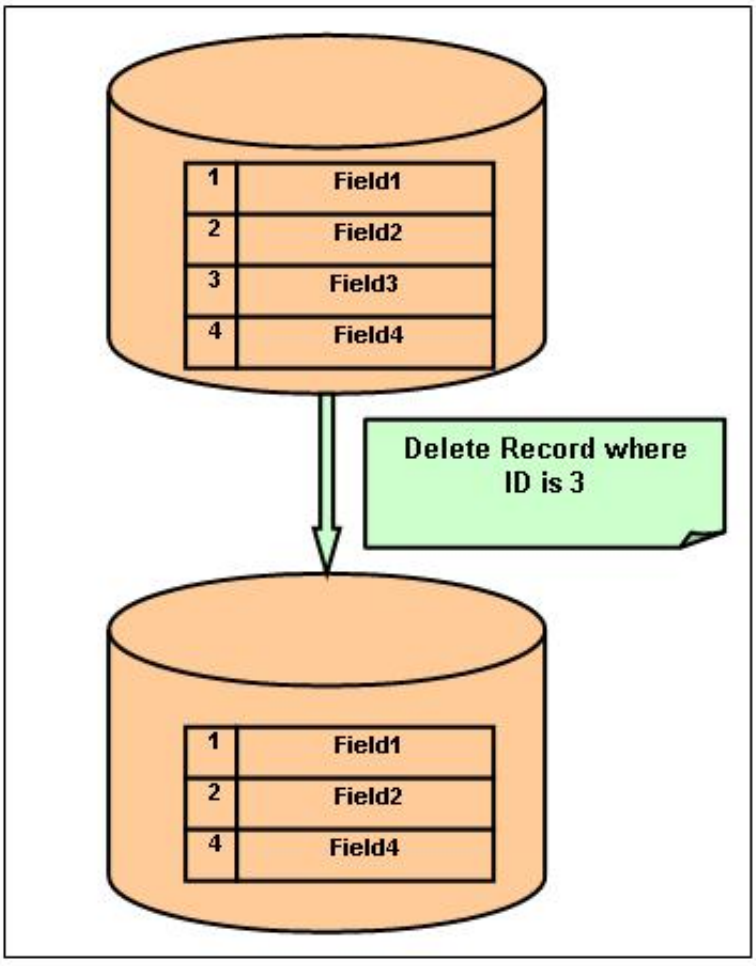
RMS manages persistent data through Record Stores. Each record store has Record Header and a collection of Records. The below figure gives an idea of the architecture of the RMS database.



Records in RMS are completely different from the records in the normal database system. Do not compare MIDP RMS with your normal database, RMS has no primary keys, foreign keys, stored procedures etc. Each record consists of a record ID and a single binary data field. All data should be converted to byte array before being added to the record store. Multiple fields are not allowed in a single record.

Compatible algorithm should be written by the programmer like storing multiple fields into a single record. If your record has First Name, Last Name as fields then you can store these fields into a single record by making them coma or pipe separated and then stores them to a Record Store. While retrieving the data the record should be decoded for further use.

Record ID's are unique identifiers that are added along with the inserted data by the RMS system. Record ID is 1 for the first record and gets incremented by one on every addition of a new record. If any record is deleted from the Record Store the Record ID's will not be reset in sequential order. The below figure shows diagrammatically the effect of deletion of record with record ID 3 from the record store.



Record Header contains the data (meta-data) about of the Record Store. The various header information stored in a Record Header is.

- Reference to the first record in the Record Store.
- Version Number of the Record Store. This parameter is zero for a newly created data store and is incremented by one for each modification to database.
- Number of records in the Record Store.
- Last Modified Time of the record store is updated to the header. This information can be used for coding custom synchronization algorithms as RMS by itself does not do any data synchronization.
- Next available Record ID in the Record Store.
- Reference to next available Record location. This reference is used by the RMS to insert a new record.

Multiple MIDlets using same Record Store

Multiple MIDlets can access the same Record Store if these MIDlets are within the same MIDlet Suite. As we already know MIDlets are packaged as JAR files and each of these JAR file is a MIDlet Suite. MIDlets from one JAR file cannot access the Record Store created by the MIDlet from another JAR File.

Programming with RMS

The javax.microedition.rms is the package that does all the Record Store dirty work for you. This package defines classes and interfaces for Record Store manipulation.

Class Record Store

Class RecordStore is the main class in this package and provides several methods to create, insert, update, and delete Record Stores

Create a Record Store

Class javax.microedition.rms.RecordStore proivides a static method openRecordStore(String recordStoreName, boolean createIfNecessary) which can be used to both create and open a Record Store. Parameter recordStoreName is the name of the record store. If createIfNecessary parameter is true then new Record Store is created if the Record Store with the name recordStoreName does not exist in the device and if it is set as false the record store is not created. If the record store is found in the device createIfNecessary flag is neglected.

```
RecordStore rs = RecordStore.openRecordStore("MobileSalsaDB",true);
```

Closing Record Stores

Like in JDBC you will need to do some cleaning up when the Record Store is not in use. It is very important to close an idle Record Store as RMS interactions eat up a lot of device runtime cache. To close a Record Store call closeRecordStore() .

```
Usage: rs.closeRecordStore();
```

Deleting a Record Store

A Record store can be deleted by deleteRecordStore() method.

```
Usage: RecordStore.deleteRecordStore("MobileSalsaDB");
```

Inserting and Deleting records

What is the use of a database if you can only create and delete one. RecordStore class gives the following methods for Inserting and Deleting records.

public int addRecord(byte[] data, int offset, int numBytes) inserts a record data with offset as its starting index and numBytes as its length.

You have already read in this section that you can only insert byte arrays into a Record Store. Though the above method looks complicated the listing below makes your life simpler.

Listing to Insert a record

```
String strSalsa = "Jar of Spanish Salsa"; //
String I wish to add to byte bytesSalsa[] = strSalsa.getBytes(); // Convert data to Byte
```

Array. rs.addRecord(bytesSalsa,0, bytesSalsa.length); // Add Byte Array to Record Store.

Method deleteRecord(int recordId) is invoked to delete a record from the record store. This deleted record and its ID is lost forever. We cannot reuse the deleted record ID.

Usage: rs.deleteRecord(1);

Query and Update records

Class RecordStore supplies the getter and setter methods for reading and manipulation of the existing records in the Record Store. The records can be referenced by the recordId's and can be queried and updated.

- Use getRecord(int recordId, byte[] buffer, int offset) to get the record with recordId. The record is stored in the variable buffer from the initial offset.
- byte[] getRecord(int recordId) is similar to the above method but you cannot specify the initial offset. Here the offset is always 0.
- setRecord(int recordId, byte[] newData, int offset, int numBytes) is used to update the record at specified record ID. The updation is done with the record newData with offset as its starting index and numBytes as data length.

Listing to Update a record

```
String strUpdateSalsa = "Salsa is a dance";
Byte byteUpdateSalsa = strUpdateSalsa.getBytes();
rs.setRecord(1, byteUpdateSalsa, 0, byteUpdateSalsa.length());
```

J2ME Wireless Toolkit Tip: To clear the values in RMS in WTK emulators select File > Utilities from the main menu and click on the "Clean Database" button.

Advanced RMS

The RMS system also has number other utility classes and interfaces which help us to do advanced tasks like traversing and filtering records.

Traversing the records with RecordEnumeration

RecordEnumeration Interface creates a bidirectional Enumeration object of the Record Store. This Enumeration object is used to read the contents of the Record Store and cannot be modified. Like a view is created from a database an Enumeration object can be created having selected values depending on filters and sorted orders. RecordFilter and RecordComparator are used for this purpose.

RecordStore.enumerateRecords (RecordFilter filter, RecordComparator comparator, boolean keepUpdated) is used to create a RecordEnumeration object. keepUpdated is set to true the Enumeration object is always latest. The Record Store is polled regularly to keep the Enumeration object latest and thus affecting the performance. If keepUpdated is set to false the stale Enumeration object can be updated using RecordEnumeration.rebuild() method.

Record Enumeration object is then used to traverse through the records by using

- nextRecord()
- previousRecord()

Listing of RecordEnumeration

```
private void showEnumRecordsForm() {
        enumList = new List("Enumeration Demo", Choice.IMPLICIT);
        try {
                // Create Enumeration
                RecordEnumeration recEnum = rs.enumerateRecords( null, null, false );
                // Traverse the Enumeration to get get records.
                while( recEnum.hasNextElement() ){
                        byte[] data = recEnum.nextRecord();
                        enumList.append(new String(data), null);
                }
                // Always destroy Enumeration
                recEnum.destroy();
        } catch (Exception e) {
                e.printStackTrace();
        }

        enumList.addCommand(backCommand);
        enumList.setCommandListener(this);
        display.setCurrent(enumList);
}
```

RecordComparator and RecordFilter Interfaces

As discussed in earlier RecordFilter is to retrieve records with search criteria. This interface supplies only a single function

Public boolean matches(byte[] candidate);

Listing for Filtering records which start with Letter 'A'

```
Public class AlphaFilter implements RecordFilter {
Public AlphaFilter() {}
Public boolean matches(byte[] candidate){
     String strCandidate = new String(candidate);
     If (strCandidate.substring(0,1).equals("A"))
          Returns true;
     Else
          Returns false;
}
```

RecordComparator Interface is used to Compare or sort records in the RecordStore. Like RecordFilter it has only one method.

int compare (byte[] b1, byte[] b2)

Listing to sort records

```
int compare (byte[] b1, byte[] b2)
{
String str1 = new String(b1);
String str2 = new String(b2);
int num1 = Integer.parseInt(str1);
```

```
int num2 = Integer.parseInt(str2);

If (num1>num2)
     Return RecordComparator.FOLLOWS;
Else if (num1=num2)
     Return RecordComparator.EQUIVALENT;
Else
     Return RecordComparator.PRECEDES;
}
```

Program a database trigger

Listener can be set to the Record Store similar to a trigger in a database to monitor Record Store events. Programmatically these Listeners are similar to AWT/SWING/MIDP ICDUI event handling.

RecordListener Interface should be implemented to monitor Record Store events. Following methods should be implemented for various events.

- Record added: void recordAdded(RecordStore recordStore, int recordId)
- Record changed: void recordChanged(RecordStore recordStore, int recordId)
- Record deleted: void recordDeleted(RecordStore recordStore, int recordId)

Before implementing the above methods Record Store has to be set with a Listener using the RecordStore methods.

- Void addRecordListener(RecordListener listener);
- Void removeRecordListener(RecordListener listener);

Implementing RecordStore Listeners

```
// Implementing RecordListener
void recordAdded(RecordStore recordStore, int recordId)  {
        System.out.println(RecordStore  +  recordStore.getName
        +  is modified at Record ID  + recordId);
}
```

j2meSalsa goodies

Execute RMS Demo online	RMS Demo application shows all the essential components of RMS programming. To execute this program select "RMS Demo" application on your left frame.
View Java source code	RMSDemo.java: This program demonstrates RMS basics and RMS enumeration
Download this code for deploying directly to WTK	Click here to download Zip File Containing WTK compatible file structure.

Back | Next

Leave a message...

Best Community Share

poonam • 3 years ago
i have created one program containing rms and trying to create another program which will check if rms exists in the first program or not if exists display some msg
but i m unable code this pls help me out.
6 Reply Share ›

vikrant tanwar • 5 months ago
how to prevent duplicate data in rms database .i am working on rms database so plz help me
2 Reply Share ›

zakirullah • 4 years ago
how to use foreign key in recordstores
2 Reply Share ›

Hosseinousat • 2 years ago
how do you write record have 2 field in rms
1 2 Reply Share ›

Vip_patm • a year ago
thanx sir...
1 Reply Share ›

Syedmohsin087 • 2 years ago
Thanks a lot dude Iam searching for this tutorial can you upload more examples on that
1 Reply Share ›

Hosseinousat • 2 years ago
how ro create rms database in j2me
1 Reply Share ›

Contractornimesh • 3 years ago
how ro create rms database in j2me
1 Reply Share ›

Masdl • 3 years ago
yes u r write
1 Reply Share ›

Reply Share ›

site comments powered by Disqus

This page is a part of a frames based web site. If you have landed on this page from a search engine click here to view the complete page.