Floggy Framework: RMS no more!

This article describes the Floggy Framework, a powerful and flexible framework for providing data persistence on Java ME. Floggy is compared and contrasted with the better known RMS API (Record Management System).

Introduction

15 Nov 2009

RMS (Record Management System) is the well-known API to provide data persistence in the JavaME platform. However, due to its limitations and some complexities to cope with Java I/O, some projects were proposed to facilitate the development of JavaME applications that need to store data. To illustrate how boring RMS is, imagine if you wish to store an object that has a primitive *int*, a primitive *boolean* and a String object. In RMS, an option is:

```
//Saves data
public void save() {
   //Creates data streams
   ByteArrayOutputStream strmBytes = new ByteArrayOutputStream();
   DataOutputStream strmDataType   = new DataOutputStream(strmBytes);

   //Data to be created
   byte[] record = null;

   strmDataType.writeUTF("str"); //writes the String object
   strmDataType.writeBoolean(true); //writes the boolean primitive
   strmDataType.writeInt(10); //writes the int primitive

   strmDataType.flush(); //cleanup buffer
   record = strmBytes.toByteArray(); //gets the array of data
   recordStore.addRecord(record, 0, record.length); //store data in the RecordStore
   ...
}

//Reads data
public void read() {
   byte[] recData = new byte[50];
   ByteArrayInputStream strmBytes = new ByteArrayInputStream(recData);
   DataInputStream strmDataType = new DataInputStream(strmBytes);

   for(int i=1; i<=recordStore.getNumRecords; i++) {
        record.getRecord(i, recData, 0);
        strmDataType.readUTF(); //reads the String object
        strmDataType.readBoolean(); //reads the boolean primitive
        strmDataType.readInt(); //reads the int primitive
   }
   ...
   ...
}
```

As we can see, to save a data using RMS we should create data streams, write each data and store these data in the RMS repository. In addition to these hard-working (imagine more than 10 data types) tasks, in order to read the stored data, we should follow the same steps while reading: that is, if we wrote String, boolean, int, we should have to read in the same order, which can bring many problems.

In order to reduce such complexities, the following APIs are available to abstract the data persistence details from the developer:

- J2MicroDB, an open source database from Universitat Politècnica de Catalunya in Spain
- Perst Lite from mObject

An interesting persistence framework is called Floggy, which provides an object persistence framework for J2ME/MIDP applications. In this post I will show you how to use Floggy in your JavaME applications and how simple it is.

Floggy Framework

Floggy framework is a Brazilian persistence framework for J2ME platform, licensed by Apache License version 2.0. The framework allows the persistence of objects without dealing with RMS low level issues. The traditional RMS (Record Management System) persists only array of bytes, which brings challenges for complex objects storage.

To persist an object through the framework it is only necessary to implement the interface net.sourceforge.floggy.persistence.Persistable. There are many objects supported and some restrictions.

The following category types can be used in the floggy framework:

```
    * All primitive types: boolean, byte, char, double, float, int, long and short;
    * Package java.lang: Boolean, Byte, Character, Double, Float, Integer, Long, Short, String, StringBuffer;
    * Package java.util: Calendar, Date, Hashtable, Stack, TimeZone, Vector;
    * All objects which implements net.sourceforge.floggy.persistence.Persistable;
    * One-dimensional array of any supported type like byte[], Integer[], Date[] or Persistable[].
```

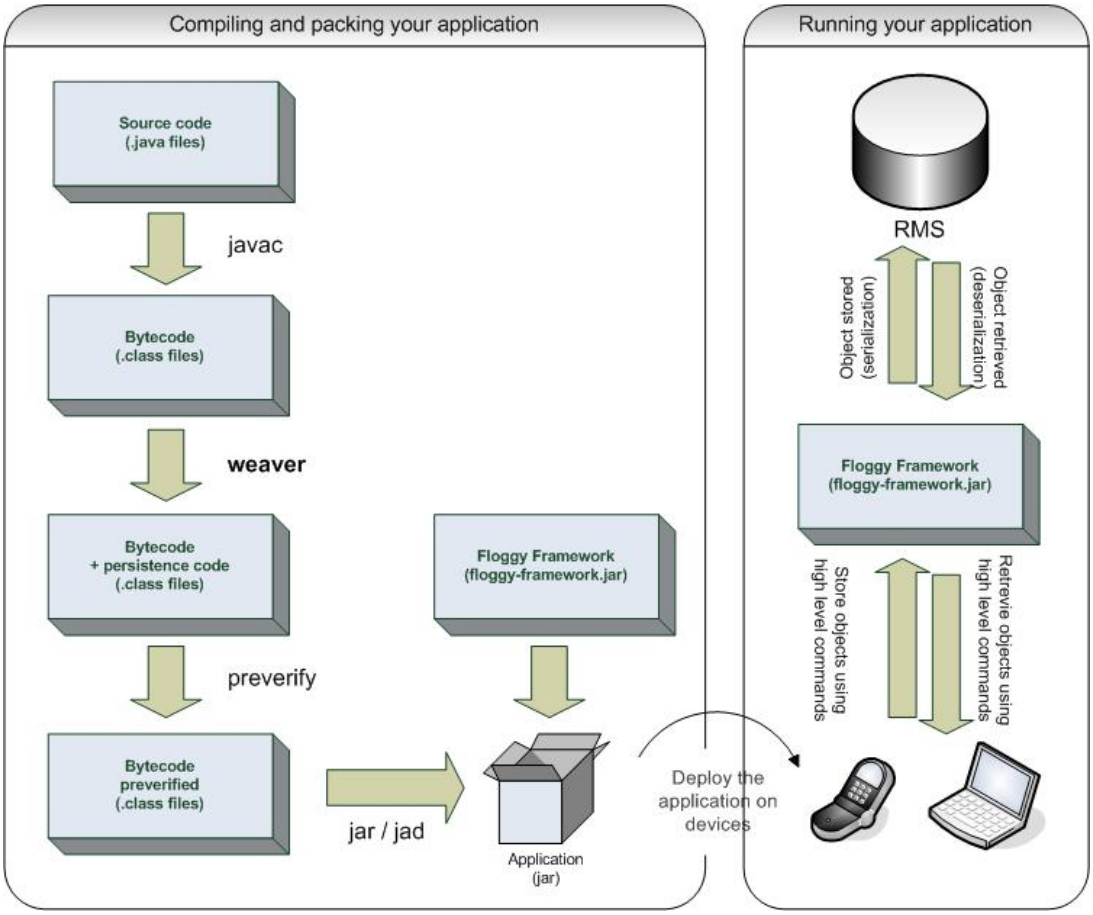
Floggy has the following restrictions:

```
    * Bidirectional relationships;
    * n:n relationships.
```

Floggy Architecture

(C) Copyright Nokia 2013. All rights reserved.

The architecture is divided into two levels: compilation/packaging and application. In the compiling level, the waever component is responsible for receiving the byte-code and creates the persistence infrastructure for each object to be stored. In packaging, the API framework floggy is "pushed" into the application logical. Finally, during the implementation step, the framework Floggy can be used through its methods: savings, removals, object editions and queries, etc.



It's important to point out that the framework creates an abstract interface to manage data in the JavaME applications through the artifacts of RMS mechanisms. That is, JavaME developers do not need to deal with the RMS low level architecture.

Installation in eclipse IDE

To use the Floggy framework in the eclipse IDE is quite simple. The installation can be done in the IDE using the following steps:

```
1. Help >> Software Updates >> Find and Install >> Search for new features to install
2. Next >> New remote Site >> http://floggy.sourceforge.net/eclipse/
```

For more installation details, see: http://floggy.sourceforge.net/

In order to use the framework, click with the right mouse button on the project and select the "Add Floggy nature" option. If you can't find this option, you probably had a problem during the installation process. Perform all steps again.

The following code examples show how to create CRUD operations using floggy. In addition, more examples describe how to create more interesting queries (such as filtering, ordering, etc) using this framework. It is so simple to use floggy: if you wish a class to be persistable, you should only implement an empty interface *net.sourceforge.floggy.persistence.Persistable* or extends a class that implements it.

WARNING: All classes that implement this interface should have a public default constructor like any other POJO (Plain Old Java Object).

Consider the following class, which represents a "Food" and implements the Persistable interface.

```
public class Food implements Persistable {

    private long id; //food's identification
    private String type; //food's type: mass, disert, etc
    private String name; //food's name
    private double price; //food's price
    private String description; //food's description
    private int calories; //food's calories
    private transient char anyField; //transient fields are not persisted!!


    //Like any other POJO, Persistable classes should have a public default constructor
    public Food() {}

    public Food(long id, String type, String name, double price, String description,
            int calories) {
        super();
        this.id   = id;
        this.type = type;
        this.name = name;
        this.price = price;
        this.description = description;
```


(C) Copyright Nokia 2013. All rights reserved.

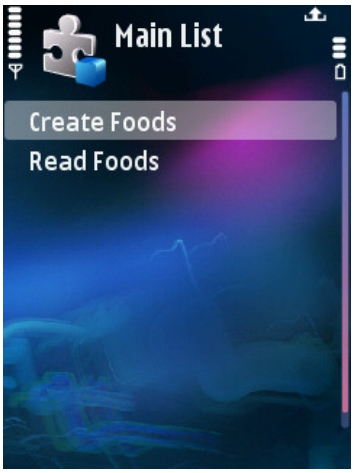
```
        this.calories = calories;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getName() {
        return name;
    }

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

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public int getCalories() {
        return calories;
    }

    public void setCalories(int calories) {
        this.calories = calories;
    }

}
```

Saving objects

The above class is a POJO-based class which will be used in our next examples. First of all, in order to perform CRUD operations (or other complex queries) using Floggy, we should have a reference of PersistableManager class, which manages all process regarding creating, retrieving, deleting or updating an object in the environment. The following example shows how to save a new food.

```
//PersistableManager is a singleton-based class
PersistableManager persistence = PersistableManager.getInstance();
try {
      Food f = new Food(...); //creates a new food object
      int id   = persistence.save(f); //saves object. The "id" refers to the object identification (regarding to the RMS infrastruct
} catch (FloggyException e) {
    ...
}
```

The new food's information can be recovered by a form-based user interface, such as shown in the following Figures.



Retrieving objects

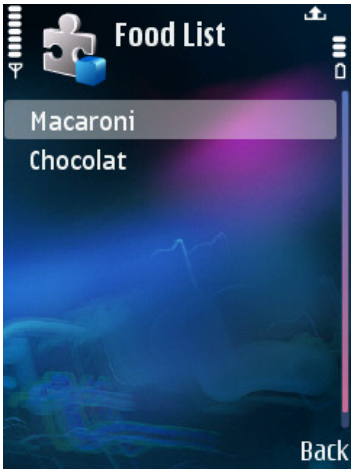
Simple, isn't it? Imagine yourself developing such application and dealing with RMS? At the same way of persisting objects, we can retrieve all of them as simple as the first operation:

```
try {
      Food food = null;
```


(C) Copyright Nokia 2013. All rights reserved.

```
        PersistableManager persistence = PersistableManager.getInstance();//PersistableManager is a singleton-based class
        ObjectSet set = persistence.find(Food.class, null, null); //Creates a set of "Food" references.

        for (int i = 0; i < set.size(); i++) {
                food = (Food)set.get(i); //gets each storaged food reference
                ...
        }
} catch (FloggyException e) {
        ...
}
```

The following figures demonstrate how the "Food" references are retrieved and organized. The list of foods (names) are set to an LCDUI-based List. Each "Food" reference is associated in a HashTable data structure where the hash's key is the index list and it's value is a Food reference.



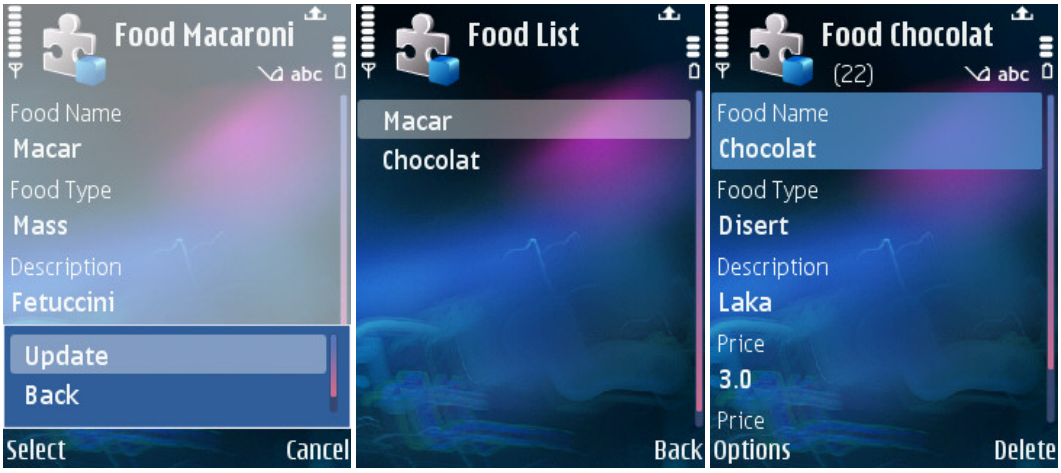
ID (LCDUI List.append())	Food References
0	Food@1cb37664
1	Food@f828ed68

Deleting and Updating objects

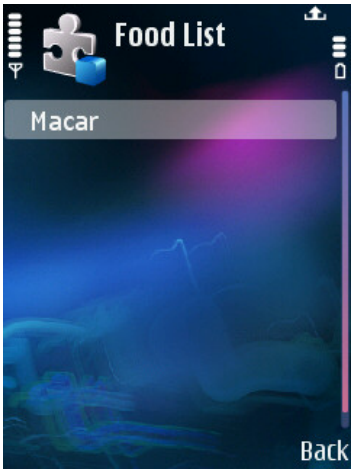
To delete all entities regarding to the same class, we just need to execute: persistence.deleteAll(Food.class); On the other hand, if you wish to delete all data stored by a MIDlet suit, invoke persistence.deleteAll(); If you wish to delete a specific object, you can do the following process:

```
try {
        Food food = null;
        PersistableManager persistence = PersistableManager.getInstance(); //PersistableManager is a singleton-based class
        persistence.delete(foodObj); //Deletes a food which reference is "foodObj"
} catch (FloggyException e) {
        ...
}
```

To update an object, you can get the object (using the second example or by retrieving through the object identification - persistence.get(id, foodReference)), edit it's needed properties, and save it again. The following figures show the delete and update process:



(C) Copyright Nokia 2013. All rights reserved.



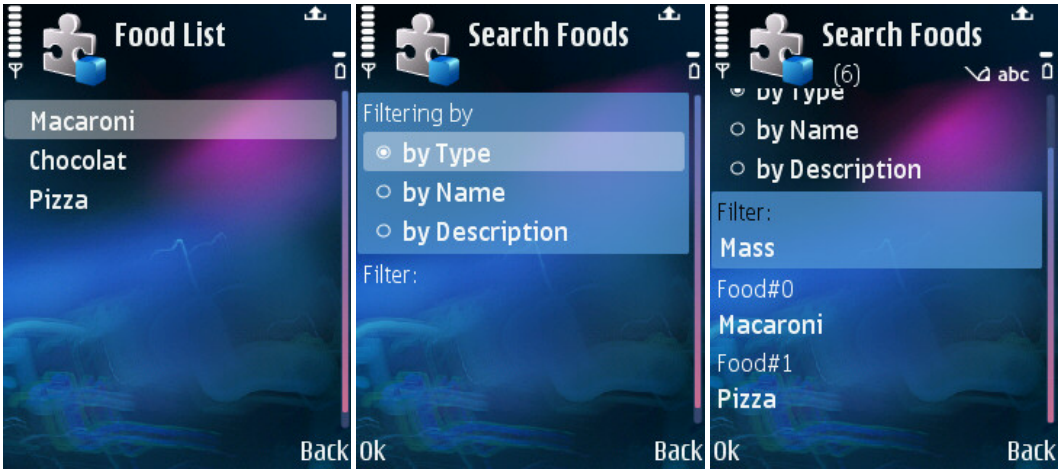
Finding objects

The discussed operations are very simple. However, Floggy also provides mechanisms for filtering and ordering data (such as RMS does) based on classes interfaces. The following examples show how to filter food based on their type, name or description. The filtering process can be achieve by implementing the Filter interface (net.sourceforge.floggy.persistence.Filter). The following filter finds the "foods" which matches some search criteria, such as food's name, type or description.

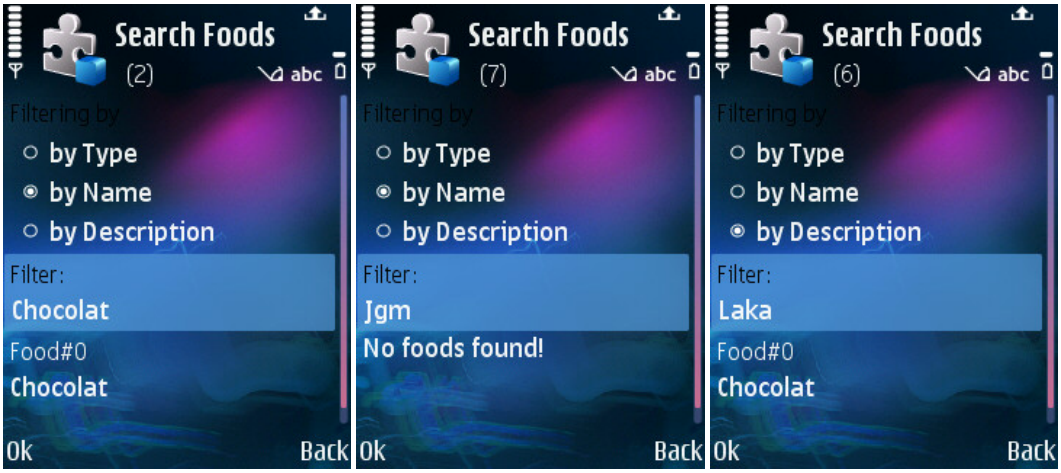
```
public class FoodFilter implements Filter {

    private String type; //which field to filter: food's type, name or description
    private String target; //the array of chars to be matched

    public FoodFilter(String type, String target) {
        this.type = type;
        this.target = target;
    }

    public boolean matches(Persistable arg0) {
        Food f = (Food) arg0;

        if(type.equals("by Type")) //if the filtering is by the food's type
            return f.getType().equals(target);
        else if(type.equals("by Name")) { //if the filtering is by the food's name
            return f.getName().equals(target);
        } else if(type.equals("by Description")) //if the filtering is by the food's description
            return f.getDescription().equals(target);
        else //returns false otherwise;
            return false;
    }

}
```

The following code demonstrates how to list a subset of foods based on some search criteria:

```
//creates a filter with type "type" and the string to be matched ("target"). The "set" object contains the subset of foods which mat
ObjectSet set = persistence.find(Food.class, new FoodFilter(type, target), null);
```



(C) Copyright Nokia 2013. All rights reserved.

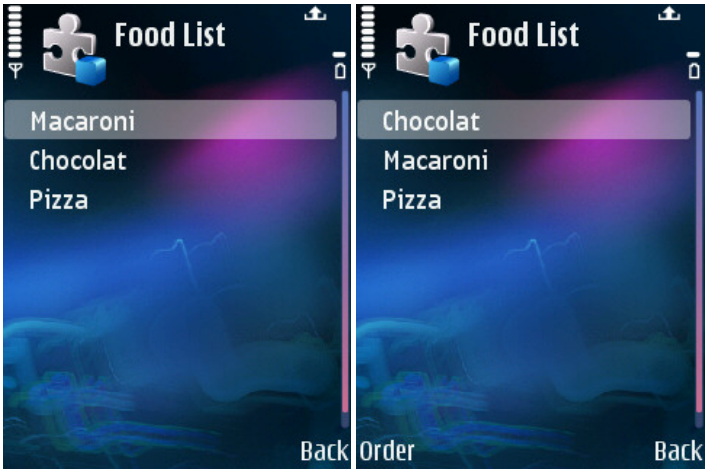


Ordering objects

As simple as filtering a subset of objects, Floggy framework also provides an interface (net.sourceforge.floggy.persistence.Comparator) for ordering data based on the foods' properties. Our example orders the set of foods based on their names:

```
public class FoodOrdering implements Comparator { //Implements net.sourceforge.floggy.persistence.Comparator interface

    public int compare(Persistable arg0, Persistable arg1) { //arg0 and arg1 will be compared
        Food f1 = (Food) arg0; //cast to the "Food" class
        Food f2 = (Food) arg1;

        if(f1.getName().compareTo(f2.getName()) < 0) //checks if f1 preceds f2
            return PRECEDES;
        else if(f1.getName().compareTo(f2.getName()) > 0) ////checks if f1 follows f2
            return FOLLOWS;
        else
            return EQUIVALENT; //otherwise, f1 and f2 are equals.

    }

}
```

The following figures show a disordered food list and an ordered food list (after pressing the Order command), respectively.



Although providing an interesting API to store data in the JavaME platform, RMS has many limitations regarding development complexities, such as Java I/O restrictions when adding and reading data. On the other hand, the Floggy framework is a good candidate to provide fast and easy data storage mechanisms in JavaME applications, reducing time development and software maintenance by abstracting the low-level Java I/O requirements that developers should cope with.

Downloads

Media:Floggy-Examples.zip: Examples described in this post

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.