Mobile part documentation

1. Submitted mobile part

CD Content

In mobile directory on submitted CD there are things related to mobile part of mujMail project. In this section we briefly description content of this folder.

api_ref.pdf	Contains packages and classes description.
documentation.pdf	This document.
javadoc	This directory contains generated mobile part javadoc.
jar	Directory contains precompiled version of mujMail.
src.zip	Packed sources of mujMail.
user_manual.pdf	User manual. Describes basic mujMail behavior.
user_manual_CZ.pdf	Czech localization of user manual.

2. Installation

Installation into device can be done in many different ways and is vendor specific. In general most easies way is to use web browser inbuilt in mobile phone, go to our web pages and get download jar file. Mobile phone typically recognizes that you are downloading java application and add it into java application list in you phone.

For Nokia phones you can build or download mujMail onto your computer and use Nokia PC Suite to install in into your mobile device.

For Sony Ericsson mobiles it typically enough send it into mobile phone as ordinary file. Use file manager and browse to downloaded mujMail image and choose install to add mujMail into java application list.

3. Compilation

Prerequisites

If you wanted to start developing mujMail you have to install all needed tool to be able write, run, test and debug mujMail. We strongly recommend test newly added feature on real mobile and not only in installed emulator.

Before you start to work you need to have Java JDK installed, NetBeans IDE and Java Wireless toolkit. Our team uses both NetBeans and Eclipse for development, in this manual we cover only NetBeans settings.

You can download Java Development Kit from Sun's web pages. Use this quick link http://java.sun.com/javase/downloads/index.jsp and download current version of JDK. Install it.

Download and install NetBeansIDE. You can use this quick link http://www.netbeans.org/downloads/. Select version that contains JavaME. It means Java or All editions. This version should contain all necessary tools what you need to start development.

Of course you need to have sources of mujMail. Easies and most suitable way is use our public accessible svn repository. Install some SVN client (TortoiseSVN) or use command line utility. Repository address is
https://javaphone.svn.sourceforge.net/svnroot/javaphone/mujMail/src-mobil

Create NetBeans project

Next step is run installed NetBeans IDE and create project.

1. File → New project → Mobility → MIDP application → Next
2. Tick off Create Hello MIDlet → Next
3. Device configuration: cldc 1.1, Device Profile: midp 2.0 → Finish

Copy all source codes to the directory src of the new project.
Set up the project properties to be able to run mujMail.

Right click to the project → Properties

1. Platform: Optional Packages
   tick off all except File Connection and PIM Optional Packages (4. item in NB 6.5) and Mobile Media API (9. from the end)
2. Application Descriptor
   Attributes: General Attributes for JAD and Jar Manifest:
   midlet-name: mujMail
   midlet-Vendor: Students of www.mff.cuni.cz

version: 2.xx

Application Descriptor → MIDlets → Add:

midlet-name: MujMail-dev

midlet class: mujmail.MujMail

midlet icon: /icons/logo.png

If you want to run unit tests, add unit also the midlet that runs unit test.

Application Descriptor → MIDlets → Add:

midlet-name: Unit tests

midlet class: test.TestingMidlet

3. Build → Sources Filtering

   Untick folders lib and doc

   If you want not to run tests, untick also folder test

4. Build → Compiling:

   tick Compile with Optimalization

   Encoding : UTF-8 (for stable version: Cp1250)

5. Build → Libraries and Resources:

   Add Jar/Zip → Add both 2 libraries from ./src/lib

6. Build → Obfuscating:

   For production version testable on mobile phone set obfuscation level to high.

   For testing or development version (on emulator) set obfuscation level off.

7. Build → Creating JAR:

   Tick Compress Jar

Press OK

From now you should be able compile and run mujMail in emulator.

4. API Documentation

In this part we want briefly describe overall mujMail architecture. Knowledge of this structure is essential to understand how thing works (or goes wrong ☺). We will in a use case describe thinks influents. In the end be describe in more detail features we add into mujMail.

mujMail consists from these essential parts.

- **MailAccounts** – Mail accounts hold information about mail server. Server address, port, login name, password, used protocol and others data are stored here. Mail accounts are represented by MailAccountPrimary class.
- **Email representation** – Email is not represented by one object. We represent mail in structured way. We divide mail into header and body parts. Body part can represent an attachment, default mail text that is shown to user, in mail stored images, forwarded email etc.

Database storage – Is set of classes that are responsible for storing downloaded emails parts into persistent record store databases and for loading. Classes that solve this problem are MailDB, RMSStorage, ContentStorage.

- **Boxes** - Subchilds of TheBox class. Is responsible for showing list if mails to user.
- **Protocols and mail parsers**. Takes place in InProtocol, POP3 and Imap4 class. These classes are responsible for communication with mail servers. Sends protocol commands and serves replies from server are stored
- **Settings** – This class holds information about SMTP server and other global settings.
- **Sending mails** – Sending mails is mainly influenced by 3 classes. SendMail class that shows form were email can be written, MailSender that transforms mail from form into textual representation (Makes mail header and mail body) and last SMTP class thet is responsible for managing SMTP commands (opening connection and login, list of recipients,
- **Viewing mails bodies** – Class MailForm is shows to user mail content on display.
- **Exception handling** – to extend standard exception capability, new MyException is introduced. For showing alerts to user there is MyAlert class that makes easy to show different alerts to user.

Around this necessary parts there is many different classes used for some specialized purposes. And you can understand to the core of mujMail without knowledge of them.

Now we focus on actions that happen if you start retrieving new mails (headers). Briefly say mujMail does this thinks:

1. Take mail accounts one by one
2. Make connection to server.
3. Log in and send commands to obtain new server contents.
4. Sends retrieve command for each new email
5. Parse server replies and create new MessageHeaders according sent information.
6. Store MessageHeader into persistent database and adds them into Box.
7. Set focus into target box, where new emails come.

And now if we have overall overview about thinks that happened look on start procedure start in details.

1. Execution starts in commandAction (action invoked by pressing buttom) handler, that calls retrieve procedure on selected Box - typically Inbox folder. Here we test if there is at least one account available for downloading mails. For each such account we obtain from account protocol object that can communicate with server and call getNewMails on them.
2. getNewMail method creates new StoppableTask which will download new header from server. This creating of new thread is needed, because we can't communicate with server in main thread. Communication can hang and in that case would mujMail hangs too.
3. New created background task locks. This locking in needed to ensure that no one else sends data. (It could be ConnectionKeeper object, or some other running downloading task). This is needed because mujMail reuse one connection for more communication. It brings some design problems, but it saves time, because opening new connection is relative slow process with many roundtrips.
4. mujMail try to open connection. If not open from previous session it open connection (you can imagine connection as both directional pipeline to server) and sends login information

…..

In rest of this chapter we will focus on functionality added by our team. We have done a lot of code refactoring and some in core mujMail functionality, but these changes don't modify original meaning. He hope that all our changes makes code better manageable. Changes are more detailed described in included javadoc. This part only gives you only brief overview how thinks work and what do, to be able understand javadoc and our implementation goals.

Mail accounts

Mail accounts store information about email servers and information needed for connecting to them. Mail accounts are stored in mujMail object and can be retrieved by mujMail.getMailAccounts().

There are two types of mail accounts. Primary accounts are stored in persistent database called ACCOUNTS, are primary source of information about account. These accounts are visible for user. Derived accounts are used only by user folders to customize primary accounts behavior and are invisible to users.

Class AccountSettings is responsible for managing accounts. It loads accounts during mujMail startup, create new account, save them, shows form for editing settings.

Class Sync is responsible for backuping and restoring accounts and others configuration settings. Synchronization can be invoked by calling Sync.startSyncDlg(java.lang.String).

Connections

Connections contain (network) transmission related staff - code for connections management, encryption, compression and buffering. Connections are both directional.

We have two important interfaces here.

ConnectionInterface is outer interface that uses all other components of mujMail (mainly in protocol parsing classes). Creates illusion of line oriented buffered input and string accepting output.

ConnectorInterface is internal interface to be used only in connection package. Interface is intended to create unified vision on different streams for sending data (into file, over network, GZip compression). It's the most low level communication component. Most typically is used for communication with mail servers.

The most important class that is used outside this package is ConnectionCompressed.

Debugging support

We add basic support for debugging. We are able to (persistently) store textual information. Debug package provides capability for storing and viewing debug information on mobile device. By default information are copied to standard output and stored in vector to be able display them to user.

DebugConcole class is used for writing debug information. Provides line string oriented interface for programmers to print debug in information. To print debug log call DebugConsole.println(String) function.

DebugConsoleUI shows debug information in special form.

Note: Persistent storing of debug information is useful for on device debugging if application falls.

Basic touch screen device support

Outside from official specification we add basic support for touch screen devices. Because we don’t use only standard Java forms, mujMail doesn’t work on devices without keyboard.From user point of view, tapping on different part of screen is transformed into standard keyboard events like moving down or up arrow. We add this functionality because users want uses mujMail in such phones and call for this.

We transforming pointer events produced by touchscreen to higher level events. There are currently implemented higher level events that emulates keyboard.

Adding touchscreen support to custom user interface class

To listen to mujMail high level pointer events define class implementing interface MujMailPointerEventListener and create object of this class. Than, create object of class MujMailPointerEventProducer. Pass the instance of MujMailPointerEventListener created in previous step to the constructor of this object. Than, J2ME low level pointer events must be catched - to do this overload methods Canvas.pointerPressed(int, int), Canvas.pointerDragged(int, int) and Canvas.pointerReleased(int, int) and call appropriate methods of newly created object. Now, every time when some mujMail event is produced, appropriate method on MujMailPointerEventListener is called

To implement interface MujMailPointerEventListener, adapter class MujMailPointerEventListenerAdapter should be used. This prevents compilations errors when some new method to interface MujMailPointerEventListener is added. Than, it does not force user to implement all methods of this interface it is not needed.

Creating new pointer events

To create new pointer event, subclass class MujMailPointerEvent. In method MujMailPointerEvent.handleEvent() the event should call some method on listener passed as the argument of this method. If it should be some new action, new method to interface MujMailPointerEventListener must be added.

Than the event must be produced. That is why methods in class MujMailPointerEventProducer producing events - that means for example method MujMailPointerEventProducer.transformPointerPressed(int, int) - must be changed in order to produce new event. Another possibility is to create descendant of MujMailPointerEventProducer that will override such methods and than use object of this new class to produce events instead of object of class MujMailPointerEventProducer.

Searching in mails component

One point in our specification was support searching in mails. This task is done by this quite separate part of mujMail. We refractor Box classes while create search component. We divide Box functionality a modify hierarchy of classes. Boxes now are divided into persistent which stores data in RecordStore databases and nonpersistent that are used for storing search results.

Classes in this package can be divided into two groups:

- User iterface classes that enable user to enter search settings or display search results.
- Classes used to find messages that fulfill given criteria.

User interface

All user interface functions are accessible using static methods of class SearchWindows. It provides methods both for displaying dialog where user can enter search settings and for displaying results of the search. Settings dialog remembers message parts selected to search in and boxes selected to search in SaveableSelectedState and WasSelectedReminder. The results of the search are displayed using class SearchBox.

Each matched message contains information about occurrences of search phrases found in the message. This information is accessible via method MessageHeader.getSearchResult(). This information can be later used for better displaying of search results in SearchBox.

Running search

Searching is run by executing method SearchCore.search(mujmail.search.SearchSettings, mujmail.search.SearchBox, mujmail.tasks.StoppableProgress).

Search algorithm

Method SearchCore.search() enumerates all messages in all boxes contained in search settings. If the message matches search criteria specified in search settings, it is added to search box.

The message matches if the date of the message is in interval specified in search settings and if the message matches given search phrases.

Search phrases are represented by class SearchPhrase. This class provides method SearchPhrase.findFirstMatch(mujmail.MessageHeader) that finds first occurence of the phrase in given message. This method lists all message parts in that the phrase should be searched and search the phrase there.

Message parts that should be searched in are represented by class SearchMessagePart. This class provides method SearchMessagePart.findFirstMatch() that finds first match of given search phrase in this message part. This method is abstract. This means that every concrete searchable message part must implement searching itself.

Fulltext search and search modes

Most of searchable message parts uses fulltext search in string. Object that provides fulltext searching is accessible by calling method SearchCore.getFulltextSearcher.

Method FulltextSearcher.searchInString() uses instance of class FulltextSearchAlgorithm for searching the string and FulltextSearchModes stored in SearchPhrase to check whether the location of the string matches the FulltextSearchModes. This means that the location of string meets given condition - for example it is whole word etc.

Implementing new fulltext search algorithm

To implement new algorithm for fulltext searching, implement the interface FulltextSearchAlgorithm. To make new search algorithm used while searching, create instance of class FulltextSearcher with new search algorithm as the parameter and make method SearchCore.getFulltextSearcher() to return this instance.

Tasks

Tasks provide uniform stoppable framework for execution actions. Task can operate on background and can be stopped. Tasks usually have progress bar and changeable title. Task stopping works on polling base. That is why J2ME doesn't provide resources for threads killing. Provides classes and interfaces for creating, running and managing tasks.

Basically, task is an action that is runned in a new thread and that is registered before execution and unregistered after the action is done. This enables various management actions of such tasks. Tasks also support displaying a progress of the action to user. User can cancel displaying this progress by pressing Back button. User can also see all running tasks and their progresses in Tasks manager. If the task is descendant of StoppableBackgroundTask, progress contains stop button that enables to stop the task.

Package level classes ProgressManager and StoppableProgressManager provides user interface for displaying of progress of BackgroundTask respectively StoppableBackgroundTask.

Defining tasks

To define new task, create new class that inherits either from BackgroundTask or StoppableBackgroundTask if the task should be stoppable. Than write the action that should be performed in the task in method BackgroundTask.doWork() that is abstract in parent class.

It is not possible to stop task preemptively. That is why cooperative multitasking must be used. If the task is descendant of StoppableBackgroundTask, it must control whether it should terminate. StoppableBackgroundTask implements interface StoppableProgress that contains method StoppableProgress.stopped(). If this method returns true, the task should terminate.

While running the method BackgroundTask.doWork(), it is possible to display progress of the action to user. Methods of interface Progress that StoppableBackgroundTask implements can be used to do this.

When some method of object outside BackgroundTask or StoppableBackgroundTask is called from and it makes sense to display progress to user or to stop the task during the execution of such method, such method should have parameter of type Progress respectively StoppableProgress. The method will be than called with this in the place of such argument. The method can than use methods of such interface to display the progress to the user or to terminate itself.

It is also possible to disable displaying progress of task to user by calling method BackgroundTask.disableDisplayingProgress() before starting the task.

Starting tasks

To start the task call method start. Before the task is started, it is checked whether number of running tasks of the same class is less than given limit. If it is not possible to start the task immediately, the dialog where user can cancel running the task is displayed. It is possible to disable displaying this dialog by calling method BackgroundTask. disableDisplayingUserActionRunnerUI() before starting the task. The task is placed between waiting tasks and it is started when the number of tasks of the same class is less than limit.

To start the task immediately, without the check if there are less tasks of the same class than given limit started, call method BackgroundTask.disableCheckBeforeStarting() before starting the task.

Observing tasks progress

It is possible to register receiving notifications every time the progress of the given task is changed. The list of events that are received is described in enumeration class TaskEvents. Classes BackgroundTask and StoppableBackgroundTask are descendants of class Observable that provides methods for registering and unregistering objects that want to listen to these events.

Managing tasks

There is class TasksManager for managing tasks. Class TasksManagerUI provides user interface for some task management features such as displaying list of running or waiting tasks or displaying progress of running tasks.

Running actions on task start or task end

It is possible to receive notifications every time some task starts or terminates. To register for receiving such events there are methods

- TasksManager.addEndTaskObserver(mujmail.util.Observer)
- TasksManager.addStartTaskObserver(mujmail.util.Observer).

Than, it is possible to check given condition every time some task starts or terminates and if it is true, run given action using class ConditionalActionRunner. For displaying user interface to users in case that it is possible to use class ConditionalActionRunnerUI.

HTML processing

MujMail is able to process emails with HTML bodies. Specific J2ME environment gives us conditions that implementation should care about. HTML code stored in email is typically much less complicated than ordinary web pages so lightweight parser should by enough.
Most code responsible for implementation of HTML takes place HTML package, where parsing takes place. Second influenced part is the MailForm class, where some modification for drawing HTML has been made.

Parser class is responsible for parsing HTML stream. There are some requirements for Parser functionality:

- it should be the as simple as possible in meaning of robustness - we do not require to handle every error in HTML stream (like missing closing tag, incorrectly paired tags). Parser is not validating the input just wants to highlight some subset of HTML tags
- parser have to process incomplete HTML source. It's available in mujMail to limit number of downloaded bytes/lines of e-mail so generally we cannot expect correctly paired tags not even closed (HTML can ends with "

When stream is parsed, vector of elements is returned.

For drawing of these elements we need just to call draw method as we assume that vector of elements is vector of instances implementing Drawable interface. Each supported HTML element is represented by separate class. These classes are stored in mujmail.html.element package and main task of these classes is draw element onto screen.

User mail boxes

User mail boxes serve as new separate storages for emails. Use can specify which account to retrieve into selected user mail box. Main part of implementation takes place in mailbox package which currently takes care only about user folders.

Main part of work does BoxList class. BoxList manages user boxes. Holds list of user boxes. Take care about basic boxes operations like creating, loading, removing.

User folders are implemented as InBox instances. Theirs names are stored in special database. Each user folder has (and also loads and saves) list of account's to retrieve. See InBox.saveBoxRetrieveAccounts().

Mails threading

Mail threading is special type of sorting emails where mail logically corresponds are grouped together. We support basic type of threading that is equivalent to gmail conversations. In general mails dependency creates tree structure, we decide not to show this tree structure that could be rather deep. Instead we have only 1 level of depth. Main reason for this decision is small

mobile display, where more level hierarchy consumes worth display place, where more important information about sender, data and subject can be found.

With treading we refractor runtime mail storing system. Original mujMail uses vector where mail stored in this box takes place. This structure isn't suitable for threading. So we introduce new interface IStorage, where typically used methods take place. From most notable are add and get i-th component and sort. One implementation of this public interface is class ThreadedEmail, that uses internal structures to hold threading interfaces and implements over them this mail storing and accessing interface. Class contains vector of root messages and for each root messags it contains Vector of child messages. Child messages vectors are stored in hash map, in which root message ID is the key. ThreadedEmails structure is used also for other boxes too (f.e. Outbox, Draft ...). In these boxes we do not need threads, so in these cases only root messages are used and, for memory saving, there are not empty vectors, but nulls.

Most important part of threading system is Algorithm class where sorting mails into separate threads take place. Algorithm is based on description written by Jamie Zawinski at http://www.jwz.org/doc/threading.html

JSR75

This package represents low level layer for accessing into file system. (This abilities are described in JSR75 extension) Detection if device is able to work with file system is done during application startup. This runtime detection prevents us from creating separate versions for special devices that doesn't support file system access.

Implementation of this low level API is done in jsr_75 package. This package performs dynamic loading of objects of JSR75 classes on devices where JSR75 is available and using objects of dummy classes on devices where JSR75 is not available. This is necessary because if some JSR75 class is directly imported, the application cannot run on some devices without JSR75.

The solution of this problem is to import given JSR75 class to the wrapper class that is not imported to any class in the midlet but loaded by Class.forName method. If the loading of the class is not successful, it means that JSR75 is not available and it is used dummy class. Both dummy class and wrapper class implements given interface and are casted to such interface. For implementation details see class FileSystemFactory. Another solution is to use preprocessor.

The examples are classes MyFileConnectionJSR, MyFileConnectionDummy and interface MyFileConnection. Interface MyFileConnection is interface which both two classes implements and that is used for working with them. MyFileConnectionJSR is wrapper class which imports JSR75 class javax.microedition.io.file.FileConnection and uses this class to perform filesystem operation. Method getFileConnection of class FilesystemFactory tryes at first load load class MyFileConnectionJSR. If it is successful, the object is created and casted to MyFileConnection. If the exception is thrown, it is loaded class MyFileConnectionDummy with dummy operations.