Messaging

Messaging Architecture

Messages may be created, sent, received, stored, manipulated and deleted. This feature is known as the messaging architecture.

There are a number of useful things you may want to do with messages from within your own application, such as:

- Create and send a message.
- Perform some action when a new message is received.
- Manipulate an existing message, or group of messages: for example, searching through the body text of all received messages.

S60 includes three key APIs for performing these kinds of tasks:

- The **Client MTM** API.
- The **Send-As** API.
- The **CSendAppUi** class.

Key Messaging Concepts

This subsection introduces some of the basic concepts of S60 messaging that underlie all three messaging APIs.

Messaging Server and Session

The Messaging architecture builds around a server, which manages all messaging resources on a phone. This is accessed from a client process through an instance of the session class CMsvSession. Any client process that uses messaging services must have at least one instance of this class. In almost all cases, one instance is all it needs, and this can usually be shared by reference between all classes that require access to the messaging server.

Messaging Entries

The data managed by the messaging server takes the form of a collection of entries. An entry can be one of four different types:

- A folder.
- A message.
- An attachment.
- A service.

An entry can have child entries and can itself be the child of another entry (its parent). In this way, a tree structure is built up which can be thought of as analogous to a file system with its folders, subfolders and files. At the root of the tree is the **root index entry** (to continue the file-system analogy, the root index entry plays a role like that of the root folder of a drive). This contains four standard folders: *Inbox*, *Outbox*, *Drafts* and *Sent Items*, which may in turn contain any number of messages and/or user-defined subfolders. Each subfolder itself may contain messages and/or further subfolders, and so on. A message may have attachments as child entries or, less typically, other messages.

MTMs: Message Type Modules

An MTM is a plug-in to the messaging architecture to provide support for a specific message type. It comprises a server-side DLL which interacts directly with the messaging server, and a suite of client-side DLLs which provide an API for client applications.

Generic Entry Handling/Unified Inbox

The messaging architecture is designed to allow all types of entry to be treated generically, as far as possible. Just as most filing systems allow you to, say, delete or rename a folder in exactly the same way as you would a file, the S60 messaging architecture allows you to use exactly the same APIs to perform most routine management tasks on any kind of entry. These tasks include deleting, moving, copying, navigating to parent and child entries, and finding details such as size and date. All of these tasks can be carried out without knowing (or caring) about the type of the entry.

The main classes used for this generic entry handling are CMsvEntry, TMsvEntry and CMsvStore.

Entry Storage

The Messaging Server is responsible for storing all types of entry and providing concurrent client processes with safe, shared access to them. The server MTM component interacts with the Messaging Server to handle sending and receiving of messages.

The Messaging Server can store information about each entry in three different locations: the messaging store, the messaging index and the file system.

The Messaging Store

Each entry has a file store associated with it, used to persistently store its in-memory representation. The format of an entry's data in the Messaging Store depends on its type: note that there is no generic format for Messaging Store data. Message entries use it to store things like body text (where present) and headers (with the format dependent on the specific MTM). Service entries use it to store all their configuration information. Folder entries do not use the store (note they still have a store, but they leave it empty), and attachment entries may use it, based on the individual MTM implementation.

An entry's store is accessed using the CMsvStore class.

The Messaging Index

For every entry, regardless of its type, the Messaging Server maintains a generic set of summary information in the Messaging Index. The index is loaded into RAM when the Messaging Server starts, and it stays in memory until the server closes, so it provides a quick way to access information about an entry. Note that the index does not store all information about an entry—only some generic information. For example, it does not store the message body, but it does contain information about the size, date and type of the entry, its unique ID, and so on. The idea is that the index entry contains enough information about an entry to display a summary of it (for example, in an Inbox view) without opening a file store or needing to load an MTM. This enables summary views to display a list of messages quickly.

The index entry is represented by the TMsvEntry class.

The File System

Finally, each entry is assigned a specific folder in the file system, where it may choose to store further data. As with the store, this is optional and its usage is MTM-specific. Use of it tends to vary widely between MTM implementations, and the folder is typically only accessed directly by the Server MTM. As an application developer you will almost certainly never use it, but you will see APIs which refer to it (such as CMsvEntry::GetFilePath()), so it just helps to be aware of it.

Key Messaging Classes and Data Types

This section presents a quick reference guide to some of the most commonly used messaging types.

CMsvSession

Represents a client-side session to the messaging server. Note that it is a C-class, not an R-class as you may have expected. Normally one instance per client thread is enough.

TMsvId

Simply a typedef of a TInt32. The messaging server assigns a unique TMsvId to each entry. In most cases this is performed dynamically, but there are a few fixed IDs—for example, for the standard folders (Inbox, Outbox, Drafts and Sent). Once you have an entry's ID you can use it to obtain any other information about the entry, including its index entry and store data.

TMsvEntry

Represents an individual index entry. As mentioned above, the index entry contains a restricted, generic set of summary information for an entry.

CMsvEntry

This is perhaps the most commonly misunderstood class in the messaging architecture. Perhaps because of its name (which looks misleadingly like TMsvEntry), it is often assumed that an instance of this class contains all the data belonging to an individual entry (such as its headers, body text, and such like). This is not the case, as you will see as soon as you look at its API. CMsvEntry is not an in-memory representation of an entry—likewise, calling CMsvEntry::NewL() does not create a new entry in the messaging server.

Instead, CMsvEntry is better thought of as an entry handle or entry context. CMsvEntry provides you with an interface onto a specific entry, through which you can obtain other objects which contain the entry's data. As such, a CMsvEntry can be reassigned to "point to" a different entry, simply by providing the new entry's ID. In fact, this is how a CMsvEntry object should be used—it is an expensive class to instantiate, so it is recommended that you "recycle" CMsvEntry objects wherever possible and avoid needlessly creating new ones.

CMsvStore

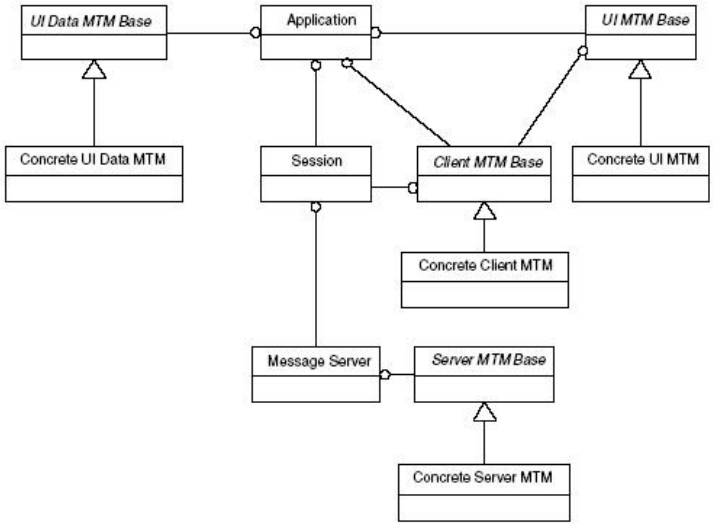
CMsvStore represents a store for an individual entry. Its API can be used to store and retrieve the body text of a message (provided it has one) and other data (such as headers).

CMsvEntrySelection

This is just an array of TMsvIds, and as such provides a common way for passing around information about groups of entries. For example, methods such as ChildrenL() in CMsvEntry (used to enumerate an entry's child entries) return a CMsvEntrySelection pointer.

CMsvOperation

Many long-running activities in messaging, such as sending all items from the Outbox, or checking for email on a POP3 server, require more than an active object to handle them. As well as being notified that the request has completed, applications frequently need to be able to get progress information while the request is still running (for example, in order to update a progress dialog of the form "Downloading message x of y"). As a result, most asynchronous functions in messaging, as well as taking a trequestStatus& parameter identifying an active object which will handle their completion, also return a CMsvOperation object which can be used to obtain progress information before the request completes.



Internal Links

Message Type Modules