Adding new filters to KmailCvt

Author: Hans Dijkema
Email: h.dijkema@hum.org

Abstract

KmailCvt is a utility to import (and maybe later export) various formats in Kmail and Kab. This document describes how to add new filters to KmailCvt. KmailCvt has been written in C++ and uses the KDE and Qt framework.

The Filter Framework

KmailCvt is essentially nothing more than just a simple dialog that displays a list of (import) filters you can choose from. I've constructed a framework to make writing filters easy for programmers who want to contribute a new import format to KmailCvt. The following classes are used:

class filters

This class is a container class for the known filters. It provides a function **void add(filter *)** to add a new filter to this container. That's all that's interesting to the programmer. In the file **kmailcvt2.cpp** there's a function **void doFilters(void)**. In this function the programmers add the new filters. Example:

imports−>add(**new** filter_pab)

You see, it's really easy. When the user choses a filter from the dialog, **filters.import()** is called. This function selects the corresponding filter class and calls the import function of this class. Viola, this is the framework to the dialog.

class filter

Well, this is the class from which every filter is derived. It inherits functionality from **class kab** and **class kmail,** which provide API's to kmail and kab. Class kab and class kmail are described later on. This base class provides the following functionality:

filter(**char** *name, **char** *author)

This is the constructor function of the filter. Each filter that is derived from class filter, should provide a name to the filter. This name is added to the list of filters in the dialog box. It's also used to construct a list of filters in the about box. This is where the author is used. Note: as KmailCvt2 is GPL2, each filter should also be GPL2.

virtual void import(filterInfo *info)

This function is the actual important function. It's the function that is called by

filters.import(). Note: if one writes an export filter, this function can of course be interpreted as 'export'. The framework has no way to determine if something is import or export. It is a virtual function with a default functionality to alert that no function has been implemented. When you derive a class from class filter, you have to implement this function.

Example:

```
filter_pab::filter_pab() : filter(i18n("Import MS Exchange Personal Addressbook
(.PAB)","Hans Dijkema")
{
(...)
}

void filter_pab::import(filterInfo *info)
{
(...)
}
```

See filter_pab.cxx.

class filterInfo

Class filterInfo provides an API to the framework to deliver information about the conversion that takes place. Always use this class to output your information. This makes it transparent wether, you're filter is 'plugged into' KmailCvt or an other e.g. console application. It has the following functionality:

One liners

void from(**const char *****from)

When you call this function, you should provide the object that you are currently converting from. E.g. "map1.dbx". It's wise to add ""from:\t"" in front of your from string, for transparency to the user.

void to(**const char *****to)

The same as from(), except this should be the object you are converting to.

void current(**const char *****current)

The same as from(), except, this should be the object within 'from' that you are currently converting.

Example

'from' could be the directory with OE5 folders.
'to' could be the kmail directory.
'current' could be the currently converted .dbx file.

Messages

void alert(**const char *****caption,**const char *****message)

Use this function to alert a message to the user, e.g. an error message or an information box. Caption is the title of the message (usually your filtername). Message can contain '\n', so multiple lines are possible!

Logging

void log(**const char *****log)

This function provides a way to log information to a conversion log. In the framework, this conversion log is a list of lines. You can log one line at a time, so don't use '\n'!

Progress indication

void current(**float** percentage=−1.0)

This function provides a way to indicate the progress of the current subitem that's being converted. When called without arguments (current()), it clears the progress indicator.

void overall(**float** percentage=−1.0)

This function does the same as the current() counterpart, but it indicates the progress of the overall process.

Other functions

void clear(**void**)

This function clears the info block in the dialog, it's like a clear screen.

Qwidget *****parent(void**)

Use this function to give the parent window to a dialog (e.g. a file open dialog) that you are using in your import function.

Example:

choosen=KFileDialog::getExistingDirectory(dir,info−>parent(),"ImportOE5");

class kmail

Class kmail is a parent of base class filter. This class provides the API to kmail. It has following functionality:

bool kmailStart(filterInfo *info)

Opens a connection to kmail, to be able to add messages.
= true, succeded making connection.
= false, otherwise ==> **abort conversion.**

Void kmailStop(filterInfo *info)

Closes the connection to kmail.

void message(filterInfo *info,**char ***folder,**FILE ***msgIn)

Adds message 'msgIn' to mail folder 'folder'. If mail folder 'folder' does not exist it will be created. If 'msgIn' already exists in mail folder 'folder', it is not added (note this may not work, as long as there's no API to kmail from the kmail group).

class kab

Class kab is a parent of base class filter. This class provides the API to kab. It has following functionality:

bool kabStart(filterInfo *info)

Makes a connection to Kab.
=true, succeded making connection
=false, otherwise ==> **abort conversion.**

void kabStop(filterInfo *info)

Close connection to Kab.

void kabAddress**(**filterInfo *info, const char *adrbookname,
char *givenname, char *email=NULL,
char *title=NULL,char *firstName=NULL,char *additionalName=NULL,char *lastName=NULL,
char *adress=NULL,char *town=NULL,char *state=NULL,char *zip=NULL,char *country=NULL,
char *organization=NULL,char *department=NULL,char *subDep=NULL,char *job=NULL,
char *tel=NULL,char *fax=NULL,char *mobile=NULL,char *modem=NULL,
char *homepage=NULL,char *talk=NULL,
char *comment=NULL,char *birthday=NULL
);

Adds an address to Kab, or refills the entry of an existing Kab entry. Note: 'givenname' is the key in the addressbook. 'adrbookname' is the address identifier for the post address. 'givenname' is mandatory, so don't call kabAddress without it.

Empty strings (strings consisting only of white space or "") or 'NULL' strings will be ignored. Variables speak for themselves.

This is all

Ok, this is all there is to know. If you want to implement a new filter, just make a derivate of base class ’filter’, and implement the ’import’ function. You can call the API functions to Kab and Kmail, to let those handle your data. You can use the filterInfo API to output whatever you want to the KmailCvt dialog box. Please confine to the indicated usage of filterInfo. The internals of the filter you create don’t matter, as you can see with the ones I created.

May they be fast and not to resource filling!