Custom List Manager

By Larry Sharpe

Technical Note 07-23

Abstract

This Technical Note presents a technique for building customized lists. This technique combines 4D's existing list functionality with a customized system in order to provide features not available when using 4D's hierarchical lists.

A sample database is provided.

Overview

One of the needs of most 4D databases is having lists, a way to have your database users pick an item from a list of items in order to populate a field. There are several potential issues when using 4D's method of lists. The user is only allowed to pick one list item at a time and 4D does not allow for easily modifying the user interface to make it match the rest of your database's look and feel. Perhaps the biggest problem with using the standard 4D Lists is that when you update your structure (and you will) any lists updated by the users will be lost. This is because the list data is stored in the structure not the data file.

To deal with these problems we will store the list information in the data file and add code to manage the user interface functionality. This Technical Note adds one table to your database, one Project Method (PM) and slight changes to your forms to manage each list that you define in a much cleaner way.

Another large part of this Technical Note is how to use these new lists functions as a keywords addition to your database. I have found that almost every database has a need for storing keywords that are related to a parent record. Using this code you can have a way for the users to manage the list of user-defined keywords that they can pick from without having to type it in each time. There can be zero, one or multiple keywords attached to each parent record. With this slightly more involved addition of the lists capabilities, you will need another form and a PM to handle the keywords.

There was a Technical Note published by Steve Hussey back in November, 2000 that shows a way to manage lists. Since it has been a few years since that Technical Note and it was written for 4D v6.0, this Technical Note will be a redesigned version of that idea. This new Technical Note shows how the list functions can be redesigned to allow only one list item selection at a time, i.e. a popup list of cities; used to verify that the entered data is part of a valid list, i.e. a valid state; or used to allow picking many list items at a time, i.e. a list of keywords attached to a person's record.

Demo Database Implementation

This section discusses the implementation presented in the sample database.

The Table and Fields

You will need to add the [xLists] table and its fields. I have added the "x" to the table name so that it sorts to the bottom of the list of tables in the structure. I use this for the table names and the PMs so that commonly used code is separated from the client specific code. I find it is much easier to maintain code using this naming convention.

In the [xLists] table there are three fields, ListName (A30), SortOrder (L) and Element (A80). Each record is an element of a specific list and is set to a specific sort order in that list. Repeating the list name for each element may seem redundant but is much simpler than having two tables, one for the list names and a related table for the elements that belong to those lists.

The Form

The main form used to maintain the lists is named "ListEditor". This form allows the user to add elements to each of the lists that the developer has created. The users are not allowed to create lists, only the elements for the existing lists. The list of lists is actually stored as a 4D hierarchical list with the name "ListNames". Adding an element to this master list will make it show up in the ListEditor form. Most of the objects in this form have code in them which keeps track of which specific list you are working with and if you are adding, modifying, moving or deleting elements to that list. The good news is that it will work with all of your lists with no changes.

The PM

The method "xLists" has all the code needed to maintain your list data. This PM is used to edit the lists (using the table, fields and form described above), save the changes to any list being edited, and to load the data into inter-process arrays during start up or when the lists has been updated. This code is set to run as a **Case of** statement. I have found that it is very handy to have all the related code in one method as it is easier to maintain and keep track of using one method instead of many. It is very similar to how a form method works.

The first case in the **Case of** statement looks for when the method is called with no parameters, like when it is called from the menu bar. This will cause the dialog to open and allow the users to edit their lists. You could check to see if the user has permission to run this menu item or not, but that is not part of this Technical Note.

The “Modified” case of the **Case of** statement handles saving changes to a specific list. There are two very import variables passed to this method. The first, "sListName", tells the method which list to update and the other, "bListModified", lets the code know if anything was actually modified and needs to be updated. Any

time lists elements are added, modified or deleted the "bListModified" variable is set to True. When a different list is selected or the window is closed, the previously selected list is checked to see if it needs to be updated using this variable. Instead of trying to keep track of each list element, and there may be many of each, when this variable is True we know that something has been changed, so we delete ALL the saved records for that list and create new [xLists] records with the current data. This is much easier than trying to figure out what has been changed and what has not. Once this step is done we set the "bListModified" variable back to False until we update the list elements again.

The last selection of the **Case of** code is the "Build" section which is used to create inter-process arrays that are used throughout the database. This function is called from the OnStartup code and after the xLists PM has been accessed. The arrays used here are just examples to see how they are loaded. In a real database, I sometimes have up to 15 to 20 arrays defined like this. Remember that these are inter-process arrays, not process or local arrays. That means that they are the same arrays used by all clients and processes in your database. Use a process array if you are only using that array for that user or process. You can still use the [xLists] table and the code shown here to updated the elements of those process arrays, you just do not load them here, you would load them when the process or method is run.

Using the Demo Database

There are two demo tables, [People] and [PeopleKeywords]. [People] is a simple ID, name and address type table, [PeopleKeywords] is also a very simple table that has an ID field, a PeopleID field (to relate it to a specific [People] record) and a Keyword field. Each table's ID field is set using the Trigger method when the new record is first created using the **Sequence Number** command.

All forms for these two tables are very basic with the exception of the [People]Input form where there is a List Box area that shows the related [PeopleKeywords] records. This List Box, and the two buttons next to it, are all managed by the "People_Keywords" PM. Each of these objects calls this method when something needs to be done. All code and variables relating to the Keywords are defined and run in this one PM. Note that the + and - buttons could be removed and all the functionality need to maintain the Keywords for a person could be managed by just double-clicking the list and using the window displayed.

When the user opens the window (double-clicking or + button) to modify the keywords related to a person they are using the [People]Keywords form. This form shows all the "master" keywords that have been defined and what keywords have already been "selected" for that particular [People] record. Double-clicking or dragging-dropping on either list will add or remove the keywords as needed. Once again the "People_Keywords" PM is used to manage all this data as part of the **Case of** statement.

If a [People] record is deleted from the Output Form using the Delete… menu item,

the "People_Delete" PM code calls the "People_Keywords" PM in order to delete all related [PeopleKeywords] records before deleting the [People] record. There is no need to have those unused keywords laying around the data file.

Once again, having all the code and variables used for this set of data defined and used in one method, "People_Keywords", makes it very easy to duplicate this type of functionality somewhere else in the database. All you need to do once you have defined your new keyword table (e.g. [CompaniesKeywords]) is copy and paste the form and the project method, do a search and replace on the method, variable and field names used for the new values and test it. I have used "keywords" here many times, but this could also been something very differently named for your particular database needs.

In the [People]Input form are two other areas that we are using custom lists. Next to the [People]City field is a popup menu that is populated by the [xLists] data and allows the user to quickly pick a city from the list. In the [People]State field the code checks to see if a hand-entered state is part of the valid list of states. Both of these examples have very simple code attached to them and they use arrays populated from the [xLists] data as appropriate.

Limitations and Enhancements

One thing that this Technical Note does not address is that it does not allow for Icons or Images to be associated with List elements. This functionality could be added if needed by adding another field to the [xLists] table and the appropriate code to manage that fields data entry and viewing of the lists.

Conclusion

This Technical Note shows how a common function needed in many 4D database applications can be added with little change to your current database code. Please examine the code for much more specific comments of what each section of the code is doing and if it needs to be modified in order to be moved into your database.

Larry Sharpe can be contacted at (831) 373-6266 or LSharpe@infoservice.com