Evoke

cappture your imagination

Integrating Evoke with MultiValue Databases

Table of Contents

Introduction

This document describes the way in which you can integrate your MultiValue database environment with Evoke.

Background

Evoke supports the concept of *data repositories*. Most of the time this term is synonymous with the term *database*; however, Evoke's architecture is such that any type of persistent data store may be integrated with an app.

Evoke's app generator allows you to associate one or more specific types of data repository with an app. This then guides the code generation process such that the code required to interact with the selected repository type(s) is produced within the Visual Studio solution.

Generated code structure

The app generator will always include a project within the generated Visual Studio solution called "RepositoryAccess". This project contains the .NET code required to drive the data repository access part of your app.

The RespositoryAccess project is basically divided into 2 parts; a part which is repository agnostic and another part which is repository specific. This latter part will be further sub-divided based on the number of different repository types selected for the app at the time of code generation.

This 2-part division of the RepositoryAccess project allows a clean partitioning of that code which is not concerned with the specific type of repository being used and that code which is highly tuned to support the features and access technology of a specific repository implementation. The connection joining these 2 main parts is based on a series of interfaces automatically created by the app generator.

Therefore, based on the above architecture, it can be seen that dealing specifically with MultiValue database connectivity within Evoke is a matter of concern only for the repository specific section of the RepositoryAccess project. This document, therefore, focuses on this section of code along with the code that resides on the MultiValue server itself.

.NET generated code

Strictly speaking, the .NET code (C# or VB.NET) created by the app generator does not require any customization work within Visual Studio in order to make it work. However, if you wish to integrate some custom functionality (for example some validation or other business logic) specifically for your MultiValue database, you may add it at the various places set aside for this purpose by the generator. Adding custom .NET code is covered at the start of the *Working with Generated Visual Studio Projects* document.

Database server resident code

One of the main purposes of the generated .NET code is to invoke the relevant DataBASIC subroutines residing on your MultiValue database server. The specific server-side routine that is called and the data that is passed to it depends entirely upon the type of repository related action that has occurred within the UI.

The app generator can create the initial content of these server-side routines (known as CRUD routines – Create, Read, Update and Delete) and you are then able to add your own code as necessary in order to implement the required database file access logic.

The app generator, when instructed, will install the CRUD routines into a file (which it also creates) called EVOKE.CRUD.BP. The naming and content of these CRUD routines is based on the data entities created within the app designer. Each data entity will be represented by 2 routines called EV.CRUD.xxx and EV.CRUD.xxx.CUSTOM, where xxx represents the name of the data entity.

The EV.CRUD.xxx routine is "owned" by the app generator, that is, you should not amend its content. Conversely, the EV.CRUD.xxx.CUSTOM routine (referred to hereafter as the "custom CRUD routine" is "owned" by you, and will only be modified by the generator once – i.e. when it is first created by the generator. The structure of each custom CRUD routine is covered in the following section.

The generated DataBASIC CRUD routine

When you ask the app generator to install the CRUD routines, as mentioned previously, it will create 2 subroutines per data entity. The routine named EV.CRUD.xxx (where xxx represents the name of the data entity) contains code which can be viewed as the default implementation of each of the possible CRUD actions. The second routine installed by the generator, EV.CRUD.xxx.CUSTOM, allows you to, where relevant/necessary, override these default implementations by incorporating your own DataBASIC code.

The default implementation of each CRUD action within the EV.CRUD.xxx subroutine is based on the data entity mapping definition set up within the app generator. Please refer to *the App Generator User Guide* document for more details on this topic. This mapping data, in essence, defines a logical structure for the "blob" of entity data that passes between .NET and DataBASIC. The knowledge of this structure by both .NET and DataBASIC is obviously crucial because it is the only way in which these 2 environments can successfully synchronize their individual elements of data exchange logic.

Thus, the default implementation of each CRUD action within DataBASIC is based on the assumption that each entity maps onto a single database file, and that the structure of this file is exactly the same as the app generator's mapping definition blob described above. This may be true for some entities, but probably not true for many others. In this latter case you will need to override the default implementation of CRUD actions with your own code within the CUSTOM version of the CRUD routines.

Sometimes it is useful to look at the generated CRUD routine in order to look at an example of how some of the calling signature arguments can/should be used.

The DataBASIC custom CRUD routine

Each custom CRUD routine has the same very simple structure. It has a few lines of initialization code at its start but is then essentially one big Case statement, which each Case clause relating to one possible type of database action – database actions initially triggered by activity within the UI.

Calling signature

Every custom CRUD routine has the same calling signature, the arguments of which are explained below.

Argument: ContextData

This argument contains contextual data passed from the UI. It represents any data which is required to support the implementation of the given action. The content of ContextData will consist of both data included automatically by Evoke and data added programmatically by custom UI code.

The structure of the content of ContextData is a series of attribute mark delimited values. Each of such values, in turn, consists of 2 values separated by a value mark; the first of which is the name or Id of the particular context data value, the second value being a list of one or more sub-value mark delimited values (i.e. the actual value of the context data entry).

Argument: ControlIn

(System usage only) This argument contains data supplied by Evoke to control the execution of the action.

Argument: DataIn

This argument contains the main incoming data payload relating to the action. The data contained in this argument (where relevant) is explained for each separate CRUD action below.

Argument: ControlOut

(System usage only)

Argument: DataOut

This argument contains the main return data payload relating to the action. The data contained in this argument (where relevant) is explained for each separate CRUD action below.

Argument: ErrorOut

If an error occurs during the CRUD action, this argument should be set to a description of the error.

Argument: ActionCompleted

This argument should be set to integer value "1" (equated to variable TRUE) if you have implemented the invoked CRUD action within the custom routine. If you do this, the default implementation of the CRUD action will be skipped.

CRUD actions

The vast majority of each custom CRUD routine takes the form of a Case statement based on the incoming first attribute value of argument ControlIn.

The different types of database actions that the custom CRUD routine supports are as follows.

Action: Create

This represents the creation of a new instance of the associated entity within UI code. The main requirement here is the supply of the item ID (primary key) to be used for the new instance. The new item ID should be assigned to the DataOut argument.

Action: Read

This action relates to a request to read a single instance of an entity.

The first attribute of the DataIn argument will be set to the required item ID.

The second attribute value of argument ControlIn will be set to string value "Lock" if a ReadU needs to be used, otherwise a normal non-locking Read should be used.

The DataOut argument should be assigned the data values of the read entity. The structure of this data needs to be the same as defined by the data entity mapping definition within the app generator. See above for more details on this.

Action: ReadMultiple

This action relates to a request to read one or more entity instances based on the supply of one or more item IDs.

The DataIn argument will be set to an attribute mark delimited list of required item IDs.

The DataOut argument needs to be assigned the data relating to the list of read entities. The structure of this list is simply a concatenation of multiple instances the same DataOut structure as used in the Read CRUD action (see above), except that at the front of each entity needs to be a 5 digit hexadecimal character count and the associated item ID, separated from the data entity blob by an attribute mark. You can see an example of this process in the generated CRUD code. An example of which is given below:

```
SELECT DataIn
EOLIST = 0
LOOP
    READNEXT ID ELSE
        EOLIST = 1
    END
UNTIL EOLIST DO
    READ PRODUCTDATA FROM PRODUCTFILE, ID THEN
        DATALENGTH = LEN(ID) + LEN(PRODUCTDATA) + 1
        NEWDATA = OCONV(DATALENGTH, "MCDX") "R%5":ID:AM:PRODUCTDATA
        DataOut := NEWDATA
    END
REPEAT
```

Action: Delete

This action relates to a request to delete a single instance of an entity.

The first attribute of the DataIn argument will be set to the ID of the entity to be deleted.

Action: Logic

This action relates to a request to execute a piece of server-resident business logic associated with the entity type.

The first attribute of the DataIn argument will be set to the ID of the required section of logic.

The second attribute of the DataIn argument will be set to a multivalued list of data values passed in from the UI to support the execution of the logic.

The DataOut argument needs to be assigned the required return data payload from the logic. This value will be returned as-is to the UI.

Action: Select

This action relates to a request to return one or more entity instances based on the supply of a selection ID and optional filter values

Attribute positions 2, 3 and 4 of the ControlIn argument may be used to provide page-based details for the selection as follows:

Attribute position	Description
2	A GUID identifying a specific selection invocation.
3	The (zero-based) index of the required page.
4	The number of items to be returned per page.

The first attribute of the DataIn argument will be set to the ID of the required selection.

The second attribute of the DataIn argument will be set to a multivalued list of filter values passed in from the UI to be used within the filter/selection criteria clause of the selection command/process.

The third attribute of the DataIn argument will be set to a multivalued list of property names indicating the required sort order for the returned data items - highest sort order first.

The DataOut argument needs to be assigned the data relating to the list of selected entities. The structure of this list is the same as per the ReadMultiple CRUD action above.

Action: Update

This action relates to a request to update the data associated with a single instance of an entity.

The second attribute value of argument ControlIn will be set to string value “Lock” if a WriteU needs to be used, otherwise a normal lock-releasing Write should be used.

The first attribute of the DataIn argument will be set to the target item ID.

The second and subsequent attributes of the DataIn argument will be set to the new data fields of the target item ID.

The DataOut argument should be assigned the data values of the updated entity. The structure of this data needs to be the same as defined for the Read CRUD action above. Thus, the DataOut argument can be used to relay data modifications made within DataBASIC back up to the UI if required.

Action: Unlock

This action relates to a request to release an item lock previously placed by a Read CRUD action.

The first attribute of the DataIn argument will be set to the target item ID.

Action: Validate
This action relates to a request to perform a server-resident property validation action.

The first attribute of the DataIn argument will be set to the target property name.

The second attribute of the DataIn argument will be set to the candidate value of the property, i.e. the value to be validated.

The third attribute of the DataIn argument will be set to a multivalued list of data values passed in from the UI to support the execution of the validation logic.

The DataOut argument needs to be assigned the optional return data payload from the validation. This value will be returned as-is to the UI.

Calling Non-entity specific logic
The *Logic* CRUD action allows you to execute a section of custom logic associated with an entity type.

For situations where a section of logic does not really relate to a specific entity type, you may execute that logic from the UI by specifying a unique ID for that logic, for example “MonthEndClose”.

The way in which this logic is invoked by the UI depends on the UI environment language. An example for JavaScript is given below:

```
App.callRemoteLogic(“MonthEndClose”, “10”, function(dataOut) { } );
```

This code will result in a subroutine called “EV.MonthEndClose” being called on the Multivalue database server, pass the value “10” as its LogicData. It will be expected to have the following calling signature:

```
SUBROUTINE EV.MonthEndClose (ContextData, LogicData, DataOut, ErrorOut)
```

Argument: ContextData
This argument contains contextual data passed from the UI as per a normal CRUD action (see above).

Argument: LogicData
This argument contains a multivalued list of data values passed in from the UI to support the execution of the logic.

Argument: DataOut
This argument needs to be assigned the required return data payload from the logic. This value will be returned as-is to the UI.

Argument: ErrorOut
If an error occurs during the logic action, this argument should be set to a description of the error.