Lab 10.1: Building an Advanced Database Application

Objectives

After completing this lab you will be able to:

- Write an application to extract database information, such as the names of tables, fields, and queries.
- Use CDaoFieldInfo structures to determine field data-type.
- Use CDaoQueryDef to get query information.
- Use parameterized queries.
- Create dynamic queries.
- Determine support for bookmarks, and use bookmarks in a supported recordset.
- Attach a table from a foreign database to a database object.

Prerequisites

You should know how to work with dialog boxes and have completed Chapter 10 before attempting this lab.

Lab Setup

This demonstration shows what you will accomplish during the lab.

Estimated time to complete this lab: **75 minutes**.

Exercises

The following exercises provide practice working with the concepts and techniques covered in this chapter.

Exercise 1: Extracting Database Table and Query Names

In this exercise, you will populate a tab control and associated list box with database information.

Exercise 2: Creating the Recordset and Extracting Field Names

In this exercise, you will create a recordset and then extract field names from a database table or query definition.

Exercise 3: Displaying Database Records

In this exercise, you will populate the rows of a list view control with records from a recordset.

Exercise 4: Finding Records and Using Bookmarks

In this exercise, you will write a function that creates and uses bookmarks.

Exercise 5: Attaching to an External Database Table

In this exercise, you will attach the database viewer to an external database table.

Copy the contents of \Labs\C10\Lab01\Xxx\Baseline, where Xxx is the exercise number, to your working directory.

The completed code for these exercises is in \Labs\C10\Lab01\Xxx, where Xxx is the exercise number.

Exercise 1: Extracting Database Table and Query Names

In this exercise, you will add code to the baseline application to populate a tab control and associated list box with database information.

About the Baseline Application

AppWizard was used to create an MDI Application without Print or Print Preview, and with database headers only. The view class is derived from **CFormView**.

Note The baseline application will not build until you have completed Exercise 1.

The completed application consists of an MDI application that opens one or more databases. A separate document stores each database object. Each open database can display one or more tables, each table in a separate view. (Each document has one or more views.) The application also displays the results of stored select Queries. The application prompts the user for Query parameters when appropriate. The user can click column headers to further refine the display of records from the recordsets. Finally, the completed application enables the user to attach to an external database table.

This table describes the functions that you will create, modify, or use in one or more of the lab exercises for Lab 10.1.

Function	Description
void CBrowserView::GetDBNames(int nInfo)	Helper function to get the definition count from the database for tables, queries, and relations.
void CBrowserView::OnViewData( )	When a user selects a table or query for display, **OnViewData** creates a new view using a template (created in **CBrowserApp::InitInstance**) and then calls **CDataView::SetData**.
BOOL CDataView::SetData(CDaoDatabase * pDB, const CString & name, const int nType)	Creates a recordset and extracts field names from a database table or query. **SetData** then populates the list view. Its parameters are the database pointer, the table or query name, and whether it is a table or query.
void CDataView::SetParams(CDaoQueryDef * pQDef)	Used with parameterized queries, to get parameter values from the user and set the query parameters.
void CDataView::LoadHeaders( )	Adds a column to the list control for each field in the recordset and puts the field name into the column header.
void CDataView::LoadData(CString * pstrCriteria)	Used to populate the rows of the list view control.
CString strVARIANT(const COleVariant & var)	Defined in VarDecoder.Cpp.
void CDataView::OnColumnclick(NMHDR* pNMHDR, LRESULT* pResult)	When the user clicks a column header, posts a message for **CDataView::OnHeadClick** to handle.
long CDataView::OnHeadClick(UINT uP, long)	Builds a query for the field clicked and then calls **CDataView::LoadData**.
void CDataView::FindQueryData(CString & strCriteria)	Sets a bookmark, if possible. **FindQueryData** then finds and displays queried data. Finally, it returns to the bookmark, or if there is no bookmark, to first record.
CBrowserView::OnViewAttach( )	Attaches the database viewer to an external database and calls **CDataView::SetData**.
CDataView::AttachTable( )	Prompts the user for a database and

	table name. **AttachTable** then attaches the table to the current database and opens a dynaset type recordset.
CDataView::~CDataView( )	**CDataView** destructor closes open data objects. In Exercise 5, you add cleanup code to this function to delete the **CDaoTableDef** object.

Exercise 1: Extracting Database Table and Query Names

In this exercise, you populate the tab control and associated list box with database information.

➢ Provide access the database object

1. Add a protected **CDaoDatabase** variable m_daoDB to the document class.
2. Add a public accessor member-function **CDaoDatabase * GetDatabase( )** that returns a pointer to the database variable.

```
CDaoDatabase * CBrowserDoc::GetDatabase( )
{
    return & m_daoDB;
}
```

➢ Extract database table and query names and add to the list

1. In the **CBrowserView** class, locate the code for the member function **GetDBNames**.
2. Modify the first switch case within the function's **for** loop to extract the table information using the function **CDaoDatabase::GetTableDefInfo**. Use the loop counter variable as the first argument and the structure variable **tInfo** as the second argument to **GetTableDefInfo**.

   In the TABLE case, before adding the table-name string to the list box, test the constant dbSystemObject against the attributes returned in the **tInfo** structure. If the dbSystemObject attribute is set, do not add the table name to the list box, because system tables are internal tables that do not interest the user.

```
case TABLE :db.GetTableDefInfo( i, tInfo );
if ( ! ( dbSystemObject &&
            tInfo.m_lAttributes ) ) //Don't want system tables
m_List.AddString( tInfo.m_strName );
break;
```

3. Modify the second switch case within the function's **for** loop to extract the query information using the function **CDaoDatabase::GetQueryDefInfo**. Use the loop counter variable as the first argument and the structure variable qInfo as the second argument to **GetQueryDefInfo**. Then add the query name to the list.

```
case QUERY : db.GetQueryDefInfo( i, qInfo );
    m_List.AddString( qInfo.m_strName );
break;
```

Notice that the definition count from the database for the appropriate tab (Table, Query, or Relation) is determined by using pointers to member functions.

➢ Build and test your application

1. Open an Access-type database, for example Labs\C10\Biblio.Mdb.
2. Click the Tables tab.

   The list box should show table names. Click the Queries tab. The list box should show query names.

3. Double-click a query or table name.

   You should get an empty frame, and a message box stating that the recordset could not be opened.

The completed code for this exercise is in \Labs\C10\Lab01\Ex01.

Exercise 2: Creating the Recordset and Extracting Field Names

In this exercise, you will add code to the baseline application to create a recordset and then extract field names from a database table or query definition.

You add code to create a recordset and then extract field names from a database table or query definition to the function **CDataView::SetData**. (**CBrowserView::OnViewData** calls **CDataView::SetData** with the user-selected table or query name.) You then use the extracted field names to populate column headers in a **CListView** derived class.

➢ **Create a recordset**

1. Locate the function **CDataView::SetData** in DataView.Cpp.
2. Remove the opening statement: `return FALSE`.

   The statement enabled you to test your code in Exercise 1.
3. Within the **try** block, determine whether the parameter nType is equal to the enumerated value TABLE.

```
        if ( TABLE == nType )
        {
```

   a. If it is equal to TABLE, create a **CDaoRecordset** object on the heap. Pass the database pointer as an initializer. Assign the returned pointer to the member variable, m_pRecordset.

```
      m_pRecordset = new CDaoRecordset( m_pDB );
```

   b. Define a **CString** object. Create a SQL statement using the **CString** parameter, name, to choose all columns from the table. Bracket the table name.

```
              CString sql = "Select * from [" + name + "]";
```

   c. Use the SQL statement to open the **CDaoRecordset** object as a dynaset type recordset.

```
              m_pRecordset->Open( dbOpenDynaset, sql );
          }
```

4. Otherwise test to see whether the parameter nType is equal to QUERY:

```
else if ( QUERY == nType )
```

5. If it is a query, do the following:

   a. Create a **CDaoQueryDef** object on the heap. Pass the database pointer as an initializer. Assign the pointer returned by new to the member variable, m_pQuery.

```
              m_pQuery = new CDaoQueryDef( m_pDB );
```

   b. Use the **CString** argument, name, to open the **CDaoQueryDef** object.

```
              m_pQuery->Open( name );
```

   c. Determine the type of query by calling the function **CDaoQueryDef::GetType**. If the query type is equal to **CDaoQueryDef::dbQSelect**, add code to call **CDataView::SetParams**, passing it the query pointer, and then create and open a new recordset object:

```
              if ( dbQSelect == m_pQuery->GetType( ) )
              {
                  SetParams( m_pQuery );
                  m_pRecordset = new CDaoRecordset( m_pDB );
```

```
                m_pRecordset->Open( m_pQuery );
            }   //End if Select-Query
```

d. Otherwise, if the query type is not equal to **CDaoQueryDef::dbQSelect**, use a message box to inform the user that the query is not a select query. Then return FALSE to indicate no open recordset.

```
else
            {
                AfxMessageBox( "Not a Select Query" );
                return FALSE;
            }
        } //End Query-type
```

6. To finish this exercise, complete the function **CDataView::LoadHeaders**. This function adds a column to the view's list control for each field in the recordset. **LoadHeaders** then places the field name in the column header.

   a. Locate the **try** block within **CDataView::LoadHeaders**. Your code goes within this **try** block. After the definition of the two integer variables, width and align, define a **CDaoFieldInfo** structure named **fInfo**.

   ```
   CDaoFieldInfo fInfo;   //Structure of field information
   ```

   b. Define an integer variable, columns, and initialize it to the number of fields in the recordset using **CDaoRecordset::GetFieldCount**.

   ```
   int columns = m_pRecordset->GetFieldCount( );  //How many fields
   ```

   c. Inside the **for** loop that follows, use **CDaoRecordset::GetFieldInfo** to fill the **CDaoFieldInfo** structure for each column.

   ```
       m_pRecordset->GetFieldInfo( i, fInfo );
   ```

   d. If the field type is dbText or dbMemo, set the alignment variable to LVCFMT_LEFT. Otherwise, set the alignment variable to LVCFMT_RIGHT.

   ```
       align =
           dbText == fInfo.m_nType || dbMemo == fInfo.m_nType
           ? LVCFMT_LEFT : LVCFMT_RIGHT;
   ```

   e. Set the width variable with a call to **CListCtrl::GetStringWidth**. Use the **CDaoFieldInfo** m_strName member variable as the string argument.

   ```
       width = lstctrl.GetStringWidth( fInfo.m_strName );
   ```

   f. Finally, insert the column by calling **CListCtrl::InsertColumn**, using the column number, field name, alignment, and twice the width as the **InsertColumn** arguments.

   ```
       //Make the column twice the label width
       lstctrl.InsertColumn( i, fInfo.m_strName, align, 2 * width );
   ```

➢ **Build and test your application**

When you select a table or query from the database (by double-clicking in the list box, or by selecting and using the menu), your report view should appear with field names in the header control. Names of numeric fields should align to the right.

The completed code for this exercise is in \Labs\C10\Lab01\Ex02.

Exercise 3: Displaying Database Records

In this exercise, you will add code to the baseline application to populate the rows of a list view control with records from a recordset.

You will add code to populate the rows of the list view control with records from the recordset to the function **CDataView::LoadData**. You will add code to a **try** block within the function framework. The block contains a nested loop. The outer loop steps through records within the recordset; the inner loop moves across the fields of the record.

➢ **Display a page of records**

1. Locate the **try** block within **CDataView::LoadData**. Observe that an integer variable, rows, defined before the **try** block, indicates how many rows can display at once. Within the **try** block, define three variables: var, a **COleVariant** to hold field values; str, a **CString** to convert the field values to text; and columns, an integer to hold the field count. Initialize the integer variable using **CDaoRecordset::GetFieldCount**.

```
COleVariant var;
CString str;
int columns = m_pRecordset->GetFieldCount( );  //How many fields
```

2. Two nested **for** loops to move through the records and across the fields have been provided for you. The outer loop stops if either the row count is equaled, or the recordset **IsEOF** becomes true. The field count variable is used to determine when to stop the inner loop. Add a statement at the top of the inner **for** loop to set the **COleVariant** variable for the current field.

```
        m_pRecordset->GetFieldValue( c, var );
```

   The code to convert the **COleVariant** variable to a **CString**, to insert the field values into the list view control, and to close the inner **for** loop, is provided.

3. Now, following the closing brace of the inner for loop, add code to complete the outer **for** loop.
   a. Test whether the function argument **CString * pstrCriteria** has a value. If so, there is a criteria string to locate records. That is, the user clicked on a field header, and provided a search value.

   ```
   if ( pstrCriteria )  //We're passed a criteria string *
   ```

   b. Add an opening brace for the **if** block and call **CDaoRecordset::FindNext** with the argument *pstrCriteria (the string object). You must cast the argument to an LPCTSTR. If this function call returns FALSE, break out of the (outer) loop, because no further records meet the criteria. Add a closing brace for the **if** block

   ```
   {
       //Try to find another record, if not found then
       if ( ! m_pRecordset->FindNext( ( LPCTSTR ) * pstrCriteria ) )
           break;  //stop the looping on rows of data
   }
   ```

   c. Otherwise, if **pstrCriteria** is NULL, we are not using a criterion, so call **CDaoRecordset::MoveNext**.

   ```
   else  //No criteria, so we just take the next record
           m_pRecordset->MoveNext( );
   ```

4. Finally, if the recordset is **IsEOF**, move to the last record. This leaves the recordset on a record.

```
if ( m_pRecordset->IsEOF( ) )
            m_pRecordset->MoveLast( );  //Leave cursor on a record
```

➢ **Build and test your application**

The provided command handlers to page up and down through the data enable you to browse all the rows in the recordset. If you click a column header, a dialog box prompts you for a criterion for the designated field. A **CString** is built from the dialog box information, and placed in the recordset's member variable m_strFilter. A call is made to **CDaoRecordset::Requery**, and then to **CDataView::LoadData**.

The completed code for this exercise is in \Labs\C10\Lab01\Ex03.

Exercise 4: Finding Records and Using Bookmarks

In this exercise, you will add code to the baseline application to create and use bookmarks.

You will complete the function **CDataView::FindQueryData**, which sets and moves to bookmarks.

When the user clicks a list-view column-header, and the recordset is based on a query, the handler, **CDataView::OnHeadClick**, calls **CDataView::FindQueryData**. The function **FindQueryData** sets a bookmark at the top of the current page, if possible. If **FindQueryData** finds one or more records, starting after the current page, it displays up to one full page. If there is no matching record, the **FindQueryData** returns to the bookmark, if any. If there is no bookmark, **FindQueryData** moves to the first record in the recordset.

➢ **Add code to CDataView::FindQueryData to set and use bookmarks**

1. Locate the **try** block in **CDataView::FindQueryData**. Add a statement to define a **COleVariant** variable, varBookmark, to use as a recordset bookmark.

```
COleVariant varBookmark;
```

2. After the statement that initializes the integer rows to the number of items in the list, add an **if** statement using **CDaoRecordset::CanBookmark** to test whether the recordset supports bookmarks. Then add an opening brace for the **if** block.

```
//If we can, we get a bookmark
if ( m_pRecordset->CanBookmark( ) )
{
```

3. Inside the **if** block, add code to do the following:

   a. Use the value in the integer variable, rows, to move toward the beginning of the recordset.

```
m_pRecordset->Move( - rows );  //Move to top of display
```

   b. If **CDaoRecordset::IsBOF** is true, move to the first record in the recordset.

```
if ( m_pRecordset->IsBOF( ) )  //Too far for a bookmark?
    m_pRecordset->MoveFirst( );
```

   c. Get the bookmark for the current record.

```
    varBookmark = m_pRecordset->GetBookmark( );
```

   d. Use the integer variable, rows, to move back to the original position (toward the end of the recordset) and add a closing brace to end the CanBookmark **if** block.

```
    m_pRecordset->Move( rows );  //Return to original location
}
```

4. Add an **if** statement that calls **CDaoRecordset::FindNext** with the **FindQueryData** function's parameter, strCriteria, as the argument.

```
if ( m_pRecordset->FindNext( strCriteria ) )  //Found it!
```

5. If **FindNext** indicates success, call the function **CDataView::LoadData**, passing it the address of strCriteria as the argument.

```
LoadData( & strCriteria );   //Load a page of found items
```

6. Otherwise, add an **else** block.

```
else//we didn't find a record
```

```
{
```

7. Inside the **else** block, add code to do the following:
   a. If the recordset supports bookmarks, use **CDaoRecordset::SetBookmark** to return to the original page.

```
      if ( m_pRecordset->CanBookmark( ) )   //We have a mark
              //return to bookmarked record
              m_pRecordset->SetBookmark( varBookmark );
```

   b. Otherwise, if the recordset does not support bookmarks, move to the first record.

```
      else    //If no mark, go to first record
                  m_pRecordset->MoveFirst( );
```

   c. Display a message box to tell the user that the data was not found.

```
      MessageBox( "Record not found" );
```

   d. Call the function **CDataView::LoadData** without any arguments. This causes the page of records to reload without using any criterion. Then add a closing brace to complete the failure to find a record block.

```
      LoadData( );  //Bookmark or no, fill the list
  }
```

➢ **Build and test your application**

When the user views a table, clicking a column header requeries the recordset with the user-supplied criteria. The page movements show only records that meet the criteria.

If the report is query-based, only one page of records matching the criteria displays. Any subsequent page movement within the recordset brings up contiguous records, since the page movement handlers call **CDataView::LoadData** with no parameter.

The completed code for this exercise is in \Labs\C10\Lab01\Ex04.

Exercise 5: Attaching to an External Database Table

In this exercise, you will add code to the baseline application to attach the database viewer to an external database table

You complete functions **CBrowserView::OnViewAttach** and **CDataView::AttachTable** to attach the viewer to an external database table.

A new menu item, Attach, has been added to the View menu. A partially complete handler, **CBrowserView::OnViewAttach**, also has been provided for this menu item.

A partially complete function **CDataView::AttachTable** also has been provided. **AttachTable** opens a file dialog to get a Microsoft Access style database name from the user. It then uses a dialog box to prompt for a table name. You will add code to attach the table to the current database, and open a dynaset type recordset.

➢ **Attach the database viewer to an external database**

1. Open the **CDataView** class header file. Find the **enum** statement that has TABLE, QUERY, FIND. Add the identifier, ATTACH, after the identifier FIND. You will reference this identifier in the next step.

```
    enum{ TABLE = 0, QUERY, FIND, ATTACH };
```

2. Open the **CBrowserView** class implementation file near the bottom of the function **CBrowserView::OnViewAttach**, just before the call to **CDocument::UpdateAllViews**, add an **if** statement to test for an unsuccessful call to **CDataView::SetData**, with the parameters: **pDoc->GetDatabase()**, **"Attached Table"**, and **CDataView::ATTACH**.

```
    if ( ! pDataView->
          SetData( pDoc->GetDatabase( ), "Attached Table",
                   CDataView::ATTACH ) )
```

3. Follow the **if** test with a call to **AfxMessageBox** to display the literal string "Could not create recordset."

```
              AfxMessageBox( "Could not create recordset" );
```

➢ **Attach the table to the current database and call SetData to open a dynaset type recordset**

You will complete the function **CDataView::AttachTable**. When the thread of execution reaches your code, the user has entered a valid file name in a **CFileDialog** object, dlgFile.

1. Open the implementation file for the **CDataView** class. Locate the function **CDataView::AttachTable**. Before the last statement: `return true;` insert code to create a connect string. Define a **CString**, strConnect. Initialize it with the string literal ";DATABASE=" and concatenate the path from the file dialog to strConnect.

```
CString strConnect = ";DATABASE=" + dlgFile.GetPathName( );
```

2. Create a new **CDaoTableDef** object. Use the data member m_pDB as the initializer. Store the returned pointer in the data member m_pTable.

```
m_pTable = new CDaoTableDef( m_pDB ); //Get a CDaoTableDef object
```

3. Call **CDaoTableDef::Create**. Attach the table using the dialog box variable m_strAttached (which contains the name of the table supplied by the user) for the first and third parameters. Use **0** (zero) for the second parameter and the connect string from Step 1 as the fourth parameter.

```
m_pTable->Create( m_strAttached, 0,   //Create the tabledef
m_strAttached, strConnect );
```

4. To open the table and append to the database, call CDaoTableDef::Append.

```
m_pTable->Append( );        //Open and Append to database
```

5. Finally, call **CDataView::SetData**, using the parameters: m_pDB, m_strAttached, and TABLE. **SetData** will create the recordset as a dynaset.

```
SetData( m_pDB, m_strAttached, TABLE ); //Create a dynaset
```

➢ **Modify the function CDataView::SetData to work with the new feature of attaching a table**

Recall that **SetData** actually is called twice for an attached table. The first call to **SetData** is from **CBrowserView::OnViewAttach**. On this first call, the enumerated value CDataView::ATTACH is passed as the third parameter. The second call is in the function **CDataView::AttachTable**, which you completed in the previous procedure. In Step 5 of that procedure, **SetData** is called with the enumerated value CDataView::TABLE as the third parameter.

It is important to ensure that the body of **SetData** does not execute twice, since it adds the headers to the report view. To do this, open **CDataView::SetData** and add as the first statement in the **try** block the following code:

```
if ( ATTACH == nType )
        return AttachTable( );
```

Note Since this call to **CDataView::AttachTable** is contained within a **try** block, we did not protect the database functions within **AttachTable** with **try**/**catch** blocks.

➢ **Modify the destructor CDataView::~CDataView to work with the new feature of attaching a table**

1. Open the implementation file for the **CDataView** class. Find the destructor **CDataView::~CDataView**. After the statement that closes the query, add code to test whether the data member CDataView::m_pTable is nonzero and add an opening brace for the **if** block.

```
if ( m_pTable )
{
```

2. Inside the **if** block, add code to close the table.

```
    m_pTable->Close( );
```

3. Finally, add code to delete the **CDaoTableDef** object using the pointer stored in CDataView::m_strAttached and close the **if** block.

```
    m_pDB->DeleteTableDef( m_strAttached );
}
```

➢ **Build and test your application**

The code you entered enables you to look at tables in one or more foreign databases by attaching to the table(s) through your current database connection.

1. Once you have attached to one of the test databases, choose the Attach Table option and select a database.
2. When you are prompted for a table name:
   a. If you enter a valid name, that table temporarily appears as part of your database.
   b. If the table name is invalid, you receive the "Could not create recordset" message.
   c. If the name is a duplication of a table name in the active database, you are notified that "Object 'tablename' already exists."

The completed code for this exercise is in \Labs\C10\Lab01\Ex05.