Waiter System – Practical Workshop 8

Objectives

The main objective for Workshop 8 is to complete the CRUD functionality for the employee system by extending what was already done in the previous workshops sessions to also include:

- The listing of employees by their specific role
- The ability to edit the details of an employee

==========

1. Open the DB class for coding. Include an **enum** *DBOperation* for the three Crud functions: Add, Edit and Delete.
2. Open the ***EmployeeDB*** class to include the Edit functionality as follows:
   2.1 Add a third parameter *operation* of type *DB.DBOperation* to the *FillRow* method. This is to ensure that the ID, and the EMPID field are not changed during the edit and delete operations.
   2.2 We will change the existing code to check for the state/operation being selected as follows:

```
//2.2.1 write the code for the FillRow Method header ---make sure you have added the 3rd
parameter as indicated in question
 {
     HeadWaiter headwaiter;
     Runner runner;
     Waiter waiter;
  //2.2.2 write the code here to check: if the operation is an Add operation.
     {
          aRow["ID"] = anEmp.ID;
         aRow["EmpID"] = anEmp.EmpID;
     }
---The rest of the code stays the same---
```

   2.3 In the *Utility Methods* region: We need a private method *FindRow* to find a row in the database. This method finds and returns the row for a specific employee(by ID) in a specific table. The method has two parameters: An Employee object *anEmp*, and a variable *table* of type string. The method is not complete. You are required to complete it – 2.3.1-2.3.9.

```
//2.3.1 Write the method header for the FindRow method
{
    //2.3.2 declare a variable rowIndex initialised to 0
    //2.3.3 declare a variable myRow of type DataRow
    //2.3.4 declare a variable returnValue and initialise it to -1;
     foreach (DataRow myRow_loopVariable in dsMain.Tables[table].Rows)
     {
        //2.3.5 assign myRow_loopVariable to myRow
         //Ignore rows marked as deleted in dataset
                //2.3.6 if the row is not in a delete state (i.e  if the row state of myRow is not the
                   deleted state of the DataRowState)
         {
             if (anEmp.ID == Convert.ToString(dsMain.Tables[table].Rows[rowIndex]["ID"]))
             {
                        //2.3.7 assign rowIndex to returnValue
             }
         }
                //2.3.8 Increment the rowIndex counter
     }
       //2.3.9 the method should return the variable returnValue to its caller;
    }
```

   2.4 Add a second parameter *operation* of type DB.DBOperation to the *DataSetChange* method

We will need to change the code in the *DataSetChange* method to include all database operations. The code is given below. It is not complete. Make changes to it and complete the program

```
//2.4.1 Method header – write the code for the header
{
    DataRow aRow = null;
    string dataTable = table1;
    switch (anEmp.role.getRoleValue)
    {
        case Role.RoleType.Headwaiter:
            dataTable = table1;
            break;
        case Role.RoleType.Waiter:
            dataTable = table2;
            break;
        case Role.RoleType.Runner:
            dataTable = table3;
            break;
    }
         switch (operation)
         {
         case DB.DBOperation.Add:
                 aRow = dsMain.Tables[dataTable].NewRow();
                 FillRow(aRow, anEmp, operation)
                 dsMain.Tables[dataTable].Rows.Add(aRow); //Add to the dataset
                 break;
         // For the Edit section you have to find a row instead of creating a new row.
                 aRow = dsMain.Tables[dataTable].Rows[FindRow(anEmp,dataTable)];
         //2.4.1 Write the code to Fill this row for the Edit operation by calling the FillRow method
         }
}
```

2.5 Build Update Parameters that will communicate with the SQL Commands – in this case, the Update Command. The code is given below – complete the code.

```
private void Build_UPDATE_Parameters(Employee anEmp)
   {
     SqlParameter param = default(SqlParameter);
     param = new SqlParameter("@Name", SqlDbType.NVarChar, 100, "Name");
     param.SourceVersion = DataRowVersion.Current;
     daMain.UpdateCommand.Parameters.Add(param);
      // 2.5.1 TO DO: -: Do for all fields other than ID and EMPID as for the Build Insert parameters.
      Ofcourse, depending on the role. The code is similar to the Build_INSERT_Parameters that you created
     param = new SqlParameter("@Original_ID", SqlDbType.NVarChar, 15, "ID");
     param.SourceVersion = DataRowVersion.Original;
     daMain.UpdateCommand.Parameters.Add(param);
   }
```

2.6 After the Update Parameters, we need to create an *Update Command* method that must be used to insert values into one of the three tables. Assumption is that the ID and EMPID cannot be changed. The code is given below – complete the code.

```
private void Create_UPDATE_Command(Employee anEmp)
   {
     switch (anEmp.role.getRoleValue)
     {
       case Role.RoleType.Headwaiter:
         daMain.UpdateCommand = new SqlCommand ("UPDATE HeadWaiter SET Name =@Name,Phone =@Phone, Role
         =@Role, Salary =
                                  @Salary " + "WHERE ID =  @Original_ID", cnMain);
         break;
      //2.6.1 TO DO: -: Do the same for the other tables
     }
      //2.6.2 Write the code to call the Build_UPDATE_Parameters method
   }
```

2.7 The final step in the **EmployeeDB** is to call the *Create_UPDATE_Command* method in the UpdateDataSource method below the *Create_INSERT_Command* method.

3. We will now focus on the EmployeeController class to include the Edit functionality.

3.1 In the **EmployeeController** class we need to add another search function called *FindIndex* to find the index of the employee in the collection. This method receives an employee object *anEmp* and returns the index of the employee in the collection. The is similar to the *Find* method

```
//3.1.1 write the header of the method.
    {
      int counter = 0;
      bool found = false;
      found = (anEmp.ID== employees[counter].ID);
      while (//3.1. 2 if the employee has not been found and the search has not reached the end of the
      collection)
      {
        //3.1.3 Increment the counter
        found = (anEmp.ID == employees[counter].ID);
      }
      if (found)
      {
        return counter;
      }
      else
      {
        return -1;
      }
```

3.2 We will now change the parameter list of the *DataMaintenance* method to include the database operations. Then, we change the *DataMaintenance* method to use a switch statement to cater for the different actionswhen adding, editing or deleting.

```
{//3.2.1 write the code to change the DataMaintenance method to receive an Employee object anEmp
and a variable operation of type DB.DBOperation
//3.2.2 write the code to declare a variable index and initialise to zero
//3.2.3 write the code to perform a given database operation to the dataset in memory by
        calling the DataSetChange method.
  switch (operation)
  {
    case DB.DBOperation.Add:
       employees.Add(anEmp); //*** Add the employee to the Collection
        break;
    case DB.DBOperation.Edit:
       //3.2.4 write the code to call the FindIndex method for this employee and assign to index
       //3.2.5: write the code to replace employee at this index with the
       updated employee (anEmp)
       break;
```

4. We will make changes to the EmployeeListingForm, for example to reset the EmployeeListingForm to a View state and hide thetextboxes and labels – to make the UI ready and presentable for the user

4.1 Double click on the Edit Button to create the editButton_Click event.

a) In the editButton_Click event set the Formstate to edit

b) Call the *EnableEntities* method with the value *true*

4.2 Create a *submitButton_Click* event. Once the user has made the changes to the specific employeethis button may be clicked. The code is given below – complete the code

```
{

  //4.2.1 Populate the employee object of the form with the contents of the textboxes
  (Create a  method PopulateObject similar to the one in the EmployeeForm class to
  help you to populate the employee object. After creating the method, call the
  method here.)

   / //4.2.2 If the form is in the Edit state
    {
      //4.2.3  Call the DataMaintenance method of the EmployeeController by passing it this object and  the
               database operation that you would like to do, as parameters.

    }
    else// you will write the delete code – this to be done in Workshop 9
    {

    }
      //4.2.4 Call the FinalizeChanges method
      //4.2.5 Clear all the textboxes.
      //4.2.6 Set the FormStates, state, back to the view state.
      //4.2.7 Reset the form to hide all labels, textboxes and buttons by calling the ShowAll method
      (passing it false and roleValue as parameters)
      //4.2.8 Refresh the ListView – by setting it up again! (simply call the setUpEmployeeListView method)
   }
```

4.3 In the EmployeeListingForm_Activated event, make sure that, if not already done so, perform the following actions:

```
 // 4.3.1  The view of the  employeeListView is the Details view.
//4.3.2 call the setUpEmployeeListView method
//4.3.3 call the ShowAll method to reset the controls.
```

4.4 Set up the EmployeeListingForm constructor as follows – if not already done so:

```
InitializeComponent();
employeeController = empController;
this.Load += EmployeeListingForm_Load;
this.Activated += EmployeeListingForm_Activated;
this.FormClosed += EmployeeListingForm_FormClosed;
state = FormStates.View;
```

Debug and RUN – try to select an employee, edit their details and check if the information was edited in the Dataset

Some notes on the Building of update parameters

```
private void Build_UPDATE_Parameters(Employee anEmp)
    {
      //---Create Parameters to communicate with SQL UPDATE

      //Step 1: Create a variable of type SqlParameter
        SqlParameter param = default(SqlParameter);       //An Object that is the value of the parameter.
                                                          //The default value is null.
                                                          // SqlParameter param = null;
          /* The SqlParameter represents a parameter to a SqlCommand
           * and optionally its mapping to DataSet columns.
           * This class is found in the "System.Data.SqlClient" namespace.*/



      //Step 2: //Creating instance of SqlParameter - define properties of the SqlParameter class
         param = new SqlParameter("@Name", SqlDbType.NVarChar, 100, "Name");

          /* ParameterName: It is used to specify a parameter name...."@Name"
           * SqlDbType: It is used to set the SQL Server Datatypes for a given parameter....SqlDbType.NVarChar
           * Size: It is used to set the maximum size of the value of the parameter.
           * Value: It is used for assigning or getting the value of the parameter.
           */



      //Step 3: we inform the update command what version of the value needs to be loaded.
        param.SourceVersion = DataRowVersion.Current;

        /* we inform the update command what version of the value needs to be loaded.
         *  to do this we use the: SqlParameter.SourceVersion Property
         *  This property is used by the SqlDataAdapter.UpdateCommand during
         *  an update to determine whether the original or current value is
         *  used for a parameter value. This lets primary keys be updated.
         *  This property is set to the version of the DataRow used by
         *  the DataRow.Item property, or one of the DataRow.GetChildRows methods of the DataRow object.
         *  gets or sets the DataRowVersion to use when you load Value
         *  One of the DataRowVersion values. The default is Current. You can find other values here


      /*Step 4: we call the Update Command to add the parameter to the Parameters collection.
       *      The SqlParameterCollection.Add Method adds a SqlParameter to the SqlParameterCollection.*/
      daMain.UpdateCommand.Parameters.Add(param);
```