# 5. Database Programming

## Database basics and Data control

### Introduction to Relational database and ADO.NET

Relational database model eliminated some of the problems that were associated with standard files and other database designs. By using the relational model, data redundancy can be reduced, which saves disk storage and leads to efficient data retrieval. You can also view and manipulate data in a way that is both intuitive and efficient. A relational database uses tables to store and manipulate data.

**ADO.NET** (ActiveX Data Object .NET) is the primary data access API for the .NET Framework. It provides the classes that you as you develop database application with C# and other .Net languages. These classes can be divided into two

* **The .NET data provider** which provide the classes that you use to access the data in database
* **Dataset**  which provide the classes that you can use to store and work with data in your application

**The .NET data providers**

A .NET data provider is a set of classes that enables you to access data that’s managed by a particular database server. All .NET data providers must include core classes for creating the four types of objects.

Connection establishes a connection to a database server

Command represents an individual SQL statement that can be executed against the database

Data reader provides read-only forward access to the data in a database

Data adapter provides the link between the command and connection objects and a dataset object

Data provider included with the .NET Framework

SQL Server System.Data.SqlClient enables access MS SQL Server database

OLEDB System.Data.OleDb enables access to any database that supports OLEDB

ODBC System.Data.Odbc enables access to any database that supports ODBC

Oracle System.Data.OracleClient enables access to Oracle database

Class name for data provider

**Object SQL Server OLEDB ODBC Oracle**

Connection SqlConnection OleDbConnection OdbcConnection OracleConnection

Command SqlCommand OleDbCommand OdbcCommand OracleCommand Data Reader SqlDataReader OleDbDataReader OdbcDataReader OracleDataReader

Data adapter SqlDataAdapter OleDbDataAdapter OdbcDataAdapter OracleDataAdapter

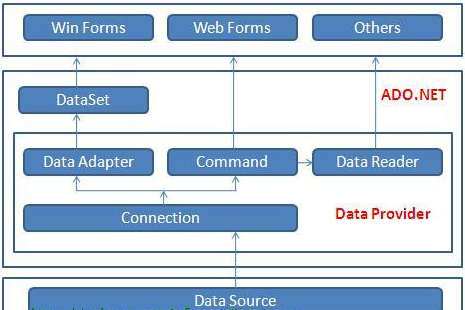


Fig 5.1: Components of ADO.NET

**DataSet**

DataSet provides a disconnected representation of result sets from the Data Source, and it is completely independent from the Data Source. DataSet provides much greater flexibility when dealing with related Result Sets.

Dataset is structured much like a relational database. It can contain one or more tables, and each table can contain one or more columns and rows.

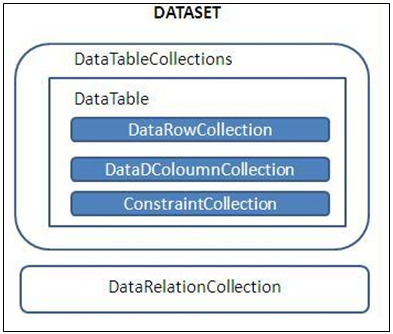


Fig 5.2: the basic dataset object

You can also work with the data in a database without using a data adapter. Use command and connection objects to access database. Instead of using a data adapter to execute the commands, you can execute the commands directly. When you do that, you also have to provide code to handle the result of the command. If you issue a command that contains an Insert, Update, or Delete statement the result is an integer that indicates the number of rows that were affected by the operation. You can use that operation to determine if the operation was successful.

If you execute a command that contains a Select statement, the result is a result set that contains the rows you requested. To read through the rows in the result set, use a data reader object.

The steps for executing database command in ADO.NET are:

1. Create connection object
2. Open the connection
3. Create a command object encapsulating both Sql command and the connection
4. Call a method on the command object to execute the command
5. Close the connection by calling the close on the connection object

**ADO.NET Objects**

You can create ADO.NET objects

* Write the code yourself
* From data source listed in the Data Sources window
  + Add Data Source using Data Source Configuration wizard

### Creating ADO.NET MS SQL Server Database objects

1. **Create a connection object**

The SqlConnection Object is handling the part of physical communication between the C# application and the SQL Server Database. An instance of the SqlConnection class in C# is supported the Data Provider for SQL Server Database. The SqlConnection instance takes Connection String as an argument and passes the value to the Constructor statement.

connetionString="Data Source=ServerName;

Initial Catalog=DatabaseName;User ID=UserName;Password=Password"

cnn = new SqlConnection(connetionString);

cnn.Open();

When the connection is established, SQL Commands will execute with the help of the Connection Object and retrieve or manipulate the data in the database. Once the Database activities over, Connection should be closed and release the Data Source resources.

cnn.Close();

The Close() method in SqlConnection Class is used to close the Database Connection. The Close method rolls back any pending transactions and releases the Connection from the SQL Server Database.

string connetionString = null;

SqlConnection conn ;

connetionString = "Data Source=ServerName;Initial Catalog=DatabaseName;User ID=UserName;Password=Password";

conn = new SqlConnection(connetionString);

try

{

conn.Open();

MessageBox.Show ("Connection Opened! ");

conn.Close();

}

catch (Exception ex)

{

MessageBox.Show("Cannot open connection ! ");

}

1. **Create a command object**

The Command Object in ADO.NET executes SQL statements and Stored Procedures against the data source specified in the C# Connection Object. The Command Object requires an instance of a C# Connection Object for executing the SQL statements.

Some important built in methods uses in the Command Object to execute the SQL statements

ExecuteNonQuery used for executing statements that do not return result sets (i.e. statements like insert data, update data etc.)

returns the no. of rows affected by the command

ExecuteReader to perform database queries and obtain the results

Returns a DataReader object

DataReader has methods and properties that you can call to iterate over the result set, such as GetValue, GetName, Read, GetDecimal, …

ExecuteScalar to retrieve a single value from Database after the execution of the SQL Statement

returns the first row of the first column in the result set

used to execute SQL functions such as COUNT, AVG, MIN, MAX, and SUM which returns a single row single, column result set

The following example uses a **SqlCommand** object to delete a record from the LibrarySys database’s “Book” table using an SQL DELETE command

SqlConnection conn = new SqlConnection

    ("server=localhost;database=LibrarySys;uid=sa;pwd=");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand ();

    cmd.CommandText = "delete from Book where ISBN = 'BU1032'";

    cmd.Connection = conn;

    cmd.ExecuteNonQuery (); // Execute the command

}

catch (SqlException ex) {

    // TODO: Handle the exception

}

finally {

    conn.Close ();

}

You can make your code more concise by creating a SqlCommand object and initializing its Connection and CommandText properties in one step:

SqlConnection conn = new SqlConnection

    ("server=localhost;database=pubs;uid=sa;pwd=");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand

        ("delete from titles where title\_id = 'BU1032'", conn);

    cmd.ExecuteNonQuery (); // Execute the command

}

catch (SqlException ex) {

    // TODO: Handle the exception

}

finally {

    conn.Close ();

}

Here’s an example that uses **ExecuteNonQuery** to add a record to the LibrarySys database’s “Book” table using an INSERT command:

SqlConnection conn = new SqlConnection

    ("server=localhost;database=LibrarySys;uid=sa;pwd=");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand

        ("insert into Book (ISBN, title, author, price) " +

        "values ('DBU1001', 'Programming Microsoft .NET', " +

        "'Abebe', 45)", conn);

    cmd.ExecuteNonQuery ();

}

catch (SqlException ex) {

    // TODO: Handle the exception

}

finally {

    conn.Close ();

}

The following example writes the largest Book price recorded in the LibrarySys database to a TextBox:

SqlConnection conn = new SqlConnection

    ("server=localhost;database=LibrarySys;uid=sa;pwd=");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand

        ("select max (price) from Book", conn);

    decimal amount = (decimal) cmd.ExecuteScalar();//cast object to decimal

    txtprice.Text = amount.ToString(“C”);

}

catch (SqlException ex) {

    //Todo: Handle exception

}

finally {

    conn.Close ();

}

Another common use for ExecuteScalar is to retrieve BLOBs (binary large objects) from databases. The following example retrieves an image from the “photo” field of the LibrarySys database’s “Customer” table and encapsulates it in a bitmap:

MemoryStream stream = new MemoryStream ();

SqlConnection conn = new SqlConnection

    ("server=localhost;database=LibrarySys;uid=sa;pwd=");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand

        ("select photo from customer where custId='DBU1001'", conn);

    byte[] blob = (byte[]) cmd.ExecuteScalar ();

    stream.Write (blob, 0, blob.Length);

    Bitmap bitmap = new Bitmap (stream);

    // TODO: Use the bitmap

pictureBox1.Image = bitmap;

    bitmap.Dispose ();

}

catch (SqlException ex) {

    // TODO: Handle the exception

}

finally {

    stream.Close ();

    conn.Close ();

}

Once the bitmap is created, you can do whatever you want with it: display it in a Windows form, stream it back in an HTTP response, or whatever. Note that in order for this sample to compile, you must include using statements that import the System.IO and System.Drawing namespaces as well as System and System.Data.SqlClient.

You might be interested in knowing how to write BLOBs to databases, too. The secret is to call **ExecuteNonQuery** on a command object that wraps an INSERT command containing an input parameter whose type is byte[]. To demonstrate, the following example inserts a record into the LibrarySys database’s “Customer” table and includes a BLOB in the record’s “photo” field:

FileStream stream = new FileStream ("photo.jpg", FileMode.Open);

byte[] blob = new byte[stream.Length];

stream.Read (blob, 0, (int) stream.Length);

stream.Close ();

SqlConnection conn = new SqlConnection

    (""Data Source=CoolTech\sqlexpress;Initial Catalog=LibrarySys;Integrated Security=True" =");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand

        ("insert into customer (CustId, photo) values ('DBU1001', @photo)",

        conn);

    cmd.Parameters.Add ("@photo", blob);

    cmd.ExecuteNonQuery ();

}

catch (SqlException ex) {

    // TODO: Handle the exception

}

finally {

    conn.Close ();

}

The following example uses ExecuteReader and the resultant SqlDataReader to write the titles of all the books listed in the LibrarySys database to a ListBox:

SqlConnection conn = new SqlConnection

    ("server=localhost;database=LibrarySys;uid=sa;pwd=");

try {

    conn.Open ();

    SqlCommand cmd = new SqlCommand ("select \* from Book", conn);

    SqlDataReader reader = cmd.ExecuteReader();

    while (reader.Read ())

        listBox1.Items.Add(reader["Title"].ToString());

}

catch (SqlException ex) {

    Console.WriteLine (ex.Message);

}

finally {

    conn.Close ();

}

Each call to SqlDataReader.Read returns one row from the result set. This example uses a property indexer to extract the value of the record’s “Title” field. Fields can be referenced by name or by numeric index (0-based, of course).

**Closing a DataReader**

A potential gotcha regarding DataReaders has to do with their Close methods. By default, DataReader.Close does not close the connection encapsulated in the command object that created the DataReader. reader.Close (); // Does NOT close the connection!

DataReader.Close closes the DataReader, which frees the connection associated with the DataReader so that it can be used again.

**Transacted Commands**

Transacted database operations are an important element of many data-driven applications. A transaction is simply two or more otherwise independent units of work grouped together into one logical unit. A classic example is an application that transfers funds from one bank account to another by debiting money from one account (that is, one database record) and crediting it to another. The updates should be performed within the scope of a transaction. Why? So that if one of the operations fails, the other will fail (or be rolled back) too.

SqlTransaction trans = null;

SqlConnection conn = new SqlConnection

    ("server=localhost;database=mybank;uid=sa;pwd=");

try {

    conn.Open ();

    // Start a local transaction

    trans = conn.BeginTransaction (IsolationLevel.Serializable);

    // Create and initialize a SqlCommand object

    SqlCommand cmd = new SqlCommand ();

    cmd.Connection = conn;

    cmd.Transaction = trans;

    // Debit $1,000 from account 1111

    cmd.CommandText = "update accounts set balance = " +

        "balance - 1000 where account\_id = '1111'";

    cmd.ExecuteNonQuery ();

    // Credit $1,000 to account 2222

    cmd.CommandText = "update accounts set balance = " +

        "balance + 1000 where account\_id = '2222'";

    cmd.ExecuteNonQuery ();

    // Commit the transaction (commit changes)

    trans.Commit ();

}

catch (SqlException) {

    // Abort the transaction (roll back changes)

    if (trans != null)

        trans.Rollback ();

}

finally {

    conn.Close ();

}

**Parameterized Commands**

It’s not unusual for an application to execute the same command on a database repeatedly, varying only the value or values used in the command.

SQL programmers often use parameterized commands (frequently referred to as “parameterized queries”) to code redundant commands, especially commands whose input values come from user input. Here’s a parameterized version of the previous section’s INSERT command:

try {

    conn.Open ();

    // Create and initialize a SqlCommand object

    SqlCommand cmd = new SqlCommand

        ("update accounts set balance = balance + @amount " +

        "where account\_id = @id", conn);

    cmd.Parameters.Add ("@amount", SqlDbType.Money);

    cmd.Parameters.Add ("@id", SqlDbType.Char);

    // Debit $1,000 from account 1111

    cmd.Parameters["@amount"].Value = -1000;

    cmd.Parameters["@id"].Value = "1111";

    cmd.ExecuteNonQuery ();

    // Credit $1,000 to account 2222

    cmd.Parameters["@amount"].Value = 1000;

    cmd.Parameters["@id"].Value = "2222";

    cmd.ExecuteNonQuery ();

}

catch (SqlException ex) {

    // TODO: Handle the exception

}

finally {

    conn.Close ();

}

**Lab Exercise**

**Using appropriate connection, command, and data reader objects,**

Create a database “LibrarySys” contains ‘Book (isbn, Title, Author, Price)’, and ‘Customer (CustId, Name,fatherNmae, Gender,DateOfBirth, Photo)’ tables.

To create a database use the Management Studio if you have installed in your computer or

Use Server Explorer tab or use **View-> Server Explorer**

Right Click on **Data Connection** -> click “**Create New SQL Server database”**

Enter Server name and Database name ->Click **OK**

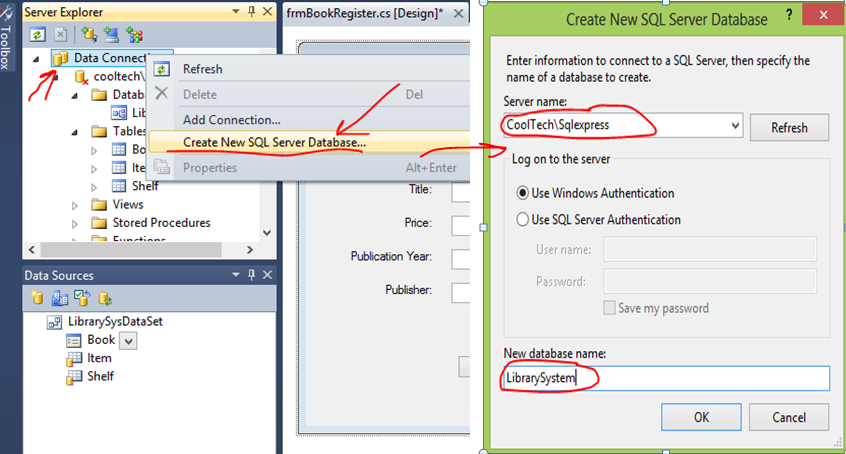


Fig 5.3: Creating a database in Visual Studio IDE

Create a project ‘LibrarySys’, Add the required controls as shown in the figure below and set basic properties of the Form and controls.

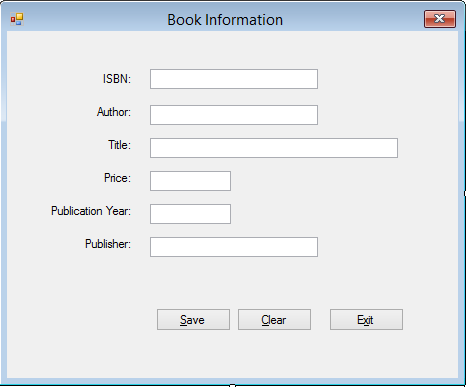


Fig 5.4: Book registration Form

Write an event handler for the “Save” button click event. When the user click the ‘Save” button, it save the data in to the Book table you created earlier.

Add a customer registration form to your project, design the UI and write a code that save the customers information in to a database the same manner as book registration as you done above.

### Working with Data Source

**Create a data source**

1. Open the Data Source windows

**Click** on the **Data Source** tab or **Data ->Data Source** command

1. Click on the Add New Data Source link, if your adding data source for the first time to your project, or use **Data->Add New Data Source** command

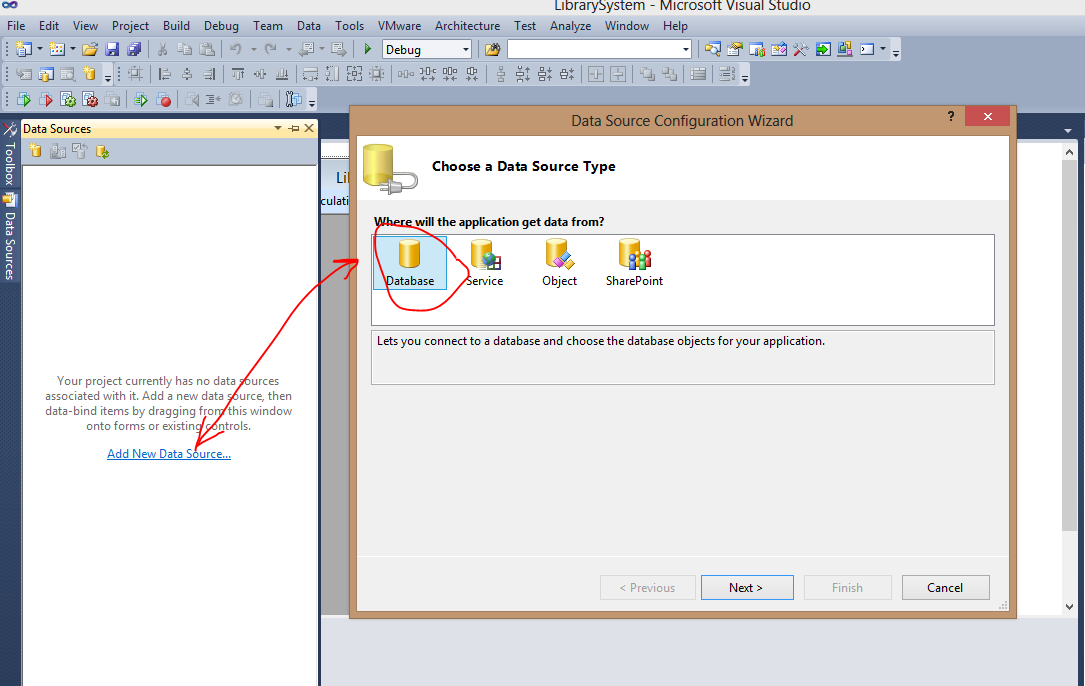


Fig 5.5: Starting a Data Source wizard and choose the data source type

1. Choose the data source type, Click **Next**
2. Choose the database model, (Dataset or Entity Data Model), select **dataset**, click Next

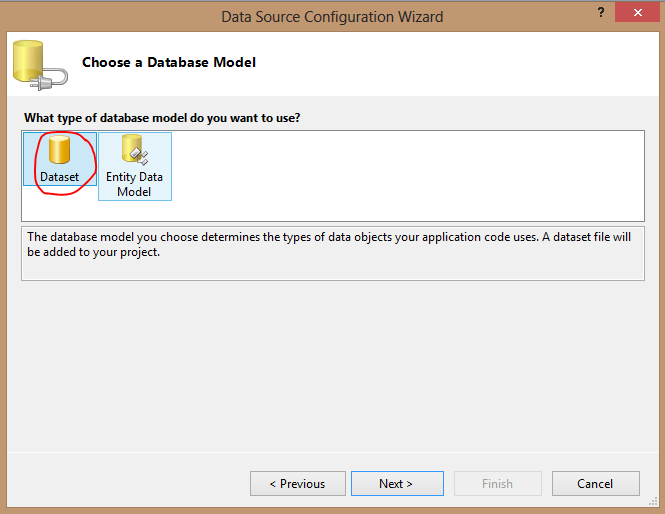


Fig 5.6: Choosing a database model

1. Choose the data connection, Click New Connection -> select Data source and click Continue

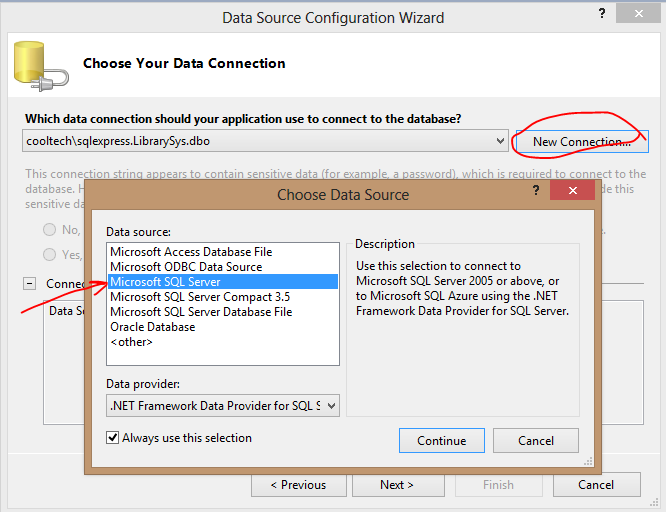


Fig 5.7: Choosing the data connection and data source

1. Add Connection: Select/Enter database server, Select/Attach Database, click Ok

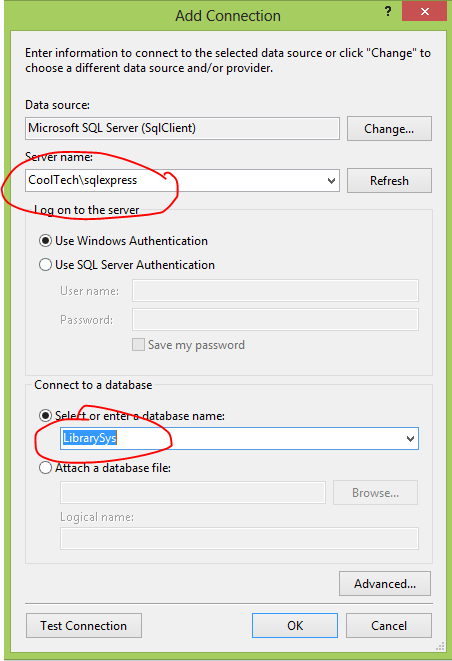


Fig 5.8: Add connection

1. Choose your data connection, have a look at your connection string, click Next

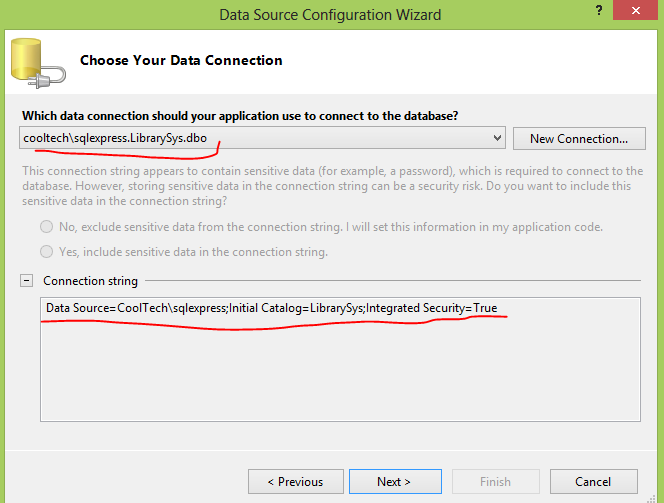


Fig 5.9: selecting data connection and connection string

1. Save connection string to application configuration file, Click Next

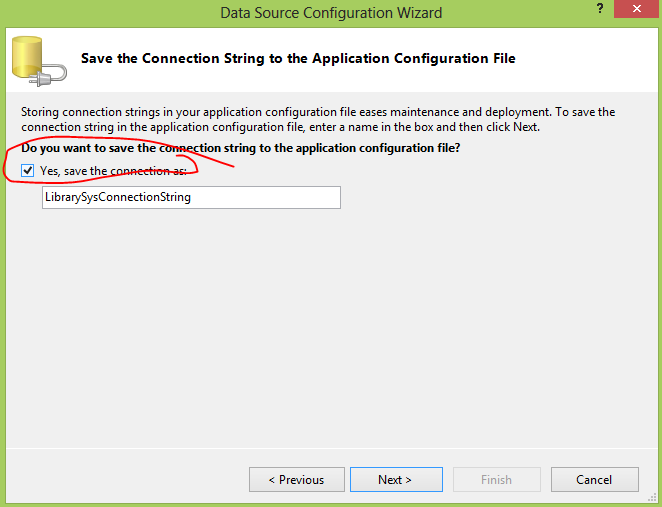


Fig 5.10: saving connection string in app config file

1. Choose Database object you need to add to your project

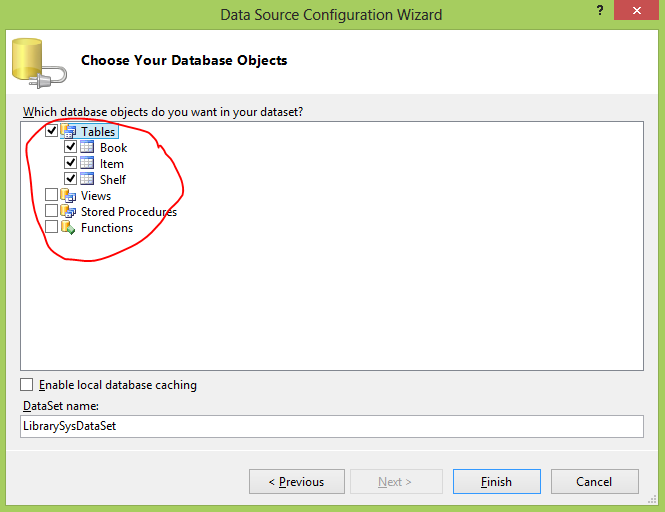


Fig 5.11: Choosing your database object

1. Click Finish, Enjoy with drag and drop of your database object

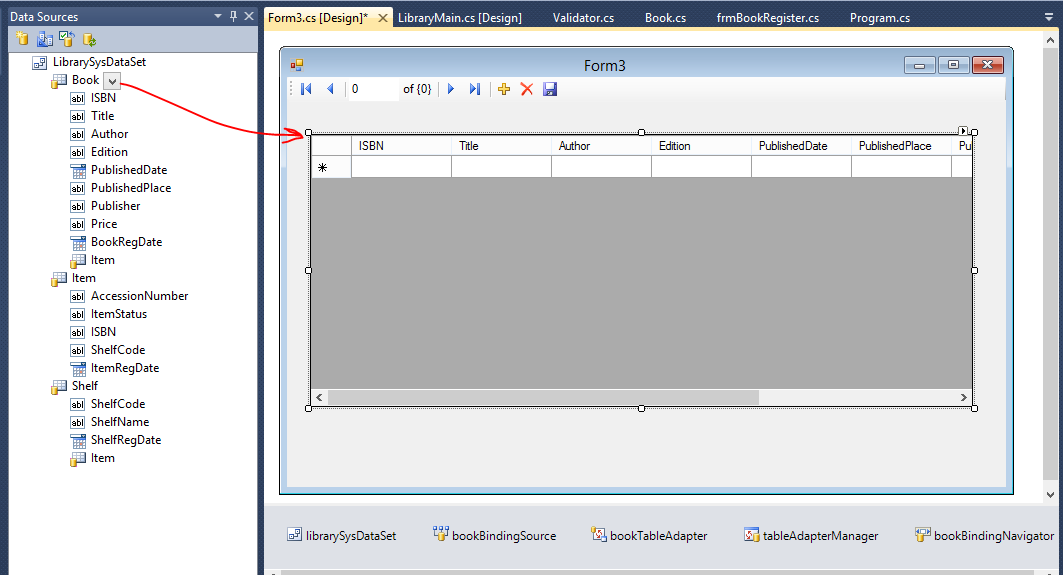


Fig 5.12: Use your database object, by drag and drop on to your Forms/controls

**Lab Exercise**

**Working on Data Source**

Add a Customer information display form to your project, you created above. Add the customer database object from the data source to the form in order to display customer information in a GridView. Run and test your project. Have a look how the data source works.

### DataSet and DataAdapters

ADO.NET’s SqlDataReader class provides stream-based access to the results of database queries. Streaming access is fast and efficient, but it’s also read-only and forward-only. You can’t, for example, back up and reread the previous record with a DataReader or change the results and write them back to the database. That’s why ADO.NET supports set-based data access as well as stream-based data access. Set-based accesses capture an entire query in memory and support backward and forward traversal through the result set. They also let you edit the data obtained through database queries, propagate the changes back to the data source, and much, much more.

Set-based data accesses revolve around two classes: DataSet, which is the equivalent of an in-memory database and is defined in the System.Data namespace, and DataAdapter, which serves as a bridge between DataSets and physical data sources.

**DataSet Class**

Think of a DataSet as an in-memory database. The actual data is stored in DataTable objects, which are analogous to tables in a database. The DataSet.Tables property exposes a list of the DataTables in a DataSet. Records in a DataTable are represented by DataRow objects, and fields are represented by instances of DataColumn. DataTable properties named Rows and Columns expose the collections of DataRows and DataColumns that make up the table.

**DataSets vs. DataReaders**

One of the most common questions that developers ask about ADO.NET is which is best, DataSets or DataReaders? The answer is: it depends. If your intention is simply to query a database and read through the records one at a time until you find the one you’re looking for, then DataReader is the right tool for the job. DataReaders, unlike DataSets, retrieve only the data that you actually use, and they don’t consume memory by storing every record that you read. If, however, you intend to use all the query results (perhaps because you’re displaying them in a table), you need the ability to iterate backward and forward through a result set, or you want to cache the result set in memory, use a DataSet.

As a result to the DataSet vs. DataReader debate, realize that many controls that support data binding to DataSets are perfectly capable of binding to DataReaders as well. Many examples in the .NET Framework SDK and elsewhere that demonstrate data binding to ASP.NET server controls show controls binding to DataSets:

DataSet ds = new DataSet ();

// TODO: Initialize the DataSet

MyDataGrid.DataSource = ds;

MyDataGrid.DataBind ();

the same code can be implemented more efficiently with a DataReader:

SqlDataReader reader = cmd.ExecuteReader ();

MyDataGrid.DataSource = reader;

MyDataGrid.DataBind ();

**DataAdapter Classes**

DataAdapter’s purpose is to perform database queries and create DataTables containing the query results. It’s also capable of writing changes made to the DataTables back to the database.

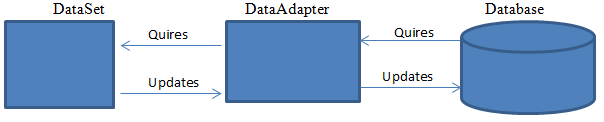


Fig 5.13: The role of DataAdapter

**Initializing a DataSet from a Database: DataAdapter.Fill**

The following code sample is representative of the numerous DataSet examples found in the .NET Framework SDK and on Web sites that cater to .NET developers:

SqlDataAdapter adapter = new SqlDataAdapter ("select \* from Book",

    "server=localhost;database=LibrarySys;uid=sa;pwd=");

DataSet ds = new DataSet ();

adapter.Fill (ds, "Book");

What does this code do? A lot. Here’s a blow-by-blow of what happens inside the call to Fill:

1. Fill opens a connection to the Pubs database using the connection string passed to SqlDataAdapter’s constructor.
2. It performs a query on the Pubs database using the query string passed to SqlDataAdapter’s constructor.
3. It creates a DataTable named “Book” in the DataSet.
4. It initializes the DataTable with a schema that matches that of the “Book” table in the database.
5. It retrieves all the records produced by the query and writes them to the DataTable.
6. It closes the connection to the database.

**DataTable**

The following code iterates through the Tables collection of a DataSet named ds and outputs the name of each table that it encounters:

foreach (DataTable table in ds.Tables)

     ltxtBox1.Items.Add (table.TableName.ToStriing());

Individual DataTables in a DataSet can be referenced by name or 0-based index.

The next example retrieves the first DataTable from a DataSet and writes the value of the first column in every row to a listBox:

DataTable table = ds.Tables[0];

foreach (DataRow row in table.Rows)

     ltxtBox1.Items.Add(row[0].ToString());

Columns, too, can be referenced by name as well as numeric index. Thus, if the name assigned to the first column in the DataTable is “CustID,” the preceding example could be rewritten this way:

DataTable table = ds.Tables[0];

foreach (DataRow row in table.Rows)

    ltxtBox1.Items.Add(row["CustID"].ToString());

Enumerating a DataTable’s columns is equally simple:

DataTable table = ds.Tables[0];

foreach (DataColumn col in table.Columns)

    Console.WriteLine ("Name={0}, Type={1}",

        col.ColumnName, col.DataType);

**Inserting Records into a DataTable**

One way to insert records into a database is to call ExecuteNonQuery on a Command object wrapping an INSERT command, as demonstrated in the first half of this chapter. You can also insert records into databases using DataSets. The general approach is to perform a query with DataAdapter.Fill, add records to the resulting DataTable, and write the changes to the database. You already know how to call Fill. Let’s talk about adding records to a DataTable.

The following example adds a record to a DataTable created from the librarySys database’s “Book” table:

SqlDataAdapter adapter = new SqlDataAdapter ("select \* from Book",

    "server=localhost;database=LibrarySys;uid=sa;pwd=");

DataSet ds = new DataSet ();

adapter.Fill (ds, "Book");

// Create a new DataRow

DataTable table = ds.Tables["Book"];

DataRow row = table.NewRow ();

// Initialize the DataRow

row["ISBN"] = "dbu1001";

row["Title"] = "Programming Microsoft .NET";

row["Author"] = "Abebe";

row["price"] = 45.00;

row["PuplishedDate"] = new DateTime (2002, 5, 1);

// Add the DataRow to the DataTable

table.Rows.Add (row);

**Selecting Records in a DataTable**

A smarter way to find the records is to use the DataTable.Select method.

The following statement returns an array containing a single DataRow—the one added to the table with Add:

DataRow[] rows = table.Select ("ISBN = 'DBU1001'");

This statement selects (returns) all DataRows whose “Price” field contains a value less than 10:

DataRow[] rows = table.Select ("price < 10.00");

And this one selects records whose “PuplishedDate” fields hold dates on or after January 1, 2000:

DataRow[] rows = table.Select ("PuplishedDate >= '#1/1/2000#'");

If you want to know how many rows Select returned, read the array’s Length property.

**Updating Records in a DataTable**

The following example selects all the records in the LibrarySys database’s “Book” table with publication date later or on January 1, 2000 and adds $10.00 to their price:

SqlDataAdapter adapter = new SqlDataAdapter ("select \* from Book",

    "server=localhost;database=LibrarySys;uid=sa;pwd=");

DataSet ds = new DataSet ();

adapter.Fill (ds, "Book");

DataRow[] rows = table.Select ("PuplishedDate >= '#1/1/2000#'");

foreach (DataRow row in rows)

    row["price"] = (decimal) row["price"] + 10.00m;

**Deleting Records from a DataTable**

Deleting records from a DataTable is a simple matter of calling Delete on each DataRow that you want to remove. The next example deletes all rows whose publication date equal to null:

SqlDataAdapter adapter = new SqlDataAdapter ("select \* from titles",

    "server=localhost;database=LibrarySys;uid=sa;pwd=");

DataSet ds = new DataSet ();

adapter.Fill (ds, "Book");

DataRow[] rows =

    table.Select ("isnull (PublishedDate, 0) = 0");

foreach (DataRow row in rows)

    row.Delete ();

**Propagating Changes Back to a Database: DataAdapter.Update**

Inserts, updates, and deletes performed on a DataTable do not automatically propagate back to the database. If you want changes written back to the database, you have to take matters into your own hands. Fortunately, the DataAdapter.Fill method makes your job incredibly simple.

Here’s a code sample demonstrating how to make changes to a database using a DataSet and a DataAdapter:

SqlDataAdapter adapter =

    new SqlDataAdapter ("select \* from titles",

    "server=localhost;database=LibrarySys;uid=sa;pwd=");

SqlCommandBuilder builder = new SqlCommandBuilder (adapter);

DataSet ds = new DataSet ();

adapter.Fill (ds, "Titles");

// Insert a record

DataTable table = ds.Tables["Book"];

DataRow row = table.NewRow ();

row["ISBN"] = "DBU1001";

row["title"] = "Programming Microsoft .NET";

row["price"] = 59.99m;

table.Rows.Add (row);

// Update the database

adapter.Update (table);

The DataAdapter’s Update method examines each row in the table passed to it and writes rows that were inserted, updated, or deleted since the last update (or since the last time the table’s AcceptChanges method was called) to the database. If a DataSet contains multiple DataTables that underwent modification, pass the entire DataSet to Update and all the changes will be propagated at once

Update call a method named GetChanges to create a DataSet or DataTable containing only rows that were inserted, modified, or deleted. They then pass the “delta” DataSet or DataTable to Update, as shown here:

// Update the database

DataTable delta = table.GetChanges ();

adapter.Update (delta);

This approach works, but it isn’t necessary. Update is smart enough to ignore rows that weren’t changed in a DataTable containing a mixture of modified and unmodified rows. Where GetChanges becomes interesting is when you want to control the order in which changes are propagated back to the database. If you want to make sure DELETEs are performed before INSERTs to avoid duplicate key errors, for example, you can do this:

// Update the database

DataTable deletes = table.GetChanges (DataRowState.Deleted);

adapter.Update (deletes);

DataTable inserts = table.GetChanges (DataRowState.Added);

adapter.Update (inserts);

Another use for GetChanges is to minimize the amount of data passed between machines when the update won’t be performed locally. Passing a DataSet or DataTable containing just the rows that changed is more efficient than passing a DataSet or DataTable containing both modified and unmodified rows.

**CommandBuilder Classes**

Now ask yourself a question. How does Update physically update the database? The short answer is that it executes SQL INSERT commands for rows added to a DataTable, UPDATE commands for rows that were modified, and DELETE commands for rows that were deleted. But where do the INSERT, UPDATE, and DELETE commands come from? Are they manufactured out of thin air?

Close. They’re manufactured by a SqlCommandBuilder object. Note the following statement from the previous code sample:

SqlCommandBuilder builder = new SqlCommandBuilder (adapter);

If you omit this statement, Update throws an exception. A DataAdapter has four very important properties that control how it communicates with a database:

* SelectCommand, which encapsulates the command the DataAdapter uses to perform queries
* InsertCommand, which encapsulates the command the DataAdapter uses to insert rows
* UpdateCommand, which encapsulates the command the DataAdapter uses to update rows
* DeleteCommand, which encapsulates the command the DataAdapter uses to delete rows

When you create a DataAdapter this way:

SqlDataAdapter adapter =

    new SqlDataAdapter ("select \* from book",

    "server=localhost;database=LibrarySys;uid=sa;pwd=");

the constructor initializes SelectCommand with a SqlCommand object wrapping the query string, but it leaves InsertCommand, UpdateCommand, and DeleteCommand set to null. When Update is called and it finds these properties still equal to null, it asks the SqlCommandBuilder to provide it with the commands it needs. If there is no SqlCommandBuilder, Update is powerless to update the database and indicates as much by throwing an exception.

SqlCommandBuilder generates INSERT, UPDATE, and DELETE commands on the fly based on information inferred from the DataAdapter’s SelectCommand. The commands that they generate are simple dynamic SQL commands. You can see these commands for yourself by calling the command builder’s GetInsertCommand, GetUpdateCommand, and GetDeleteCommand methods and inspecting the command text found inside the returned command objects:

string insert = builder.GetInsertCommand ().CommandText;

string update = builder.GetUpdateCommand ().CommandText;

string delete = builder.GetDeleteCommand ().CommandText;

In the vast majority of cases, a builder’s auto-generated commands work just fine. However, if you do a lot of database updating with DataAdapters, you might achieve a performance boost by coding your own INSERT, UPDATE, and DELETE commands in stored procedures, wrapping the stored procedures in SqlCommand and assigning those objects to the adapter’s InsertCommand, UpdateCommand, and DeleteCommand properties. The DataAdapter will respond by using your stored procedures to do its updating.

**DataView**

The DataView provides different views of the data stored in a DataTable. That is we can customize the views of data from a DataTable. DataView can be used to sort, filter, and search the data in a DataTable , additionally we can add new rows and modify the content in a DataTable.

We can create DataView in two different ways. We can use the DataView Constructor, or you can create a reference to the DefaultView Property of the DataTable. The DataView constructor can be empty, or it can take either a DataTable as a single argument, or a DataTable along with filter criteria, sort criteria, and a row state filter.

***dv = ds.Tables[0].DefaultView;***

The following source code shows how to create a DataView in C#.

string connetionString = null;

SqlConnection connection ;

SqlCommand command ;

SqlDataAdapter adapter = new SqlDataAdapter();

DataSet ds = new DataSet();

DataView dv ;

string sql = null;

connetionString = "Data Source=localhost;Initial Catalog=librarySys;User ID=sa;Password= ";

sql = "Select \* from book";

connection = new SqlConnection(connetionString);

try

{

connection.Open();

command = new SqlCommand(sql, connection);

adapter.SelectCommand = command;

adapter.Fill(ds, "Create DataView");

dv = ds.Tables[0].DefaultView;

dataGridView1.DataSource = dv;

}

catch (Exception ex)

{

MessageBox.Show (ex.ToString());

}

DatView sort

try

{

adapter.Fill(ds, "Sort DataView");

dv = new DataView(ds.Tables[0], " Price > 300", "Price Desc", DataViewRowState.CurrentRows);

dataGridView1.DataSource = dv;

}

catch (Exception ex)

{

MessageBox.Show (ex.ToString());

}

DataView Search

try

{

adapter.Fill(ds, "Find Row DataView");

dv = new DataView(ds.Tables[0]);

dv.Sort = "ISBN";

int index = dv.Find("DBU1001");

if (index == -1)

{

MessageBox.Show ("Item Not Found");

}

else

{

MessageBox.Show(dv[index]["isbn"].ToString() + " " + dv[index]["Title"].ToString());

}

}

catch (Exception ex)

{

MessageBox.Show(ex.ToString());

}

DataView Add new row

try

{

adapter.Fill(ds, "Add New");

dv = new DataView(ds.Tables[0]);

DataRowView newRow = dv.AddNew();

newRow["isbn"] = “DBU1005”;

newRow["Title"] = "database System";

newRow["Price"] = 100;

newRow.EndEdit();

dv.Sort = "product\_id";

dataGridView1.DataSource = dv;

}

catch (Exception ex)

{

MessageBox.Show (ex.ToString());

}

In Similar way, you can use dataview to delete, update and ToTable method to create a new DataTable.

**Lab Exercise**

**Using DataSet and DataAdapter**

Add a Book information display form to your project, you created above. Add GridView control on the Form. Use DataSet and DataAdapter code to bind data source for the GridView and display book information.

Write a code for deleting and updating book information using Dataset and DataAdapter.

Run and test your project.

## Making Reports in event driven

Crystal Report is a Reporting Application that can generate reports from various Data Sources like Databases, XML files etc.. The Visual Studio.NET Integrated Development Environment comes with Crystal Reports tools. The Crystal Reports makes it easy to create simple reports, and also has comprehensive tools that you need to produce complex or specialized reports in csharp and other programming languages.

Crystal Reports is compatible with most popular development environments like C#, VB.NET etc. You can use the Crystal Reports Designer in Visual Studio .NET to create a new report or modify an existing report.

### Steps for Creating Crystal Reports

Start your first Crystal Reports

Create a new simple Crystal Reports for Book table from the database LibrarySys you created earlier. The report shows the whole table data in the Crystal Reports.

1. From **Project -> Add New Item** or Right Click on LibrarySystem project **Add->Add new Item** Command to bring Add New Item Dialog window
2. Select **Reporting** tab and Select **Crystal Reports,** available if you installed Crystal report on your computer

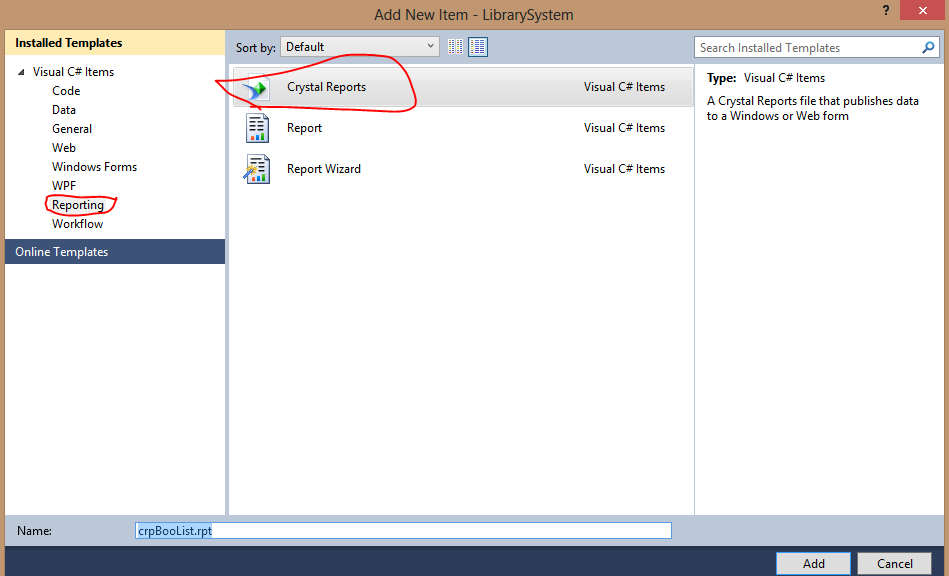


Fig 5.14: adding a Crystal report

1. Select Report type from Crystal Reports gallery, accept default and click OK

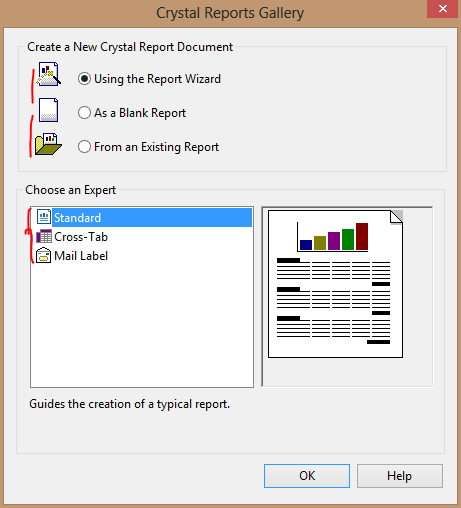


Fig 5.15: Selecting report type

1. Select the appropriate connection to your database, Click Next

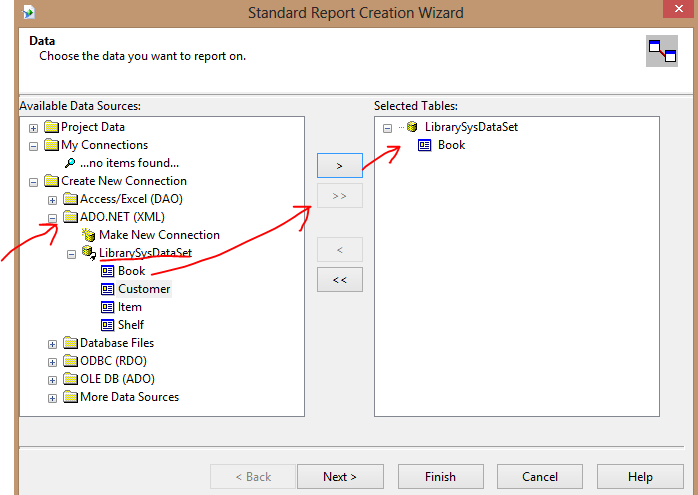


Fig 5.16: Selecting database connection

1. Select all fields from Book table to the right side list, Click Next

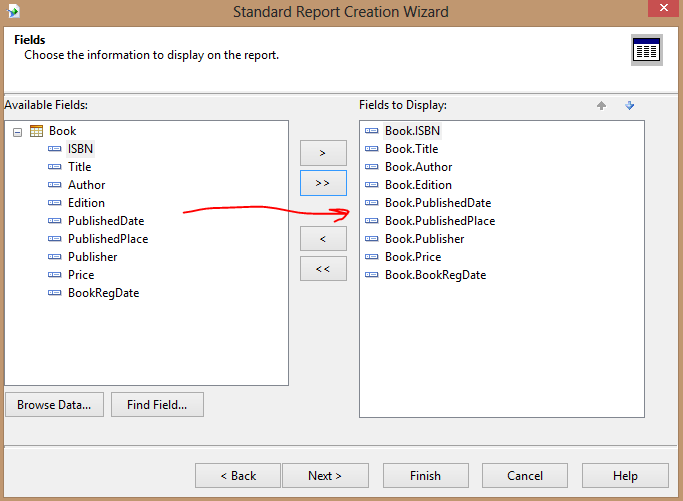


Fig 5.17: Selecting data fields

1. Click Finish Button. Then you can see the Crystal Reports designer window. You can arrange the design according your requirements. Your screen looks like the following picture.

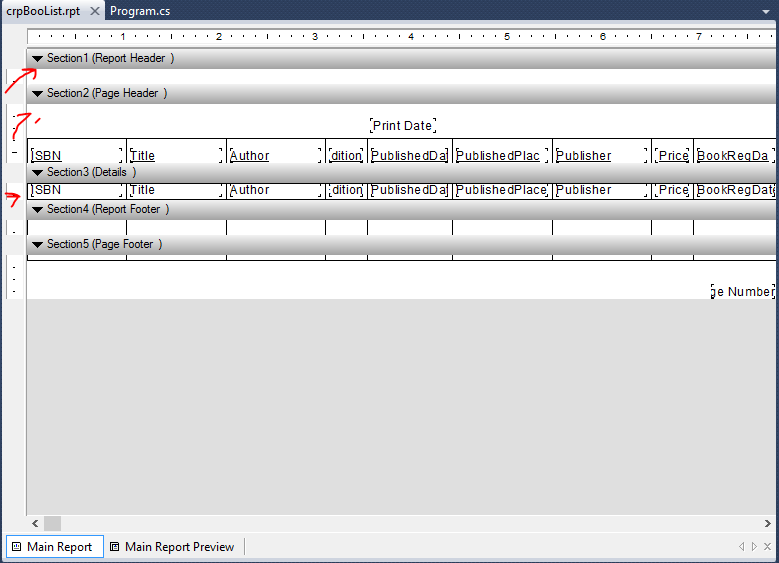


Fig 5.18: Crystal report designer window

1. Add a Form to your project and drag a CrystalReportViewer control to your form
2. Select Form's source code view and put the code on top

Using CrystalDecisions.CrystalReports.Engine;

Put the following source code in the button click event

ReportDocument cryRpt = new ReportDocument();

cryRpt.Load(@"D:\Projects\CSharp\LibrarySystem\LibrarySystem\crpBooList.rpt");

crystalReportViewer1.ReportSource = cryRpt;

crystalReportViewer1.Refresh();

1. Run your project, look at your report.

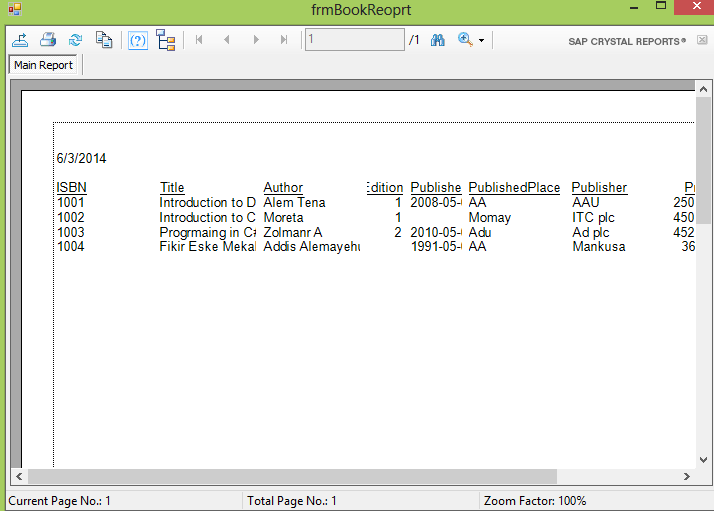


Fig 5.19: Crystal report

**Lab Exercise**

**Working on Crystal report**

1. Create a crystal report that displays data from Book and Item table grouped by Book title.
2. Create crystal report that accept parameters, such as Integer, date, string, from the user and generate the report based on the parameters.
3. Create a complex report that generate data more than two tables and accept parameters from the user.

## Deploying your event driven Applications

At point during the process of developing a Windows application, you need to deploy the application so you can test it on the target system and ultimately so your users can run it.

Way to deploy Window application:

* XCopy
* ClickOne
* Setup program

**XCOPY**

The oldest and easiest way to deploy a windows application is to copy files that are required by the application to the user’s computer.

**How to use ClickOnce works**

1. Build your project
2. Use Project Properties and click on publish tab

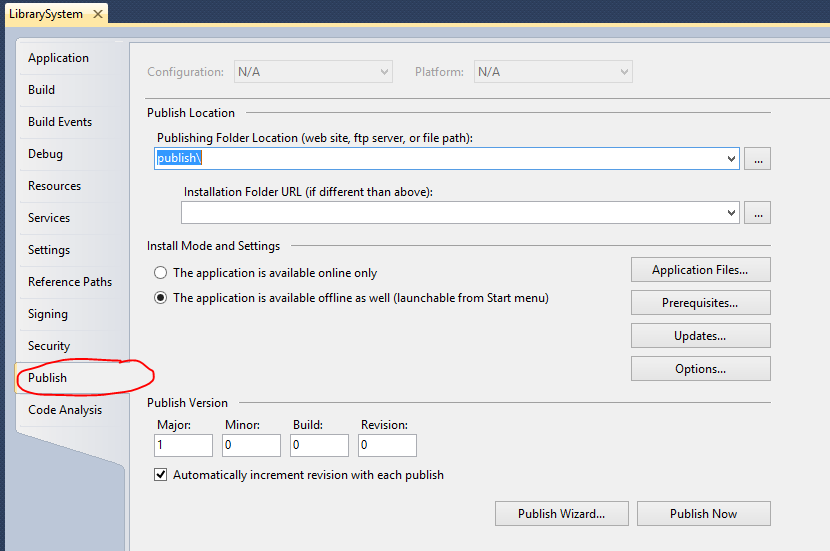


Fig 5.20: project properties window

1. Set Installation Mode and setting
2. Select application file required by the application

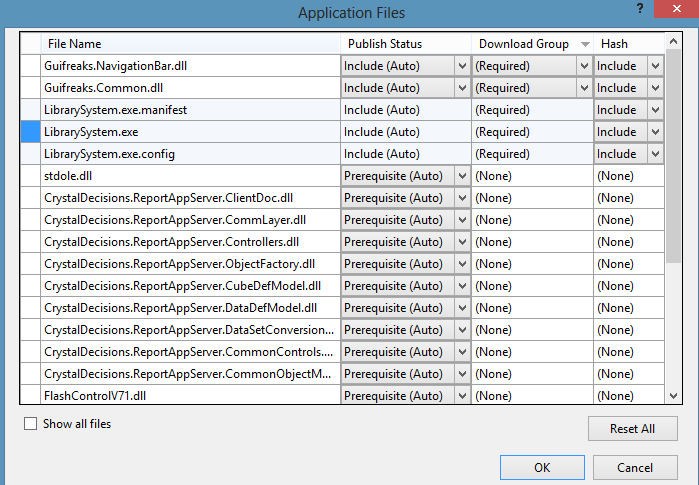


Fig 5.21: Selecting Application files

1. Select prerequisites file

Choose option from where you download prerequisite files

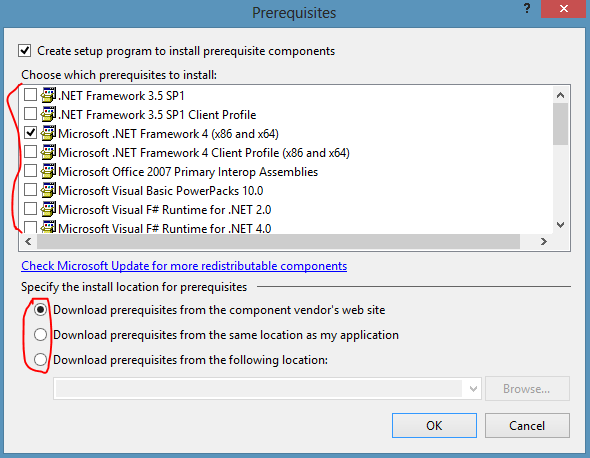


Fig 5.23: Selecting prerequisite files

1. Set application update option
2. Set version, option like publisher, product name,…
3. Click Purplish
4. Go to your DataDirectory\publish folder and look at the setup files

**Using Setup program**

1. Add a new setup project to your solution, File->New Project or right Click on the solution and use Add->new project Select Othe project type->Setup and Deployment->Visula Studio Installer->Setup project

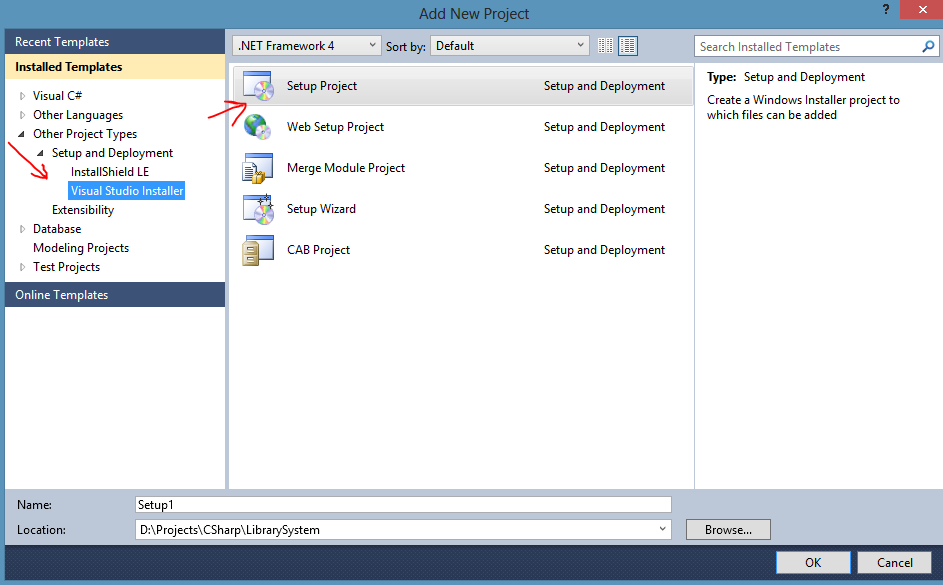


Fig 5.24: Adding Setup project

1. Select project output file, right Click on Application folder->Add->Project Output, click OK

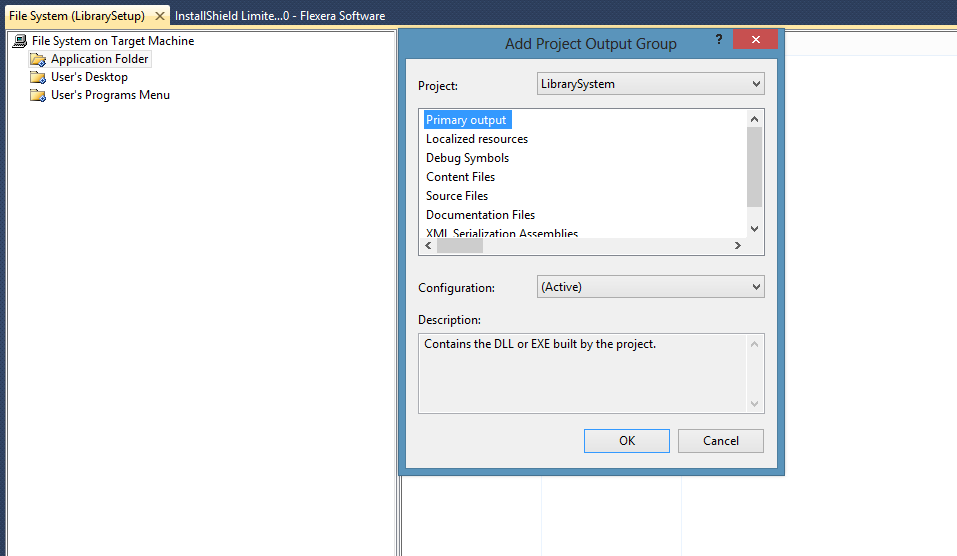


Fig 5.25: Adding project output

1. Add required file like database file Application folder->Add->File, browse to file
2. Create shortcut Users’ Desktop ->Create shortcut to users’ desktop from the primary output file
3. Create Shortcut to Users programs menu

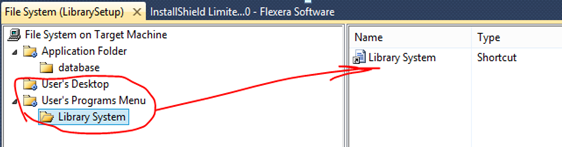


Fig 5.26: Adding shortcuts

1. Build your project
2. Go to your setup project folder and have a look at the setup files
3. Install your application on other computer and test it.

**Lab Exercise**

**Preparing a Setup file for your project**

1. Use ClickOnce way to prepare a setup file for your project
2. Use Visual Studio Installer setup project to deploy your application to user system
3. Download, install and use the InstallShield limited edition product for preparing a setup program to deploy your application. InstallShield is a setup program developed by Flexera software and provide more advanced option for prepare setup program for large windows application.