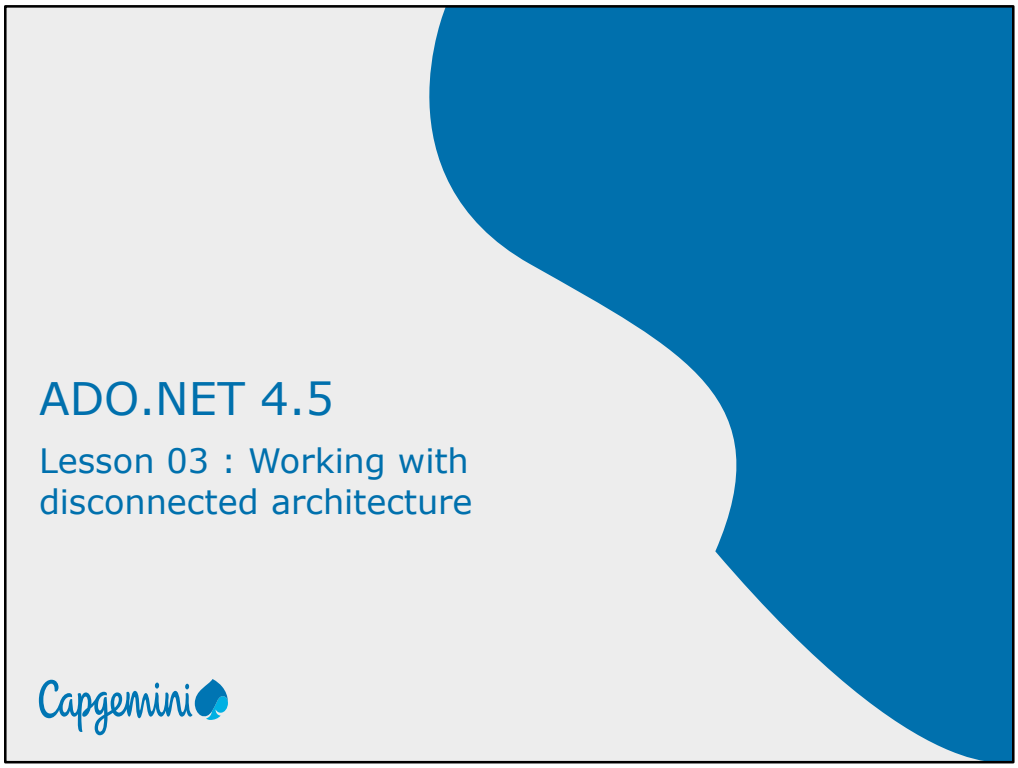

ADO.NET 4.5
Lesson 03 : Working with disconnected architecture
Capgemini

Lesson Objectives

- In this lesson, you will learn:
  - Creation and use of Data Set to retrieve data
  - Manipulation of database using Data Set
  - Implementing Transactions in ADO.NET

3.1: Creating and Using DataSet to retrieve Data

An Introduction

- A disconnected environment is one in which a user or an application is not directly connected to a server for a data
- Microsoft® ADO.NET provides extensive support for creating disconnected applications
- The data is modified independently & changes are merged back into the central data store
- It provides an application with various advantages like performance, scalability, availability

Introduction :-

This lesson describes how to retrieve data by using Microsoft ADO.NET and how to disconnect from the data source, work with that data, and then reconnect to the ADO.NET data source to save any data updates. The lesson also describes how to create a DataSet object programmatically, and explains how to populate and save data in a DataSet by using a DataAdapter. The lesson also describes how to create a customized view of the data in a DataSet by using a DataView object.

3.1: Creating and Using DataSet to retrieve Data

Sql Data Adapter

- The Sql Data Adapter class serves as a bridge between a disconnected ADO.NET objects and a data source
- The Data Adapter retrieves data into a Data Set or a Data Table from a data source using the Fill() method
- Along with data, schema information can be retrieved using the Fill Schema() method
- It updates any changes made to the Data Set back to the data source using the Update() method

SQLDataAdapter:
The SqlDataAdapter uses the Fill method , which changes the data in the DataSet to match the data in the data source, and the Update method, which changes the data in the data source to match the data in the DataSet, using the appropriate Transact-SQL statements against the data source.
In simple scenarios, the updating logic that the DataAdapter uses to reconcile changes made to the DataSet can be generated automatically from the query used to retrieve the data by using a CommandBuilder object. For more complex scenarios, custom update logic can be written to control the logic the DataAdapter uses while updating data. The update is performed on a row-by-row basis. For every inserted, modified, and deleted row, the Update method determines the type of change that has been performed on it whether it was an Insert, Update, or Delete operation. Depending on the type of change, the Insert, Update, or Delete command template executes to propagate the modified row to the data source. When the SqlDataAdapter fills a DataSet , it creates the necessary tables and columns for the returned data if they do not already exist. However, primary key information is not included in the implicitly created schema unless the MissingSchemaAction property is set to AddWithKey. You may also have the SqlDataAdapter create the schema of the DataSet , including primary key information, before filling it with data using FillSchema method. SqlDataAdapter is used in conjunction with SqlConnection and SqlCommand to increase performance when connecting to a SQL Server database. The SqlDataAdapter also includes the SelectCommand, InsertCommand, DeleteCommand, UpdateCommand, and TableMapping properties to facilitate the loading and updating of data.

SQLDataAdapter [Contd.]:

The **SelectCommand** gets or sets a **Transact-SQL** statement or stored procedure used to select records in the data source**.** When **SelectCommand** is assigned to an existing **SqlCommand**, the **SqlCommand** is not duplicated. The **SelectCommand** maintains a reference to the previously created **SqlCommand** object. If the **SelectCommand** does not return any rows, no tables are added to the DataSet, and no exception is raised.
Similarly the **InsertCommand**, **DeleteCommand** and **UpdateCommand** gets or sets a Transact-SQL statement or stored procedure to insert, delete and update records into the data source respectively. In the same way as the **SelectCommand** even the **InsertCommand**, **DeleteCommand** & **UpdateCommand** if assigned to an existing **SqlCommand**, the **SqlCommand** is not duplicated. It maintains a reference to the previously created **SqlCommand** object
When an instance of **SqlDataAdapter** is created, the read/write properties are set to some initial values. The **InsertCommand**, **DeleteCommand**, and **UpdateCommand** are generic templates that are automatically filled with individual values from every modified row through the parameters mechanism.
For every column that you propagate to the data source on Update, a parameter should be added to the **InsertCommand**, **UpdateCommand**, or **DeleteCommand**. The **SourceColumn** property of the **DbParameter** object should be set to the name of the column. This setting indicates that the value of the parameter is not set manually, but is taken from the particular column in the currently processed row.

The **ContinueUpdateOnError** property of dataadapter controls whether an Update() continues with remaining rows or stops processing if an error is encountered during the updating. If ContinueUpdateOnError is true, and an error is encountered during the update, an exception isn't raised, the RowError property of the DataRow causing the error is set to the error message that would have been raised, and the update continues processing the remaining rows. A well-designed application uses the RowError information to present the user with a list of the failed and possibly the current values in the data source for those rows. It also provides a mechanism to correct and resubmit the failed attempts, if required.

If ContinueUpdateOnError is false, the DataAdapter raises a DBConcurrencyException when a row update attempt fails. Generally, ContinueUpdateOnError is set to false when the changes made to the DataSet are part of a transaction and must be either completely applied to the data source or not applied at all. The exception handler rolls back the transaction.

3.1: Creating and Using DataSet to retrieve Data

Optimizing Connection

- Using Data Adapter to optimize connections:
  - Update and Fill method of Data Adapter automatically opens and closes the connection
  - Allow the Update and Fill method to open and close connection implicitly if it is a one time activity
  - Explicitly open the connection if the Fill and Update is going to be called several times

Managing Connections:

The DataAdapter's Fill and Update method automatically opens the connection, executes, and closes the connection. However, it is recommended that if it is a one time activity, then allow the Update and Fill method to implicitly open and close connection. Else you open the connection explicitly, that is if the methods are to be called several times. Close the connection once the required Fill and Update methods have executed.
Also while performing transaction, open the connection and once the transaction is completed, commit your work and then close the connection.

3.1: Creating and Using DataSet to retrieve Data

Data Set

- The ADO.NET Data Set is an in memory representation of data
- A Data Set provides a consistent relational programming model regardless of the source of the data it contains
- Data Set is disconnected from data source
- Example:

```
DataSet ds = new DataSet("Customers");
adapter.Fill(ds);
```

DataSet:

The DataSet is a memory-resident representation of data that provides consistent relational programming model regardless of the source of the data it contains. A DataSet represents a complete set of data including the tables that contain columns, rows and constrain the data, as well as the relationships between the tables. It is disconnected from the data source.

The principal class used by the ADO.NET disconnected model is System.Data.DataSet. A DataSet object provides an in-memory cache of data. The data in a DataSet is held in a format that is independent of any underlying data source.

There are several methods of working with a DataSet, which can be applied independently or in combination. You can:

Programmatically create DataTables, DataRelations, and Constraints within the DataSet and populate the tables with data.
Populate the DataSet with tables of data from an existing relational database management system using a DataAdapter.
Load and persist the DataSet contents using XML.

DataSet [Contd.]:

In a typical multi-tier implementation, the steps for creating and refreshing a **DataSet**, and in turn, updating the original data are to:

- Build and fill each **DataTable** in a **DataSet** with data from a data source using a **DataAdapter**.
- Change the data in individual **DataTable** objects by adding, updating, or deleting **DataRow** objects.
- Invoke the **GetChanges** method to create a second **DataSet** that features only the changes to the data.

Call the **Update** method of the **DataAdapter**, passing the second **DataSet** as an argument.

- Invoke the **Merge** method to merge the changes from the second DataSet into the first.
- Invoke the **AcceptChanges** on the DataSet. Alternatively, invoke **RejectChanges** to cancel the changes

The design of the ADO.NET DataSet makes this scenario easy to implement. Because the DataSet is stateless, it can be safely passed between the server and the client without tying up server resources such as database connections. Although the DataSet is transmitted as XML, Web Services and ADO.NET automatically transform the XML representation of the data to and from a DataSet, creating a rich, yet simplified, programming model. Additionally, because the DataSet is transmitted as an XML stream, non-ADO.NET clients can consume the same Web Service as that consumed by ADO.NET clients. Similarly, ADO.NET clients can interact easily with non-ADO.NET Web Services by sending any client DataSet to a Web Service as XML data and by consuming any XML returned as a DataSet from the Web Service.

The DataSet consists of a collection of DataTable objects that you can relate to each other with DataRelation objects. You can also enforce data integrity in the DataSet by using the **UniqueConstraint** and **ForeignKeyConstraint** objects.

Whereas DataTable objects contain the data, the **DataRelationCollection** allows you to navigate though the table hierarchy. The tables are contained in a **DataTableCollection** accessed through the Tables property. When accessing **DataTable** objects, note that they are conditionally case sensitive.

A DataSet can read and write data and schema as XML documents. The data and schema can then be transported across HTTP and used by any application, on any platform that is XML-enabled. You can save the schema as an XML schema with the **WriteXmlSchema** method, and both schema and data can be saved using the WriteXml method. To read an XML document that includes both schema and data, use the **ReadXml** method.

(Note: We will be elaborating on Data Table and Data Relation later in this lesson)

3.1: Creating and Using Data Set to retrieve Data

Data Set

- These are some of the members exposed by DataSet:

DataSetName	Clear()
EnforceConstraints	GetChanges()
HasErrors	HasChanges()
Relations	Merge()
Tables	RejectChanges()
AcceptChanges()	Reset()

DataSet Members:
These are some members of DataSet:

DataSetName - Gets or sets the name of the current DataSet
EnforceConstraints – Gets or sets a value indicating whether constraint rules are followed when attempting any update operation
HasErrors – Indicates whether there are errors in any of the rows in any of the tables of this DataSet
Relations – Retrieves the collection of relations that link tables and allow navigation from parent tables to child tables
Tables - Retrieves the collection of tables contained in the DataSet.
AcceptChanges() – Commits all the changes made to this DataSet since it was loaded or the last time AcceptChanges was called
Clear()- clears the DataSet of any data by removing all rows in all tables.
GetChanges()- Retrieves a copy of the DataSet containing all changes made to it since it was last loaded, or since AcceptChanges was called.
HasChanges()- Retrieves a value indicating whether the DataSet has changes, including new, deleted, or modified rows
RejectChanges()- Rolls back all the changes made to the DataSet since it was created, or since the last time DataSet.AcceptChanges was called
Merge() – Merges this DataSet with a specified DataSet.
Reset()- Resets the DataSet to its original state. Subclasses should override Reset to restore a DataSet to its original state.

3.1: Creating and Using DataSet to retrieve Data

Demo

- Using DataSet and DataAdapter

3.1: Creating and Using Data Set to retrieve Data

What is Typed Data Set?

- Typed DataSets are a collection of classes that inherit from the DataSet,DataTable, and DataRow classes
- They provide additional properties, methods, and events based on the DataSet schema
- These methods, properties, and events allow developer to retrieve column values, access parent and child records
- It also facilitates us to find rows, and handle null column values

What is Typed DataSet?

A typed dataset is a dataset that is first derived from the base DataSet class and then uses information from the Dataset Designer, which is stored in an .xsd file, to generate a new strongly-typed dataset class. Information from the schema (tables, columns, and so on) is generated and compiled into this new dataset class as a set of first-class objects and properties. Because a typed dataset inherits from the base DataSet class, the typed class assumes all of the functionality of the DataSet class and can be used with methods that take an instance of a DataSet class as a parameter.

There are some drawbacks to using strongly typed DataSet objects:

Using typed classes adds some overhead to code execution. If the strongly typed functionality isn't required, application performance can be improved slightly using an untyped DataSet.

The strongly typed DataSet will need to be regenerated when the data structure changes. Applications using the typed DataSet will need to be rebuilt using a reference to the new typed DataSet. This can be especially significant in a multitier application or distributed where any clients that use the typed DataSets will have to be rebuilt using a reference to the updated version, even those that would not otherwise have been affected by the change if an untyped DataSet was used.

3.1: Creating and Using Data Set to retrieve Data

What is Untyped Data Set?

- An untyped dataset, has no corresponding built-in schema
- As in a typed dataset, an untyped dataset contains tables, columns, and so on
- But those are exposed only as collections
- After manually creating the tables and other data elements in an untyped dataset, you can export the dataset's structure
- You can use the dataset's WriteXmlSchema method to get the dataset structure as an xml schema

Difference between Typed DataSet and Untyped DataSet

Typed DataSet
1. Typed DataSets are DataSets with XML Schema Definition (xsd).
2. They perform error checking regarding their schema at design time using the .xsd definitions.
3. Typed DataSets have intelliSense support..
4. Performance is slower in case of strongly typed dataset.
5. In complex environment, strongly typed dataset's are difficult to administer.

Untyped DataSet

1. Untyped DataSets are DataSets without XML Schema Definition (xsd).
2. Erros cannot be trapped at the design time as they are filled at run time when the code executes.
3. Do not support intelliSense.
4. Performance is faster in case of Untyped dataset.
5 .Untyped datasets are easy to administer.

3.1: Creating and Using Data Set to retrieve Data

When do you use Data Set?

- Data Set should be used in the application when:
  - Navigating between multiple discrete results
  - Performing data manipulation from multiple sources
  - Reusing the same set of results to perform sorting, searching, and so on.
    - Caching the results improves performance
  - Using Typed Data Set to include type checking at design time

Managing DataSets and Other Objects:

When do you use DataSet?

DataSet can be used in the application whenever discrete table of results is being navigated and also if the application manipulates data from different sources like a relational database or XML file.
Additionally, caching the data using the DataSet improves performance for tasks such as Sorting, Filtering.
To include type checking at design time use Typed DataSet. Apart from this in .Net environment statement completion is also supported when you work with Typed DataSet.

Some more noteworthy points while working with DataSet:

To Refresh Data in DataSet: To get updated values from the server use the DataAdapter.Fill method. This method matches new rows based on primary keys and applies corresponding changes from the server, if a Primary Key is defined for the DataTable. If the Primary Key is not defined, then the Fill method adds new rows for the refreshed values, which might lead to addition of duplicate rows. If you want to retain the changes made to the rows in the table and also refresh the data from the server then you must first populate it with DataAdapter.Fill method, and fill a new DataTable. Then merge that DataTable into the DataSet by setting preserveChanges value to true.
To Search Data in a DataSet: While querying DataSet based on a criteria, the performance can be improved by taking advantage of index-based lookups. An index is created, whenever a Primary Key is assigned to the DataTable, or a DataView is added for a DataTable.

Managing DataSets and Other Objects:

When do you use DataSet (contd.)?

- If the query is fired on the Primary Key columns of the DataTable, then use **DataTable.Rows.Find**, rather than using **DataTable.Select**.
- Specify a sort order for the DataView, so that an index is available for queries involving Non-primary Key columns.
- However, remember to use the **index** feature only when you are performing multiple queries. Creating the index is costly and overall benefit of using the index is reduced.

3.1: Creating and Using DataSet to retrieve Data

Data Table

- A Data Table represents one table of in-memory relational data
  Example :

```
DataTable ordersTable = new DataTable("Orders");
ordersTable.Columns.Add("OrderID");
ordersTable.Columns.Add("OrderQuantity");
ordersTable.Columns.Add("CustID");
DataColumn dc= new DataColumn
                (ordersTable.Columns("OrderID");
ordersTable.PrimaryKey = dc);
```

DataTable:

A DataTable, which represents one table of in-memory relational data, can be created and used independently, or can be used by other .NET Framework objects, most commonly as a member of a DataSet.
You can create a DataTable object by using the DataTable constructor, or by passing constructor arguments to the Add method of the Tables property of the DataSet, which is a DataTableCollection.
When accessing DataTable objects, note that they are conditionally case sensitive. For example, if one DataTable is named "mydatatable" and another is named "Mydatatable", a string used to search for one of the tables is regarded as case sensitive. However, if "mydatatable" exists and "Mydatatable" does not, the search string is regarded as case insensitive. A DataSet can contain two DataTable objects that have the same TableName property value but different Namespace property values.
You can also create DataTable objects within a DataSet by using the Fill or FillSchema methods of the DataAdapter object. If you are creating a DataTable programmatically, you must first define its schema by adding DataColumn objects to the DataColumnCollection which can be accessed through the Columns property. To add rows to a DataTable, you must first use the NewRow method to return a new DataRow object. The NewRow method returns a row with the schema of the DataTable, as it is defined by the table's DataColumnCollection. The maximum number of rows that a DataTable can store is 16,777,216. The DataTable also contains a collection of Constraint objects that can be used to ensure the integrity of the data.

3.1: Creating and Using Data Set to retrieve Data

Data Table Members

- These are some of the members exposed by Data Table:

ChildRelations	Columns
Constraints	DataSet
ParentRelations	PrimaryKey
Rows	TableName
NewRow()	Select()

DataTable Members & Events:

DataTable dTable = new DataTable("Customers“)

The following example creates an instance of a DataTable by adding it to the Tables collection of a DataSet.

```
DataSet custDS = new DataSet();
DataTable custTable = custDS.Tables.Add("CustTable")
```

Some of the important members of DataTable object.

ChildRelations - Gets the collection of child relations for this DataTable.
Columns- Gets the collection of columns that belong to this table.
Constraints - Gets the collection of constraints maintained by this table.
DataSet - Gets the DataSet that this table belongs to.
ParentRelations - Gets the collection of parent relations for this DataTable.
PrimaryKey - Gets or sets an array of columns that function as primary keys for the data table.
Rows - Gets the collection of rows that belong to this table.
TableName - Gets or sets the name of the DataTable.
NewRow() - This method creates a new DataRow with the same schema as the table.
Select()- This method Gets an array of DataRow objects.

The DataTable also exposes some events as:

ColumnChanged - Occurs when after a value has been changed for the specified DataColumn in a DataRow.
ColumnChanging - Occurs when a value is being changed for the specified DataColumn in a DataRow.
RowChanged - Occurs after a DataRow has been changed successfully.
RowChanging - Occurs when a DataRow is changing.
RowDeleted - Occurs after a row in the table has been deleted.
RowDeleting - Occurs before a row in the table is about to be deleted

3.1: Creating and Using Data Set to retrieve Data

Data Column

- The Data Column defines the schema for a single column in a Data Table
- The Data Table schema is defined by a collection of columns in the table along with any constraints
- Example :

```
// create the column and set the name and data type using
properties
 DataColumn col = new DataColumn();
col.ColumnName = "MyTextColumn";
col.DataType = typeof(System.String);
```

DataColumn

The DataColumn defines the schema for a single column in a DataTable. The DataTable schema is defined by a collection of columns in the table along with any constraints. The DataColumn defines:

The type of data that can be stored in the column
The length of the column for text-based column types
Whether the data in the column can be modified
Whether the column values for each row must be unique
Whether the column in rows can contain null values
Whether the column values are automatically generated and the rules for generating those values
Whether the column value is calculated based on an expression

3.1: Creating and Using Data Set to retrieve Data

Data Row

- The Data Row class represents a single row of data in the Data Table
- The Data Row class can retrieve, update, insert, and delete a row of data from the Data Table
- Using the Data Row class, each column value for the row can be accessed

Example :

```
DataRow row = dt.NewRow();
row[ "MyColumn"] = "Item 1";
/ /add the row to the table
Rows. Add(row);
```

DataRow

The DataRow class represents a single row of data in the DataTable. The DataRow class can retrieve, update, insert, and delete a row of data from the DataTable. Using the DataRow class, each column value for the row can be accessed. The DataRow maintains the RowState property that is used by ADO.NET to track the changes that have been made to a DataRow. This property allows changed rows to be identified, and the appropriate update command to be used to update the data source with the changes.

Creating a DataRow

A DataRow is created by calling the NewRow() method of a DataTable, a method that takes no arguments. The DataTable supplies the schema, and the new DataRow is created with default or empty values for the fields:

```
// create a table with one column
DataTable dt = new DataTable();
dt.Columns.Add("MyColumn", typeof(System. String));

//create a row with the same schema as DataTable
dtDataRow row = dt.NewRow();
row[ "MyColumn"] = "Item 1";
// add the row to the table
dt. Rows.Add(row);
```

3.1: Creating and Using Data Set to retrieve Data

Row State

- The Row State property is used by ADO.NET to track the changes that have been made to a Data Row
- The Row State property indicates whether the row belongs to a table
- It also gives you an insight whether it's a row which is newly inserted, modified, deleted, or unchanged
- The value of the Row State depends on two factors:
  - The kind of operation has been performed on the row
  - Whether Accept Changes has been called on the Data Row

RowState

The RowState property is used by ADO.NET to track the changes that have been made to a DataRow, which allows changes made to the data while disconnected to be updated back to the data source. The RowState property indicates whether the row belongs to a table, and if it does, whether it's newly inserted, modified, deleted, or unchanged since it was loaded.
The value of the RowState property can't be set directly. ADO.NET sets the row state in response to actions that affect the DataRow. The AcceptChanges() and RejectChanges() methods, whether explicitly or implicitly called, both reset the RowState value for the row to Unchanged.
ADO.NET maintains up to three versions of each DataRow object. The DataAdapter reconciles changes made since the data was loaded from the data source, thereby making changes to the disconnected data permanent. Two versions of each row are maintained to allow the DataAdapter to determine how to perform the reconciliation. The Original version contains the values that were loaded into the row. The Current version contains the latest version of the data, including the changes made since the data was originally loaded. The Original version isn't available for newly created rows. ADO.NET also allows a row to be put into edit mode which temporarily suspends events for the row and allows the user to make multiple changes to the row without triggering validation rules. The BeginEdit( ) method of the DataRow puts the row into edit mode, while the EndEdit( ) and CancelEdit( ) methods take the row out of edit mode. AcceptChanges( ) also takes the row out of edit mode because it implicitly calls EndEdit( ), as does RejectChanges( ), which implicitly calls CancelEdit( ).

RowState

The **RowState** does detect the whether a row has been changed or no. But sometimes you may need to access the information in a deleted row in a **DataTable**. If you access the deleted row directly, you will get an exception saying that the row has been deleted.

To avoid the exception, but still access the information in the deleted row of the DataTable, you just need to specify that you want to get the information from the original version of the row by specifying **DataRowVersion.Original** as follows:

```
if (dataRow.RowState == DataRowState.Deleted)
    id = (string)dataRow["CustomerID", DataRowVersion.Original];
```

You can get only the deleted rows from a DataTable by iterating through all the rows and checking the DataRowState, or you could simply create a DataView that filters on DataViewRowState.Deleted:

```
// Get Only Deleted Rows in DataTable
DataView dv = new DataView(sourceDataTable,
                           null, null, DataViewRowState.Deleted);
```

You can also convert that DataView to a DataTable using the new **DataView.ToTable()** method:

```
DataTable dt = dv.ToTable();
```

The rows in the new DataTable, dt, will not be marked as **DataRowState.Deleted** but rather **DataRowState.Added**, because this is a new DataTable. You can easily iterate through this table now as none of the rows will be mark deleted and you will have full access to the information without specifying **DataRowVersion**.

3.1: Creating and Using Data Set to retrieve Data

Demo

- Understanding Row State

3.1: Creating and Using Data Set to retrieve Data

Specifying Constraints

- Constraints can be specified to enforce restrictions on the data in a Data Table
- It helps in maintaining the integrity of the data which is stored in the Data Table
- It is an automatic rule, applied to a column or related columns, that determines the action when the value of a row is altered
- There are two kinds of constraints in ADO.NET :
  - Foreign Key Constraint and the Unique Constraint

Specifying Constraints

You can use constraints to enforce restrictions on the data in a DataTable, in order to maintain the integrity of the data. A constraint is an automatic rule, applied to a column or related columns, that determines the course of action when the value of a row is somehow altered. Constraints are enforced when the EnforceConstraints property of the DataSet is TRUE.
There are two kinds of constraints in ADO.NET: the ForeignKeyConstraint and the UniqueConstraint. By default, both constraints are created automatically when you create a relationship between two or more tables by adding a DataRelation to the DataSet. However, you can disable this behavior by specifying createConstraints = FALSE when creating the relation.
A ForeignKeyConstraint enforces rules about how updates and deletes to related tables are propagated
The DeleteRule and UpdateRule properties of the ForeignKeyConstraint define the action to be taken when the user attempts to delete or update a row in a related table. The settings for DeleteRule and UpdateRule properties for the ForeignKeyConstraint are as follows:-

- **Cascade** - Deletes or updates related rows.
- **SetNull** - Set values in related rows to DBNull.
- **SetDefault** - Set values in related rows to the default value.
- **None** -Take no action on related rows. This is the default behavior.

3.1: Creating and Using Data Set to retrieve Data

Specifying Constraints

- Code Snippet

```
ForeignKeyConstraint custOrderFK = new
      ForeignKeyConstraint("CustOrderFK",
 custDS.Tables["CustTable"].Columns["CustomerID"],
           custDS.Tables["OrdersTable"].Columns["CustomerID"]);
// Cannot delete a customer value that has associated
//existing orders.
custOrderFK.DeleteRule = Rule.None;
custDS.Tables["OrdersTable"].Constraints.Add(custOrderFK);
```

Specifying Constraints [Contd]

The UniqueConstraint object, which can be assigned either to a single column or to an array of columns in a DataTable, ensures that all data in the specified column or columns is unique per row. You can create a unique constraint for a column or array of columns by using the constructor for UniqueConstraint. Pass the resulting UniqueConstraint object to the Add method of the table's Constraints property, which is a ConstraintCollection. When creating a UniqueConstraint for a column or columns, you can optionally specify whether the column or columns are a primary key.

You can also create a unique constraint for a column by setting the Unique property of the column to TRUE. Alternatively, setting the Unique property of a single column to false removes any unique constraint that may exist. Defining a column or columns as the primary key for a table will automatically create a unique constraint for the specified column or columns. If you remove a column from the PrimaryKey property of a DataTable, the UniqueConstraint is removed.

3.1: Creating and Using Data Set to retrieve Data

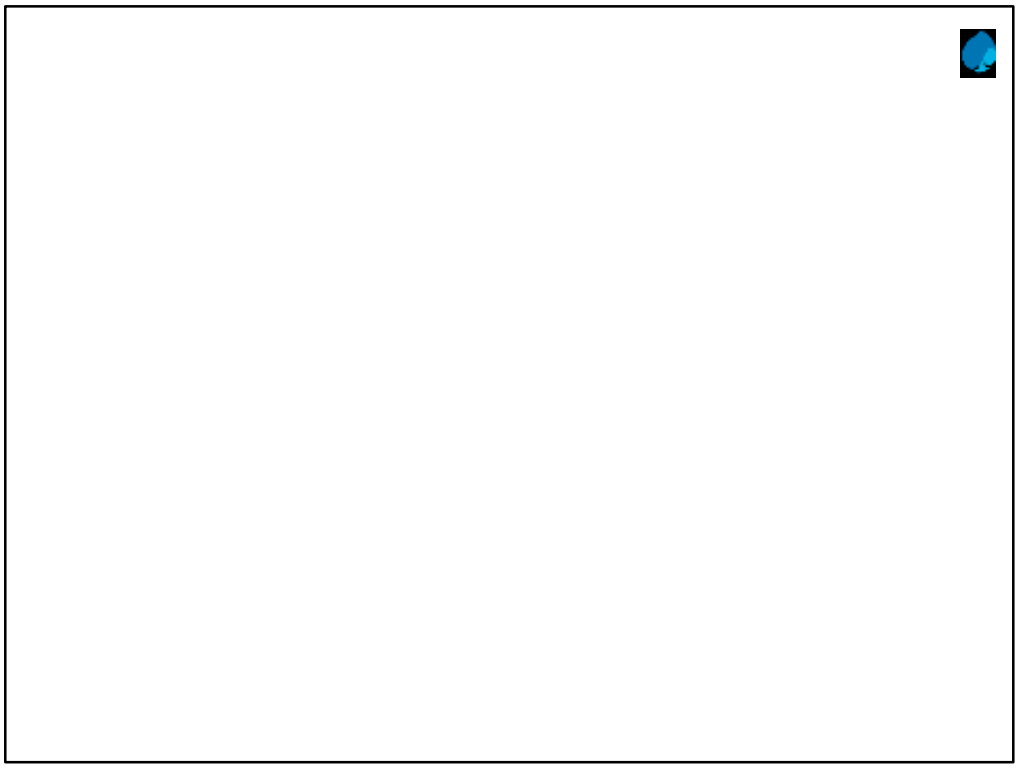
Data Relation

- A Data Relation object represents relationship between two Data Table objects in a Data Set
- A Data Set is like a database which might contain various interrelated tables
- A Data Relation object lets you specify relations between various tables across tables
- Its most logical equivalent is a foreign key specified between two tables in a database

DataRelation

One of the biggest differences between traditional ADO and ADO.NET is that the rowsets stored within ADO.NET can be truly relational. For example, a DataSet can store one DataTable containing customers and another DataTable containing the customers' orders. These DataTable objects can then be related to one another within ADO.NET, thus recreating the relationship that exists within the relational database. In ADO.NET once you retrieve two rowsets of data (in other words, parents and children) and relate them to each other, you can then retrieve all children rows for a given parent, display any one DataTable in a grid at a time, or modify several tiers of DataTable objects, and send the changes to the database all in one batch update. DataRelation objects, which are integral to ADO.NET applications, enable these features to function.

In a DataSet that contains multiple DataTable objects, you can use DataRelation objects to relate one table to another, to navigate through the tables, and to return related values. For example, in a Customer/Orders relationship, the Customers table is the parent and the Orders table is the child of the relationship. This is similar to a primary key/foreign key relationship. Relationships are created between matching columns in the parent and child tables. That is, the DataType value for both columns must be identical.
The arguments required to create a DataRelation are the columns that serve as the parent and child columns in the relationship, and a name for the DataRelation being created. Once a DataRelation has been created, it can be used for navigating between two tables or when retrieving values.



When a DataRelation is created, it first verifies that the relationship can be established. After it is added to the DataRelationCollection, the relationship is maintained by disallowing any changes that would invalidate it. Between the period when a DataRelation is created and added to the DataRelationCollection, it is possible for additional changes to be made to the parent or the child rows. An exception is generated if this causes a relationship to become invalid. DataRelation objects are contained in a DataRelationCollection, which you can access through the Relations property of the DataSet, and the ChildRelations and ParentRelations properties of the DataTable.

3.1: Creating and Using Data Set to retrieve Data

Data Relation

- There is a slight difference between a Foreign Key Constraint and a Data Relation
- Data Relation, along with validating data gives you an easy mechanism to browse parent and child rows in a Data Set

Example :

```
DataRelation custOrderRel =
custDS.Relations.Add("CustOrders",
custDS.Tables("Customers").Columns("CustomerID"),
custDS.Tables("Orders").Columns("CustomerID"))
```

There are two reasons why you might define DataRelation objects:

To provide better error checking (for example, spotting an orphaned child before you reconnect to update the data source). This functionality is providedthrough the ForeignKeyConstraint, which the DataRelation can create implicitly.
To provide better navigation

Once a relation is established, you can use it navigate from a parent row to the associated child rows or from a child row to the parent. Use the GetChildRows() or GetParentRow() method of the DataRow object:

Example

```
foreach(DataRow parent in ds.Tables["Categories"].Rows)
 {
  //Process the category row
  foreach(DataRow child in parent.GetChildRows("Cat_Prod"))
 {
  //Process the products in this category
   }
   }
```

3.1: Creating and Using Data Set to retrieve Data

Demo

- Using the Data Relation object

3.1: Creating and Using Data Set to retrieve Data

What are Data Views?

- The Data View enables you to create various views of the data stored in a Data Table
- You can use a Data View to obtain a sorted or filtered view of data in a Data Table
- It is also possible to add, modify, and delete rows in a Data Table object through a Data View

Example :

```
DataView custDV = new DataView(custDS.Tables["Customers"],
"Country = 'USA'", "ContactName",
DataViewRowState.CurrentRows);
```

What are DataViews?

A DataView enables you to create different views of the data stored in a DataTable, a capability that is often used in data-binding applications.

Using a DataView, you can expose the data in a table with different sort orders, and you can filter the data by row state or based on a filter expression

A DataView provides you with a dynamic view of a single set of data, much like a database view, to which you can apply different sorting and filtering criteria.
In some respects, a DataView is similar to a view in a database. However, a DataView is subject to several restrictions that do not apply for database views, including:

You cannot treat a DataView as a table.
You cannot use a DataView to provide a view of joined DataTable objects.
A DataView cannot exclude columns that exist in the source DataTable.

The following code example demonstrates how to create a DataView using the DataView constructor. A RowFilter, Sort column, and DataViewRowState are supplied along with the DataTable.

DataView custDV = new DataView(custDS.Tables["Customers"], "Country = 'USA'", "ContactName", DataViewRowState.CurrentRows);

3.1: Creating and Using Data Set to retrieve Data

Demo

- Using the Data View object

3.2: Manipulating Database using Data Set

SQL Command Builder

- The SQL Command Builder class is a ADO.NET feature through which the Data Set changes are reflected in the database
- The Sql Data Adapter does not automatically generate the SQL statements required to reconcile changes made to the Data Set
- To generate INSERT, UPDATE, or DELETE statements, the Sql Command Builder uses the Select Command property
- This class is limited to single-table updates using SQL statements.

SQLCommandBuilder

The SQLCommandBuilder class is a ADO.NET feature through which the DataSet changes are reflected in the database. When an instance of SQLCommandBuilder class is created it automatically generates Transact-SQL statements for the single table updates that occur. A CommandBuilder is managed provider-specific class that works on top of the data adapter object. The SQLCommandBuilder runs the SelectCommand to collect the required information about the tables and columns involved and then creates InsertCommand, DeleteCommand and UpdateCommand. The command builder class ensures that the specified data adapter can be successfully used to update the given data source. It uses some of properties defined for the SelectCommand like connection, commandtimeout and transaction. Whenever any of these properties are modified, you need to call the CommandBuilder's RefreshSchema method to change the structure of the generated commands for further updates.
The command builder is useful because it lets you update the data source withchanges made to the DataSet using very little code. It also lets you create update logic without understanding how to code the actual delete, insert, and update SQL statements. There are drawbacks, however, including slower performance because of thetime that it takes to request metadata and construct the updating logic, updates thatare limited to simple single-table scenarios, and a lack of support for stored procedures.

3.2: Manipulating Database using Data Set

Demo

- Using SQL Command Builder

3.2: Manipulating Database using Data Set

Using Command Builder

- Set the Command property of the Data Adapter so that Command Builder can generate commands
- Limit the use of Command Builder to design time and ad-hoc scenarios
- Ensure to call the Refresh Schema, if the Select Command of the Data Adapter has changed

Managing DataSet and Other Objects:

Using CommandBuilder:

As we already know, the CommandBuilder automatically generates the following properties of a DataAdapter based on the SelectCommand property of the DataAdpater:

InsertCommand
UpdateCommand
DeleteCommand

To use the CommandBuilder effectively:

Set the CommandProperty to null for the DataAdapter, only then CommandBuilder generates the commands. If you have explicitly set a CommandProperty then CommandBuilder does not override it.
Limit the use of CommandBuilder to design time or ad-hoc scenarios. The processing required to generate the DataAdapter command property actually hampers the performance. If the contents of Insert / Update / Delete are known then set them explicitly. Alternatively you can design stored procedures for the commands and configure the DataAdapter to use them.
CommandBuilder uses the SelectCommand property of DataAdapter to determine the values for other command properties. Call the RefreshSchema, if the SelectCommand of the DataAdapter had changed.

3.3: Managing Data Integrity and Concurrency

Maintaining Database Integrity - Transaction

- A transaction is a unit of work
- You use transactions to ensure the consistency and integrity of a database
- If a transaction is successful, all of the data modifications performed during the transaction are committed and made permanent
- If an error occurs during a transaction, you can roll back the transaction to undo the data modifications that occurred during the transaction

3.3: Managing Data Integrity and Concurrency

Types of Transaction

- The .NET Framework provides support for local and distributed transactions
  - Local transactions. A local transaction applies to a single data source, such as a database. It is common for these data sources to provide local transaction capabilities. Local transactions are controlled by the data source, and are efficient and easy to manage
  - Distributed transactions. A distributed transaction spans multiple data sources. Distributed transactions enable you to incorporate several distinct operations, which occur on different systems

Example of Distributed Transaction: Say you ordered something from flipcart.com and made the payment from your bank account. Your bank account server and flipcart server are two different computers in two different networks and data is updated on both the servers (Amount is being deducted from your bank account and added to the bank account of flipcart).

Example of Local Transaction: You book a Fixed Deposit using NetBanking.

3.3: Managing Data Integrity and Concurrency

Manual and Automatic Transactions

- Transactions allow a system to maintain integrity
- .NET supports both manual and automatic transactions
  - Manual Transactions – Use transactional capabilities of the data source
  - Automatic Transactions- Managed by Microsoft Distributed Transaction Coordinator (MSDTC)
- System. Transaction namespace provides transactional capabilities in .Net

Maintaining Database Integrity:

Transactions allow a system to maintain integrity when interacting with multiple data sources. Both manual and automatic transactions are supported by .Net. The System.Transaction namespace provides with transactional capabilities in .Net.

Manual Transactions: In this, a transaction object is associated with the data source and transactional capabilities of data source is used. One transaction can have multiple commands executed against the data source. Manual transactions can be controlled by using the SQL Commands and they are limited to carry out transactions against a single data source. Manual transactions are faster as compared to automatic transactions since they do not need interprocess communication with MSDTC.

Automatic Transactions: As against manual transactions, automatic transactions can span multiple data sources and they are comparatively slower than manual transactions.
(Note: Distributed Transactions are not in the scope of this course)
Whenever multiple users attempt to modify unlocked data concurrency problems occur. Locking data ensures database consistency by controlling how changes made to data within an uncommitted transaction can be used by concurrent transactions.

3.3: Managing Data Integrity and Concurrency

Manual Transactions

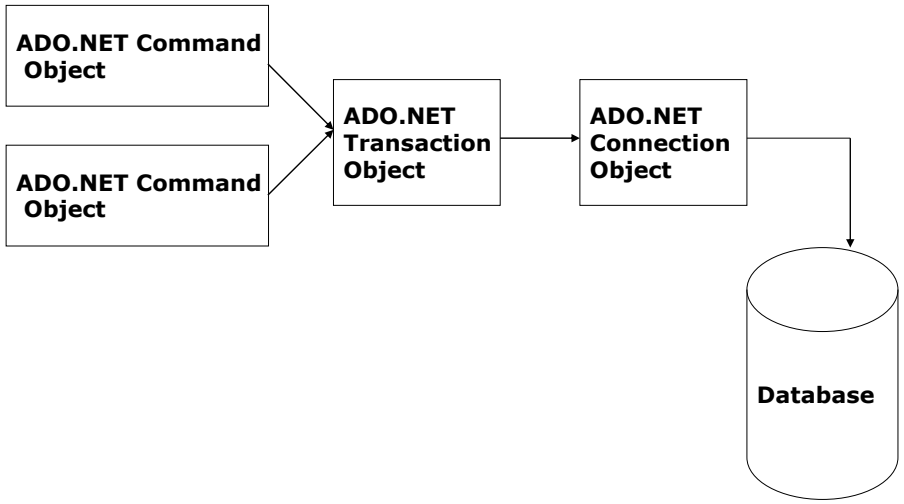
- Allows to explicitly control the transaction boundary
- Use the Begin Transaction() method on the connection object to start transaction
- Commit() or Rollback() has to be issued explicitly to complete the transaction
- Objects are provided by .Net providers to enable manual transactions

Manual Transactions:

Manual transactions allow control over the transaction boundary. You need to provide explicit instructions to start and end the transaction. Objects are provided by .Net providers to enable manual transactions. The Connection object gives a BeginTransaction() method that is used to start a transaction. The method returns a transaction object if successful, which can then be used to perform all subsequent actions associated with the transaction. The transaction object returned by the BeginTransaction() has to be associated with the Command object to execute the command within the transaction.
The transaction does not complete automatically, the Commit() or Rollback() methods have to be issued to complete the transaction. The Commit() method of the Transaction is used to commit the changes made to database through the transaction and Rollback() method of the transaction is used to undo the changes made to the database. In case the rollback is called after the commit then a InvalidOperationException() is raised.

3.3: Managing Data Integrity and Concurrency

ADO.NET Transaction Support – Connected Architecture



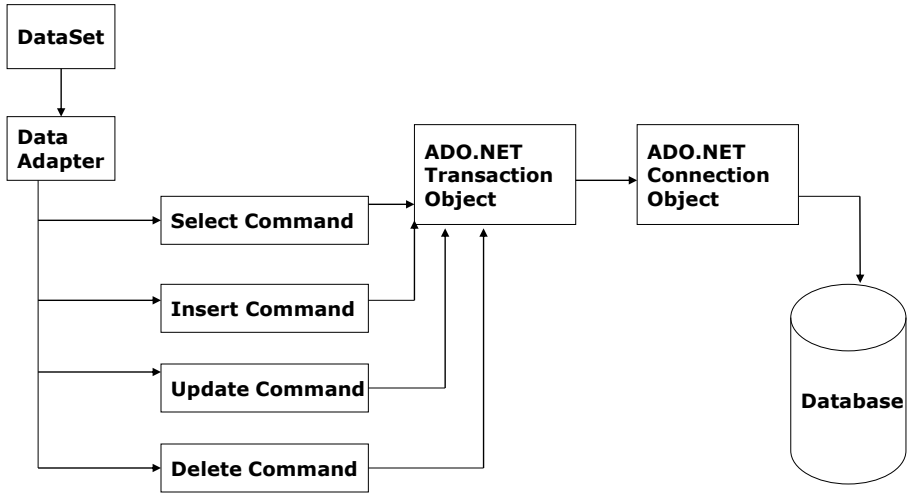
In connected mode, the typical sequence of operations in a transaction will be as follows:

1. Open a database connection.
2. Begin a transaction.
3. Fire queries directly against the connection via the command object.
4. Commit or roll back the transaction.
5. Close the connection.

Implementing transactions in connected mode is relatively simple, as we have everything happening live; however, in disconnected mode, while updating the data back into the database, some care should be taken to account for concurrency issues.

3.3: Managing Data Integrity and Concurrency

ADO.NET Transaction Support – Disconnected Architecture



In disconnected mode, generally, data is fetched first, usually one or more tables into a DataSet object, the connection with the data source is closed, the data is manipulated as required, and then the data is updated back into the database. In this mode, the typical sequence of operations will be as follows:

1. Open a database connection.
2. Fetch the required data in a DataSet object.
3. Close the database connection.
4. Manipulate the data in the DataSet object.
5. Again, open a connection with the database.
6. Start a transaction.
7. Assign the transaction object to the relevant commands on the data adapter.
8. Update the database with changes from the DataSet.
9. Close the connection.

3.3: Managing Data Integrity and Concurrency

Demo

- Transactions

Summary

- Creating and Using DataSet to retrieve Data
  - DataSet object and its members
  - SQLDataAdapter
  - DataTable
  - DataRelation
  - RowState
- Manipulating Database using DataSet
  - SQLCommandBuilder
- Implementing Transaction in ADO.NET
- System.Transaction

Add the notes here.

Review Question

- Question 1: Data Set has the following properties
  - Option 1:Memory Resident data representation
  - Option 2:Disconnected from Data Source
  - Option 3:Both the above
- Question 2: Data Table is created within the
  - Option 1:DataAdpater
  - Option 2:DataSet
  - Option 3:Data Relation
- Question 3: Row State property value depends on which operation has been performed on the row
  - True/False

Add the notes here.