Contents

DataSets, DataTables, and DataViews

8/31/2018 • 2 minutes to read • Edit Online

The ADO.NET DataSet is a memory-resident representation of data that provides a 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, order, and constrain the data, as well as the relationships between the tables.

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

- Programmatically create a DataTable, DataRelation, and Constraint within a DataSet and populate the tables with data.
- Populate the DataSet with tables of data from an existing relational data source using a `DataAdapter`.
- Load and persist the DataSet contents using XML. For more information, see Using XML in a DataSet.

A strongly typed DataSet can also be transported using an XML Web service. The design of the DataSet makes it ideal for transporting data using XML Web services. For an overview of XML Web services, see XML Web Services Overview. For an example of consuming a DataSet from an XML Web service, see Consuming a DataSet from an XML Web Service.

In This Section

Creating a DataSet
Describes the syntax for creating an instance of a DataSet.

Adding a DataTable to a DataSet
Describes how to create and add tables and columns to a DataSet.

Adding DataRelations
Describes how to create relations between tables in a DataSet.

Navigating DataRelations
Describes how to use the relations between tables in a DataSet to return the child or parent rows of a parent-child relationship.

Merging DataSet Contents
Describes how to merge the contents of one DataSet, DataTable, or DataRow array into another DataSet.

Copying DataSet Contents
Describes how to create a copy of a DataSet that can contain schema as well as specified data.

Handling DataSet Events
Describes the events of a DataSet and how to use them.

Typed DataSets
Discusses what a typed DataSet is and how to create and use it.

DataTables
Describes how to create a DataTable, define the schema, and manipulate data.

DataTableReaders
Describes how to create and use a DataTableReader.

DataViews

Describes how to create and work with `DataViews` and work with DataView events.

Using XML in a DataSet

Describes how the DataSet interacts with XML as a data source, including loading and persisting the contents of a DataSet as XML data.

Consuming a DataSet from an XML Web Service

Describes how to create an XML Web service that uses a DataSet to transport data.

Related Sections

What's New in ADO.NET

Introduces features that are new in ADO.NET.

ADO.NET Overview

Provides an introduction to the design and components of ADO.NET.

Populating a DataSet from a DataAdapter

Describes how to load a **DataSet** with data from a data source.

Updating Data Sources with DataAdapters

Describes how to resolve changes to the data in a **DataSet** back to the data source.

Adding Existing Constraints to a DataSet

Describes how to populate a **DataSet** with primary key information from a data source.

See Also

ADO.NET

ADO.NET Managed Providers and DataSet Developer Center

Creating a DataSet

8/31/2018 • 2 minutes to read • Edit Online

You create an instance of a DataSet by calling the DataSet constructor. Optionally specify a name argument. If you do not specify a name for the DataSet, the name is set to "NewDataSet".

You can also create a new DataSet based on an existing DataSet. The new DataSet can be an exact copy of the existing DataSet; a clone of the DataSet that copies the relational structure or schema but that does not contain any of the data from the existing DataSet; or a subset of the DataSet, containing only the modified rows from the existing DataSet using the GetChanges method. For more information, see Copying DataSet Contents.

The following code example demonstrates how to construct an instance of a DataSet.

```
Dim customerOrders As DataSet = New DataSet("CustomerOrders")
```

```
DataSet customerOrders = new DataSet("CustomerOrders");
```

See Also

Populating a DataSet from a DataAdapter
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Adding a DataTable to a DataSet

8/31/2018 • 2 minutes to read • Edit Online

ADO.NET enables you to create DataTable objects and add them to an existing DataSet. You can set constraint information for a DataTable by using the PrimaryKey and Unique properties.

Example

The following example constructs a DataSet, adds a new DataTable object to the DataSet, and then adds three DataColumn objects to the table. Finally, the code sets one column as the primary key column.

```
DataSet customerOrders = new DataSet("CustomerOrders");

DataTable ordersTable = customerOrders.Tables.Add("Orders");

DataColumn pkOrderID =
    ordersTable.Columns.Add("OrderID", typeof(Int32));
ordersTable.Columns.Add("OrderQuantity", typeof(Int32));
ordersTable.Columns.Add("CompanyName", typeof(string));

ordersTable.PrimaryKey = new DataColumn[] { pkOrderID };
```

```
Dim customerOrders As DataSet = New DataSet("CustomerOrders")

Dim ordersTable As DataTable = customerOrders.Tables.Add("Orders")

Dim pkOrderID As DataColumn = ordersTable.Columns.Add( _
    "OrderID", Type.GetType("System.Int32"))
ordersTable.Columns.Add("OrderQuantity", Type.GetType("System.Int32"))
ordersTable.Columns.Add("CompanyName", Type.GetType("System.String"))

ordersTable.PrimaryKey = New DataColumn() {pkOrderID}
```

Case Sensitivity

Two or more tables or relations with the same name, but different casing, can exist in a DataSet. In such cases, references by name to tables and relations are case sensitive. For example, if the DataSet **dataSet** contains tables **Table1** and **table1**, you would reference **Table1** by name as **dataSet.Tables["Table1"]**, and **table1** as **dataSet.Tables["table1"]**. Attempting to reference either of the tables as **dataSet.Tables["TABLE1"]** would generate an exception.

The case-sensitivity behavior does not apply if only one table or relation has a particular name. For example, if the DataSet has only **Table1**, you can reference it using **dataSet.Tables["TABLE1"]**.

> **NOTE**
>
> The CaseSensitive property of the DataSet does not affect this behavior. The CaseSensitive property applies to the data in the DataSet and affects sorting, searching, filtering, enforcing constraints, and so on.

Namespace Support

In versions of ADO.NET earlier than 2.0, two tables could not have the same name, even if they were in different

namespaces. This limitation was removed in ADO.NET 2.0. A DataSet can contain two tables that have the same TableName property value but different Namespace property values.

See Also

DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Adding DataRelations

8/31/2018 • 2 minutes to read • Edit Online

In a DataSet with multiple DataTable objects, you can use DataRelation objects to relate one table to another, to navigate through the tables, and to return child or parent rows from a related table.

The arguments required to create a **DataRelation** are a name for the **DataRelation** being created, and an array of one or more DataColumn references to the columns that serve as the parent and child columns in the relationship. After you have created a **DataRelation**, you can use it to navigate between tables and to retrieve values.

Adding a **DataRelation** to a DataSet adds, by default, a UniqueConstraint to the parent table and a ForeignKeyConstraint to the child table. For more information about these default constraints, see DataTable Constraints.

The following code example creates a **DataRelation** using two DataTable objects in a DataSet. Each DataTable contains a column named **CustID**, which serves as a link between the two DataTable objects. The example adds a single **DataRelation** to the **Relations** collection of the DataSet. The first argument in the example specifies the name of the **DataRelation** being created. The second argument sets the parent **DataColumn** and the third argument sets the child **DataColumn**.

```
customerOrders.Relations.Add("CustOrders", _
  customerOrders.Tables("Customers").Columns("CustID"), _
  customerOrders.Tables("Orders").Columns("CustID"))
```

```
customerOrders.Relations.Add("CustOrders",
  customerOrders.Tables["Customers"].Columns["CustID"],
  customerOrders.Tables["Orders"].Columns["CustID"]);
```

A **DataRelation** also has a **Nested** property which, when set to **true**, causes the rows from the child table to be nested within the associated row from the parent table when written as XML elements using WriteXml . For more information, see Using XML in a DataSet.

See Also

DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Navigating DataRelations

8/31/2018 • 3 minutes to read • Edit Online

One of the primary functions of a DataRelation is to allow navigation from one DataTable to another within a DataSet. This allows you to retrieve all the related DataRow objects in one **DataTable** when given a single **DataRow** from a related **DataTable**. For example, after establishing a **DataRelation** between a table of customers and a table of orders, you can retrieve all the order rows for a particular customer row using **GetChildRows**.

The following code example creates a **DataRelation** between the **Customers** table and the **Orders** table of a **DataSet** and returns all the orders for each customer.

```
DataRelation customerOrdersRelation =
    customerOrders.Relations.Add("CustOrders",
    customerOrders.Tables["Customers"].Columns["CustomerID"],
    customerOrders.Tables["Orders"].Columns["CustomerID"]);

foreach (DataRow custRow in customerOrders.Tables["Customers"].Rows)
{
    Console.WriteLine(custRow["CustomerID"].ToString());

    foreach (DataRow orderRow in custRow.GetChildRows(customerOrdersRelation))
    {
        Console.WriteLine(orderRow["OrderID"].ToString());
    }
}
```

```
Dim customerOrdersRelation As DataRelation = _
   customerOrders.Relations.Add("CustOrders", _
   customerOrders.Tables("Customers").Columns("CustomerID"), _
   customerOrders.Tables("Orders").Columns("CustomerID"))

Dim custRow, orderRow As DataRow

For Each custRow In customerOrders.Tables("Customers").Rows
    Console.WriteLine("Customer ID:" & custRow("CustomerID").ToString())

    For Each orderRow In custRow.GetChildRows(customerOrdersRelation)
        Console.WriteLine(orderRow("OrderID").ToString())
    Next
Next
```

The next example builds on the preceding example, relating four tables together and navigating those relationships. As in the previous example, **CustomerID** relates the **Customers** table to the **Orders** table. For each customer in the **Customers** table, all the child rows in the **Orders** table are determined, in order to return the number of orders a particular customer has and their **OrderID** values.

The expanded example also returns the values from the **OrderDetails** and **Products** tables. The **Orders** table is related to the **OrderDetails** table using **OrderID** to determine, for each customer order, what products and quantities were ordered. Because the **OrderDetails** table only contains the **ProductID** of an ordered product, **OrderDetails** is related to **Products** using **ProductID** in order to return the **ProductName**. In this relation, the **Products** table is the parent and the **Order Details** table is the child. As a result, when iterating through the **OrderDetails** table, **GetParentRow** is called to retrieve the related **ProductName** value.

Notice that when the **DataRelation** is created for the **Customers** and **Orders** tables, no value is specified for the

createConstraints flag (the default is **true**). This assumes that all the rows in the **Orders** table have a **CustomerID** value that exists in the parent **Customers** table. If a **CustomerID** exists in the **Orders** table that does not exist in the **Customers** table, a ForeignKeyConstraint causes an exception to be thrown.

When the child column might contain values that the parent column does not contain, set the **createConstraints** flag to **false** when adding the **DataRelation**. In the example, the **createConstraints** flag is set to **false** for the **DataRelation** between the **Orders** table and the **OrderDetails** table. This enables the application to return all the records from the **OrderDetails** table and only a subset of records from the **Orders** table without generating a run-time exception. The expanded sample generates output in the following format.

```
Customer ID: NORTS
  Order ID: 10517
        Order Date: 4/24/1997 12:00:00 AM
           Product: Filo Mix
          Quantity: 6
           Product: Raclette Courdavault
          Quantity: 4
           Product: Outback Lager
          Quantity: 6
  Order ID: 11057
        Order Date: 4/29/1998 12:00:00 AM
           Product: Outback Lager
          Quantity: 3
```

The following code example is an expanded sample where the values from the **OrderDetails** and **Products** tables are returned, with only a subset of the records in the **Orders** table being returned.

```
DataRelation customerOrdersRelation =
    customerOrders.Relations.Add("CustOrders",
    customerOrders.Tables["Customers"].Columns["CustomerID"],
    customerOrders.Tables["Orders"].Columns["CustomerID"]);

DataRelation orderDetailRelation =
    customerOrders.Relations.Add("OrderDetail",
    customerOrders.Tables["Orders"].Columns["OrderID"],
    customerOrders.Tables["OrderDetails"].Columns["OrderID"], false);

DataRelation orderProductRelation =
    customerOrders.Relations.Add("OrderProducts",
    customerOrders.Tables["Products"].Columns["ProductID"],
    customerOrders.Tables["OrderDetails"].Columns["ProductID"]);

foreach (DataRow custRow in customerOrders.Tables["Customers"].Rows)
{
    Console.WriteLine("Customer ID: " + custRow["CustomerID"]);

    foreach (DataRow orderRow in custRow.GetChildRows(customerOrdersRelation))
    {
        Console.WriteLine("  Order ID: " + orderRow["OrderID"]);
        Console.WriteLine("\tOrder Date: " + orderRow["OrderDate"]);

        foreach (DataRow detailRow in orderRow.GetChildRows(orderDetailRelation))
        {
            Console.WriteLine("\t Product: " +
                detailRow.GetParentRow(orderProductRelation)["ProductName"]);
            Console.WriteLine("\t Quantity: " + detailRow["Quantity"]);
        }
    }
}
```

```
Dim customerOrdersRelation As DataRelation = _
   customerOrders.Relations.Add("CustOrders", _
   customerOrders.Tables("Customers").Columns("CustomerID"), _
   customerOrders.Tables("Orders").Columns("CustomerID"))

Dim orderDetailRelation As DataRelation = _
   customerOrders.Relations.Add("OrderDetail", _
   customerOrders.Tables("Orders").Columns("OrderID"), _
   customerOrders.Tables("OrderDetails").Columns("OrderID"), False)

Dim orderProductRelation As DataRelation = _
   customerOrders.Relations.Add("OrderProducts", _
   customerOrders.Tables("Products").Columns("ProductID"), _
   customerOrders.Tables("OrderDetails").Columns("ProductID"))

Dim custRow, orderRow, detailRow As DataRow

For Each custRow In customerOrders.Tables("Customers").Rows
    Console.WriteLine("Customer ID:" & custRow("CustomerID").ToString())

    For Each orderRow In custRow.GetChildRows(customerOrdersRelation)
        Console.WriteLine("  Order ID: " & orderRow("OrderID").ToString())
        Console.WriteLine(vbTab & "Order Date: " & _
          orderRow("OrderDate").ToString())

        For Each detailRow In orderRow.GetChildRows(orderDetailRelation)
            Console.WriteLine(vbTab & "   Product: " & _
              detailRow.GetParentRow(orderProductRelation) _
              ("ProductName").ToString())
            Console.WriteLine(vbTab & "  Quantity: " & _
              detailRow("Quantity").ToString())
        Next
    Next
Next
```

See Also

DataSets, DataTables, and DataViews

ADO.NET Managed Providers and DataSet Developer Center

Merging DataSet Contents

8/31/2018 • 7 minutes to read • Edit Online

You can use the Merge method to merge the contents of a DataSet, DataTable, or DataRow array into an existing `DataSet`. Several factors and options affect how new data is merged into an existing `DataSet`.

Primary Keys

If the table receiving new data and schema from a merge has a primary key, new rows from the incoming data are matched with existing rows that have the same Original primary key values as those in the incoming data. If the columns from the incoming schema match those of the existing schema, the data in the existing rows is modified. Columns that do not match the existing schema are either ignored or added based on the MissingSchemaAction parameter. New rows with primary key values that do not match any existing rows are appended to the existing table.

If incoming or existing rows have a row state of Added, their primary key values are matched using the Current primary key value of the `Added` row because no `Original` row version exists.

If an incoming table and an existing table contain a column with the same name but different data types, an exception is thrown and the MergeFailed event of the `DataSet` is raised. If an incoming table and an existing table both have defined keys, but the primary keys are for different columns, an exception is thrown and the `MergeFailed` event of the `DataSet` is raised.

If the table receiving new data from a merge does not have a primary key, new rows from the incoming data cannot be matched to existing rows in the table and are instead appended to the existing table.

Table Names and Namespaces

DataTable objects can optionally be assigned a Namespace property value. When Namespace values are assigned, a DataSet can contain multiple DataTable objects with the same TableName value. During merge operations, both TableName and Namespace are used to identify the target of a merge. If no Namespace has been assigned, only the TableName is used to identify the target of a merge.

> **NOTE**
>
> This behavior changed in version 2.0 of the .NET Framework. In version 1.1, namespaces were supported but were ignored during merge operations. For this reason, a DataSet that uses Namespace property values will have different behaviors depending on which version of the .NET Framework you are running. For example, suppose you have two `DataSets` containing `DataTables` with the same TableName property values but different Namespace property values. In version 1.1 of the .NET Framework, the different Namespace names will be ignored when merging the two DataSet objects. However, starting with version 2.0, merging causes two new `DataTables` to be created in the target DataSet. The original `DataTables` will be unaffected by the merge.

PreserveChanges

When you pass a `DataSet`, `DataTable`, or `DataRow` array to the `Merge` method, you can include optional parameters that specify whether or not to preserve changes in the existing `DataSet`, and how to handle new schema elements found in the incoming data. The first of these parameters after the incoming data is a Boolean flag, PreserveChanges, which specifies whether or not to preserve the changes in the existing `DataSet`. If the `PreserveChanges` flag is set to `true`, incoming values do not overwrite existing values in the `Current` row version

of the existing row. If the `PreserveChanges` flag is set to `false`, incoming values do overwrite the existing values in the `Current` row version of the existing row. If the `PreserveChanges` flag is not specified, it is set to `false` by default. For more information about row versions, see Row States and Row Versions.

When `PreserveChanges` is `true`, the data from the existing row is maintained in the Current row version of the existing row, while the data from the Original row version of the existing row is overwritten with the data from the `Original` row version of the incoming row. The RowState of the existing row is set to Modified. The following exceptions apply:

- If the existing row has a `RowState` of `Deleted`, this `RowState` remains `Deleted` and is not set to `Modified`. In this case, the data from the incoming row will still be stored in the `Original` row version of the existing row, overwriting the `Original` row version of the existing row (unless the incoming row has a `RowState` of `Added`).
- If the incoming row has a `RowState` of `Added`, the data from the `Original` row version of the existing row will not be overwritten with data from the incoming row, because the incoming row does not have an `Original` row version.

When `PreserveChanges` is `false`, both the `Current` and `Original` row versions in the existing row are overwritten with the data from the incoming row, and the `RowState` of the existing row is set to the `RowState` of the incoming row. The following exceptions apply:

- If the incoming row has a `RowState` of `Unchanged` and the existing row has a `RowState` of `Modified`, `Deleted`, or `Added`, the `RowState` of the existing row is set to `Modified`.
- If the incoming row has a `RowState` of `Added`, and the existing row has a `RowState` of `Unchanged`, `Modified`, or `Deleted`, the `RowState` of the existing row is set to `Modified`. Also, the data from the `Original` row version of the existing row is not overwritten with data from the incoming row, because the incoming row does not have an `Original` row version.

MissingSchemaAction

You can use the optional MissingSchemaAction parameter of the `Merge` method to specify how `Merge` will handle schema elements in the incoming data that are not part of the existing `DataSet`.

The following table describes the options for `MissingSchemaAction`.

MISSINGSCHEMAACTION OPTION	DESCRIPTION
Add	Add the new schema information to the `DataSet` and populate the new columns with the incoming values. This is the default.
AddWithKey	Add the new schema and primary key information to the `DataSet` and populate the new columns with the incoming values.
Error	Throw an exception if mismatched schema information is encountered.
Ignore	Ignore the new schema information.

Constraints

With the `Merge` method, constraints are not checked until all new data has been added to the existing `DataSet`.

Once the data has been added, constraints are enforced on the current values in the `DataSet`. You must ensure that your code handles any exceptions that might be thrown due to constraint violations.

Consider a case where an existing row in a `DataSet` is an `Unchanged` row with a primary key value of 1. During a merge operation with a `Modified` incoming row with an `Original` primary key value of 2 and a `Current` primary key value of 1, the existing row and the incoming row are not considered matching because the `Original` primary key values differ. However, when the merge is completed and constraints are checked, an exception will be thrown because the `Current` primary key values violate the unique constraint for the primary key column.

> **NOTE**
>
> When rows are inserted into a database table containing an auto incrementing column such as an identity column, the identity column value returned by the insert may not match the value in the `DataSet`, causing the returned rows to be appended instead of merged. For more information, see Retrieving Identity or Autonumber Values.

The following code example merges two `DataSet` objects with differents schemas into one `DataSet` with the combined schemas of the two incoming `DataSet` objects.

```
using (SqlConnection connection =
           new SqlConnection(connectionString))
{
    SqlDataAdapter adapter =
        new SqlDataAdapter(
        "SELECT CustomerID, CompanyName FROM dbo.Customers",
        connection);

    connection.Open();

    DataSet customers = new DataSet();
    adapter.FillSchema(customers, SchemaType.Source, "Customers");
    adapter.Fill(customers, "Customers");

    DataSet orders = new DataSet();
    orders.ReadXml("Orders.xml", XmlReadMode.ReadSchema);
    orders.AcceptChanges();

    customers.Merge(orders, true, MissingSchemaAction.AddWithKey);
```

```
Using connection As SqlConnection = New SqlConnection( _
   connectionString)

    Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
      "SELECT CustomerID, CompanyName FROM Customers", connection)

    connection.Open()

    Dim customers As DataSet = New DataSet()
    adapter.FillSchema(customers, SchemaType.Source, "Customers")
    adapter.Fill(customers, "Customers")

    Dim orders As DataSet = New DataSet()
    orders.ReadXml("Orders.xml", XmlReadMode.ReadSchema)
    orders.AcceptChanges()

    customers.Merge(orders, True, MissingSchemaAction.AddWithKey)
End Using
```

The following code example takes an existing `DataSet` with updates and passes those updates to a `DataAdapter` to be processed at the data source. The results are then merged into the original `DataSet`. After rejecting changes

that resulted in an error, the merged changes are committed with `AcceptChanges`.

```
DataTable customers = dataSet.Tables["Customers"];

// Make modifications to the Customers table.

// Get changes to the DataSet.
DataSet dataSetChanges = dataSet.GetChanges();

// Add an event handler to handle the errors during Update.
adapter.RowUpdated += new SqlRowUpdatedEventHandler(OnRowUpdated);

connection.Open();
adapter.Update(dataSetChanges, "Customers");
connection.Close();

// Merge the updates.
dataSet.Merge(dataSetChanges, true, MissingSchemaAction.Add);

// Reject changes on rows with errors and clear the error.
DataRow[] errRows = dataSet.Tables["Customers"].GetErrors();
foreach (DataRow errRow in errRows)
{
    errRow.RejectChanges();
    errRow.RowError = null;
}

// Commit the changes.
dataSet.AcceptChanges();
```

```
Dim customers As DataTable = dataSet.Tables("Customers")

' Make modifications to the Customers table.

' Get changes to the DataSet.
Dim dataSetChanges As DataSet = dataSet.GetChanges()

' Add an event handler to handle the errors during Update.
AddHandler adapter.RowUpdated, New SqlRowUpdatedEventHandler( _
  AddressOf OnRowUpdated)

connection.Open()
adapter.Update(dataSetChanges, "Customers")
connection.Close()

' Merge the updates.
dataSet.Merge(dataSetChanges, True, MissingSchemaAction.Add)

' Reject changes on rows with errors and clear the error.
Dim errRows() As DataRow = dataSet.Tables("Customers").GetErrors()
Dim errRow As DataRow
For Each errRow In errRows
    errRow.RejectChanges()
    errRow.RowError = Nothing
Next

' Commit the changes.
dataSet.AcceptChanges()
```

```
protected static void OnRowUpdated(
    object sender, SqlRowUpdatedEventArgs args)
{
    if (args.Status == UpdateStatus.ErrorsOccurred)
    {
        args.Row.RowError = args.Errors.Message;
        args.Status = UpdateStatus.SkipCurrentRow;
    }
}
```

```
Private Sub OnRowUpdated( _
    ByVal sender As Object, ByVal args As SqlRowUpdatedEventArgs)
    If args.Status = UpdateStatus.ErrorsOccurred Then
        args.Row.RowError = args.Errors.Message
        args.Status = UpdateStatus.SkipCurrentRow
    End If
End Sub
```

See Also

DataSets, DataTables, and DataViews
Row States and Row Versions
DataAdapters and DataReaders
Retrieving and Modifying Data in ADO.NET
Retrieving Identity or Autonumber Values
ADO.NET Managed Providers and DataSet Developer Center

Copying DataSet Contents

8/31/2018 • 2 minutes to read • Edit Online

You can create a copy of a DataSet so that you can work with data without affecting the original data, or work with a subset of the data from a **DataSet**. When copying a **DataSet**, you can:

- Create an exact copy of the **DataSet**, including the schema, data, row state information, and row versions.
- Create a **DataSet** that contains the schema of an existing **DataSet**, but only rows that have been modified. You can return all rows that have been modified, or specify a specific **DataRowState**. For more information about row states, see Row States and Row Versions.
- Copy the schema, or relational structure, of the **DataSet** only, without copying any rows. Rows can be imported into an existing DataTable using ImportRow.

To create an exact copy of the **DataSet** that includes both schema and data, use the Copy method of the **DataSet**. The following code example shows how to create an exact copy of the **DataSet**.

```
Dim copyDataSet As DataSet = customerDataSet.Copy()
```

```
DataSet copyDataSet = customerDataSet.Copy();
```

To create a copy of a **DataSet** that includes schema and only the data representing **Added**, **Modified**, or **Deleted** rows, use the GetChanges method of the **DataSet**. You can also use **GetChanges** to return only rows with a specified row state by passing a **DataRowState** value when calling **GetChanges**. The following code example shows how to pass a **DataRowState** when calling **GetChanges**.

```
' Copy all changes.
Dim changeDataSet As DataSet = customerDataSet.GetChanges()
' Copy only new rows.
Dim addedDataSetAs DataSet = _
    customerDataSet.GetChanges(DataRowState.Added)
```

```
// Copy all changes.
DataSet changeDataSet = customerDataSet.GetChanges();
// Copy only new rows.
DataSet addedDataSet= customerDataSet.GetChanges(DataRowState.Added);
```

To create a copy of a **DataSet** that only includes schema, use the Clone method of the **DataSet**. You can also add existing rows to the cloned **DataSet** using the **ImportRow** method of the **DataTable**. **ImportRow** adds data, row state, and row version information to the specified table. Column values are added only where the column name matches and the data type is compatible.

The following code example creates a clone of a **DataSet** and then adds the rows from the original **DataSet** to the **Customers** table in the **DataSet** clone for customers where the **CountryRegion** column has the value "Germany".

```
Dim customerDataSet As New DataSet
        customerDataSet.Tables.Add(New DataTable("Customers"))
        customerDataSet.Tables("Customers").Columns.Add("Name", GetType(String))
        customerDataSet.Tables("Customers").Columns.Add("CountryRegion", GetType(String))
        customerDataSet.Tables("Customers").Rows.Add("Juan", "Spain")
        customerDataSet.Tables("Customers").Rows.Add("Johann", "Germany")
        customerDataSet.Tables("Customers").Rows.Add("John", "UK")

Dim germanyCustomers As DataSet = customerDataSet.Clone()

Dim copyRows() As DataRow = _
  customerDataSet.Tables("Customers").Select("CountryRegion = 'Germany'")

Dim customerTable As DataTable = germanyCustomers.Tables("Customers")
Dim copyRow As DataRow

For Each copyRow In copyRows
  customerTable.ImportRow(copyRow)
Next
```

```
DataSet customerDataSet = new DataSet();
customerDataSet.Tables.Add(new DataTable("Customers"));
customerDataSet.Tables["Customers"].Columns.Add("Name", typeof(string));
customerDataSet.Tables["Customers"].Columns.Add("CountryRegion", typeof(string));
customerDataSet.Tables["Customers"].Rows.Add("Juan", "Spain");
customerDataSet.Tables["Customers"].Rows.Add("Johann", "Germany");
customerDataSet.Tables["Customers"].Rows.Add("John", "UK");

DataSet germanyCustomers = customerDataSet.Clone();

DataRow[] copyRows =
  customerDataSet.Tables["Customers"].Select("CountryRegion = 'Germany'");

DataTable customerTable = germanyCustomers.Tables["Customers"];

foreach (DataRow copyRow in copyRows)
  customerTable.ImportRow(copyRow);
```

See Also

DataSet
DataTable
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Handling DataSet Events

8/31/2018 • 2 minutes to read • Edit Online

The DataSet object provides three events: Disposed, Initialized, and MergeFailed.

The MergeFailed Event

The most commonly used event of the `DataSet` object is `MergeFailed`, which is raised when the schema of the `DataSet` objects being merged are in conflict. This occurs when a target and source DataRow have the same primary key value, and the EnforceConstraints property is set to `true`. For example, if the primary key columns of a table being merged are the same between the tables in the two `DataSet` objects, an exception is thrown and the `MergeFailed` event is raised. The MergeFailedEventArgs object passed to the `MergeFailed` event have a Conflict property that identifies the conflict in schema between the two `DataSet` objects, and a Table property that identifies the name of the table in conflict.

The following code fragment demonstrates how to add an event handler for the `MergeFailed` event.

```
AddHandler workDS.MergeFailed, New MergeFailedEventHandler( _
  AddressOf DataSetMergeFailed)

Private Shared Sub DataSetMergeFailed(  _
  sender As Object,args As MergeFailedEventArgs)
  Console.WriteLine("Merge failed for table " & args.Table.TableName)
  Console.WriteLine("Conflict = " & args.Conflict)
End Sub
```

```
workDS.MergeFailed += new MergeFailedEventHandler(DataSetMergeFailed);

private static void DataSetMergeFailed(
  object sender, MergeFailedEventArgs args)
{
  Console.WriteLine("Merge failed for table " + args.Table.TableName);
  Console.WriteLine("Conflict = " + args.Conflict);
}
```

The Initialized Event

The Initialized event occurs after the `DataSet` constructor initializes a new instance of the `DataSet`.

The IsInitialized property returns `true` if the `DataSet` has completed initialization; otherwise it returns `false`. The BeginInit method, which begins the initialization of a `DataSet`, sets IsInitialized to `false`. The EndInit method, which ends the initialization of the `DataSet`, sets it to `true`. These methods are used by the Visual Studio design environment to initialize a `DataSet` that is being used by another component. You will not commonly use them in your code.

The Disposed Event

`DataSet` is derived from the MarshalByValueComponent class, which exposes both the Dispose method and the Disposed event. The Disposed event adds an event handler to listen to the disposed event on the component. You can use the Disposed event of a `DataSet` if you want to execute code when the Dispose method is called. Dispose releases the resources used by the MarshalByValueComponent.

NOTE

The `DataSet` and `DataTable` objects inherit from MarshalByValueComponent and support the ISerializable interface for remoting. These are the only ADO.NET objects that can be remoted. For more information, see Remote Objects.

For information about other events available when working with a `DataSet`, see Handling DataTable Events and Handling DataAdapter Events.

See Also

DataSets, DataTables, and DataViews
Validating Data
Retrieving and Modifying Data in ADO.NET
ADO.NET Managed Providers and DataSet Developer Center

Typed DataSets

8/31/2018 • 2 minutes to read • Edit Online

Along with late bound access to values through weakly typed variables, the DataSet provides access to data through a strongly typed metaphor. Tables and columns that are part of the **DataSet** can be accessed using user-friendly names and strongly typed variables.

A typed **DataSet** is a class that derives from a **DataSet**. As such, it inherits all the methods, events, and properties of a **DataSet**. Additionally, a typed **DataSet** provides strongly typed methods, events, and properties. This means you can access tables and columns by name, instead of using collection-based methods. Aside from the improved readability of the code, a typed **DataSet** also allows the Visual Studio .NET code editor to automatically complete lines as you type.

Additionally, the strongly typed **DataSet** provides access to values as the correct type at compile time. With a strongly typed **DataSet**, type mismatch errors are caught when the code is compiled rather than at run time.

In This Section

Generating Strongly Typed DataSets
Describes how to create and use a strongly typed **DataSet**.

Annotating Typed DataSets
Describes how to annotate the XML Schema definition language (XSD) schema used to generate a strongly typed **DataSet**, to give **DataSet** elements friendly names without altering the underlying schema.

See Also

DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Generating Strongly Typed DataSets

8/31/2018 • 2 minutes to read • Edit Online

Given an XML Schema that complies with the XML Schema definition language (XSD) standard, you can generate a strongly typed DataSet using the XSD.exe tool provided with the Windows Software Development Kit (SDK).

(To create an xsd from database tables, see WriteXmlSchema or Working with Datasets in Visual Studio).

The following code shows the syntax for generating a **DataSet** using this tool.

```
xsd.exe /d /l:CS XSDSchemaFileName.xsd /eld /n:XSDSchema.Namespace
```

In this syntax, the `/d` directive tells the tool to generate a **DataSet**, and the `/l:` tells the tool what language to use (for example, C# or Visual Basic .NET). The optional `/eld` directive specifies that you can use LINQ to DataSet to query against the generated **DataSet.** This option is used when the `/d` option is also specified. For more information, see Querying Typed DataSets. The optional `/n:` directive tells the tool to also generate a namespace for the **DataSet** called **XSDSchema.Namespace**. The output of the command is XSDSchemaFileName.cs, which can be compiled and used in an ADO.NET application. The generated code can be compiled as a library or a module.

The following code shows the syntax for compiling the generated code as a library using the C# compiler (csc.exe).

```
csc.exe /t:library XSDSchemaFileName.cs /r:System.dll /r:System.Data.dll
```

The `/t:` directive tells the tool to compile to a library, and the `/r:` directives specify dependent libraries required to compile. The output of the command is XSDSchemaFileName.dll, which can be passed to the compiler when compiling an ADO.NET application with the `/r:` directive.

The following code shows the syntax for accessing the namespace passed to XSD.exe in an ADO.NET application.

```
Imports XSDSchema.Namespace
```

```
using XSDSchema.Namespace;
```

The following code example uses a typed **DataSet** named **CustomerDataSet** to load a list of customers from the **Northwind** database. Once the data is loaded using the **Fill** method, the example loops through each customer in the **Customers** table using the typed **CustomersRow** (**DataRow**) object. This provides direct access to the **CustomerID** column, as opposed to through the **DataColumnCollection**.

```
Dim customers As CustomerDataSet= New CustomerDataSet()
Dim adapter As SqlDataAdapter New SqlDataAdapter( _
  "SELECT * FROM dbo.Customers;", _
  "Data Source=(local);Integrated " & _
  "Security=SSPI;Initial Catalog=Northwind")

adapter.Fill(customers, "Customers")

Dim customerRow As CustomerDataSet.CustomersRow
For Each customerRow In customers.Customers
  Console.WriteLine(customerRow.CustomerID)
Next
```

```
CustomerDataSet customers = new CustomerDataSet();
SqlDataAdapter adapter = new SqlDataAdapter(
  "SELECT * FROM dbo.Customers;",
  "Data Source=(local);Integrated " +
  "Security=SSPI;Initial Catalog=Northwind");

adapter.Fill(customers, "Customers");

foreach(CustomerDataSet.CustomersRow customerRow in customers.Customers)
  Console.WriteLine(customerRow.CustomerID);
```

Following is the XML Schema used for the example.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="CustomerDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="CustomerDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Customers">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

See Also

DataColumnCollection
DataSet
Typed DataSets
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Annotating Typed DataSets

8/31/2018 • 4 minutes to read • Edit Online

Annotations enable you to modify the names of the elements in your typed DataSet without modifying the underlying schema. Modifying the names of the elements in your underlying schema would cause the typed **DataSet** to refer to objects that do not exist in the data source, as well as lose a reference to the objects that do exist in the data source.

Using annotations, you can customize the names of objects in your typed **DataSet** with more meaningful names, making code more readable and your typed **DataSet** easier for clients to use, while leaving underlying schema intact. For example, the following schema element for the **Customers** table of the **Northwind** database would result in a **DataRow** object name of **CustomersRow** and a DataRowCollection named **Customers**.

```
<xs:element name="Customers">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

A **DataRowCollection** name of **Customers** is meaningful in client code, but a **DataRow** name of **CustomersRow** is misleading because it is a single object. Also, in common scenarios, the object would be referred to without the **Row** identifier and instead would be simply referred to as a **Customer** object. The solution is to annotate the schema and identify new names for the **DataRow** and **DataRowCollection** objects. Following is the annotated version of the previous schema.

```
<xs:element name="Customers" codegen:typedName="Customer" codegen:typedPlural="Customers">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Specifying a **typedName** value of **Customer** will result in a **DataRow** object name of **Customer**. Specifying a **typedPlural** value of **Customers** preserves the **DataRowCollection** name of **Customers**.

The following table shows the annotations available for use.

ANNOTATION	DESCRIPTION
typedName	Name of the object.
typedPlural	Name of a collection of objects.
typedParent	Name of the object when referred to in a parent relationship.
typedChildren	Name of the method to return objects from a child relationship.

ANNOTATION	DESCRIPTION
nullValue	Value if the underlying value is **DBNull**. See the following table for **nullValue** annotations. The default is **_throw**.

The following table shows the values that can be specified for the **nullValue** annotation.

NULLVALUE VALUE	DESCRIPTION
Replacement Value	Specify a value to be returned. The returned value must match the type of the element. For example, use `nullValue="0"` to return 0 for null integer fields.
_throw	Throw an exception. This is the default.
_null	Return a null reference or throw an exception if a primitive type is encountered.
_empty	For strings, return **String.Empty**, otherwise return an object created from an empty constructor. If a primitive type is encountered, throw an exception.

The following table shows default values for objects in a typed **DataSet** and the available annotations.

OBJECT/METHOD/EVENT	DEFAULT	ANNOTATION
DataTable	TableNameDataTable	typedPlural
DataTable Methods	NewTableNameRow AddTableNameRow DeleteTableNameRow	typedName
DataRowCollection	TableName	typedPlural
DataRow	TableNameRow	typedName
DataColumn	DataTable.ColumnNameColumn DataRow.ColumnName	typedName
Property	PropertyName	typedName
Child Accessor	GetChildTableNameRows	typedChildren
Parent Accessor	TableNameRow	typedParent
DataSet Events	TableNameRowChangeEvent TableNameRowChangeEventHandler	typedName

To use typed **DataSet** annotations, you must include the following **xmlns** reference in your XML Schema definition language (XSD) schema. (To create an xsd from database tables, see WriteXmlSchema or Working with Datasets in Visual Studio).

```
xmlns:codegen="urn:schemas-microsoft-com:xml-msprop"
```

The following is a sample annotated schema that exposes the **Customers** table of the **Northwind** database with a relation to the **Orders** table included.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="CustomerDataSet"
      xmlns:codegen="urn:schemas-microsoft-com:xml-msprop"
      xmlns=""
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="CustomerDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Customers" codegen:typedName="Customer" codegen:typedPlural="Customers">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="CustomerID"
codegen:typedName="CustomerID" type="xs:string" minOccurs="0" />
              <xs:element name="CompanyName"
codegen:typedName="CompanyName" type="xs:string" minOccurs="0" />
              <xs:element name="Phone" codegen:typedName="Phone" codegen:nullValue="" type="xs:string"
minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Orders" codegen:typedName="Order" codegen:typedPlural="Orders">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="OrderID" codegen:typedName="OrderID"
type="xs:int" minOccurs="0" />
              <xs:element name="CustomerID"
codegen:typedName="CustomerID"                  codegen:nullValue="" type="xs:string" minOccurs="0" />
              <xs:element name="EmployeeID"
codegen:typedName="EmployeeID" codegen:nullValue="0"
type="xs:int" minOccurs="0" />
              <xs:element name="OrderAdapter"
codegen:typedName="OrderAdapter" codegen:nullValue="1980-01-01T00:00:00"
type="xs:dateTime" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1">
      <xs:selector xpath=".//Customers" />
      <xs:field xpath="CustomerID" />
    </xs:unique>
    <xs:keyref name="CustOrders" refer="Constraint1"
codegen:typedParent="Customer" codegen:typedChildren="GetOrders">
      <xs:selector xpath=".//Orders" />
      <xs:field xpath="CustomerID" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

The following code example uses a strongly typed **DataSet** created from the sample schema. It uses one SqlDataAdapter to populate the **Customers** table and another SqlDataAdapter to populate the **Orders** table. The strongly typed **DataSet** defines the **DataRelations**.

```
' Assumes a valid SqlConnection object named connection.
Dim customerAdapter As SqlDataAdapter = New SqlDataAdapter( _
    "SELECT CustomerID, CompanyName, Phone FROM Customers", &
    connection)
Dim orderAdapter As SqlDataAdapter = New SqlDataAdapter( _
    "SELECT OrderID, CustomerID, EmployeeID, OrderAdapter FROM Orders", &
    connection)

' Populate a strongly typed DataSet.
connection.Open()
Dim customers As CustomerDataSet = New CustomerDataSet()
customerAdapter.Fill(customers, "Customers")
orderAdapter.Fill(customers, "Orders")
connection.Close()

' Add a strongly typed event.
AddHandler customers.Customers.CustomerChanged, &
    New CustomerDataSet.CustomerChangeEventHandler( _
    AddressOf OnCustomerChanged)

' Add a strongly typed DataRow.
Dim newCustomer As CustomerDataSet.Customer = _
    customers.Customers.NewCustomer()
newCustomer.CustomerID = "NEW01"
newCustomer.CompanyName = "My New Company"
customers.Customers.AddCustomer(newCustomer)

' Navigate the child relation.
Dim customer As CustomerDataSet.Customer
Dim order As CustomerDataSet.Order

For Each customer In customers.Customers
  Console.WriteLine(customer.CustomerID)
  For Each order In customer.GetOrders()
    Console.WriteLine(vbTab & order.OrderID)
  Next
Next

Private Shared Sub OnCustomerChanged( _
    sender As Object, e As CustomerDataSet.CustomerChangeEvent)

End Sub
```

```
// Assumes a valid SqlConnection object named connection.
SqlDataAdapter customerAdapter = new SqlDataAdapter(
    "SELECT CustomerID, CompanyName, Phone FROM Customers",
    connection);
SqlDataAdapter orderAdapter = new SqlDataAdapter(
    "SELECT OrderID, CustomerID, EmployeeID, OrderAdapter FROM Orders",
    connection);

// Populate a strongly typed DataSet.
connection.Open();
CustomerDataSet customers = new CustomerDataSet();
customerAdapter.Fill(customers, "Customers");
orderAdapter.Fill(customers, "Orders");
connection.Close();

// Add a strongly typed event.
customers.Customers.CustomerChanged += new
  CustomerDataSet.CustomerChangeEventHandler(OnCustomerChanged);

// Add a strongly typed DataRow.
CustomerDataSet.Customer newCustomer =
    customers.Customers.NewCustomer();
newCustomer.CustomerID = "NEW01";
newCustomer.CompanyName = "My New Company";
customers.Customers.AddCustomer(newCustomer);

// Navigate the child relation.
foreach(CustomerDataSet.Customer customer in customers.Customers)
{
  Console.WriteLine(customer.CustomerID);
  foreach(CustomerDataSet.Order order in customer.GetOrders())
    Console.WriteLine("\t" + order.OrderID);
}

protected static void OnCustomerChanged(object sender, CustomerDataSet.CustomerChangeEvent e)
    {

    }
```

See Also

DataColumnCollection
DataSet
Typed DataSets
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

DataTables

8/31/2018 • 2 minutes to read • Edit Online

A DataSet is made up of a collection of tables, relationships, and constraints. In ADO.NET, DataTable objects are used to represent the tables in a **DataSet**. A **DataTable** represents one table of in-memory relational data; the data is local to the .NET-based application in which it resides, but can be populated from a data source such as Microsoft SQL Server using a **DataAdapter** For more information, see Populating a DataSet from a DataAdapter.

The **DataTable** class is a member of the **System.Data** namespace within the .NET Framework class library. You can create and use a **DataTable** independently or as a member of a **DataSet**, and **DataTable** objects can also be used in conjunction with other .NET Framework objects, including the DataView. You access the collection of tables in a **DataSet** through the **Tables** property of the **DataSet** object.

The schema, or structure of a table is represented by columns and constraints. You define the schema of a **DataTable** using DataColumn objects as well as ForeignKeyConstraint and UniqueConstraint objects. The columns in a table can map to columns in a data source, contain calculated values from expressions, automatically increment their values, or contain primary key values.

In addition to a schema, a **DataTable** must also have rows to contain and order data. The DataRow class represents the actual data contained in a table. You use the **DataRow** and its properties and methods to retrieve, evaluate, and manipulate the data in a table. As you access and change the data within a row, the **DataRow** object maintains both its current and original state.

You can create parent-child relationships between tables using one or more related columns in the tables. You create a relationship between **DataTable** objects using a DataRelation. **DataRelation** objects can then be used to return the related child or parent rows of a particular row. For more information, see Adding DataRelations.

In This Section

Creating a DataTable
Explains how to create a **DataTable** and add it to a **DataSet**.

DataTable Schema Definition
Provides information about creating and using **DataColumn** objects and constraints.

Manipulating Data in a DataTable
Explains how to add, modify, and delete data in a table. Explains how to use **DataTable** events to examine changes to data in the table.

Handling DataTable Events
Provides information about the events available for use with a **DataTable**, including events when column values are modified and rows are added or deleted.

Related Sections

ADO.NET
Describes the ADO.NET architecture and components, and how to use them to access existing data sources and manage application data.

DataSets, DataTables, and DataViews
Provides information about the ADO.NET **DataSet** including how to create relationships between tables.

Constraint
Provides reference information about the **Constraint** object.

DataColumn
Provides reference information about the **DataColumn** object.

DataSet
Provides reference information about the **DataSet** object.

DataTable
Provides reference information about the **DataTable** object.

Class Library Overview
Provides an overview of the .NET Framework class library, including the **System** namespace as well as its second-level namespace, **System.Data**.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Creating a DataTable

8/31/2018 • 2 minutes to read • Edit Online

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 appropriate **DataTable** constructor. You can add it to the **DataSet** by using the **Add** method to add it to the **DataTable** object's **Tables** collection.

You can also create **DataTable** objects within a **DataSet** by using the **Fill** or **FillSchema** methods of the **DataAdapter** object, or from a predefined or inferred XML schema using the **ReadXml**, **ReadXmlSchema**, or **InferXmlSchema** methods of the **DataSet**. Note that after you have added a **DataTable** as a member of the **Tables** collection of one **DataSet**, you cannot add it to the collection of tables of any other **DataSet**.

When you first create a **DataTable**, it does not have a schema (that is, a structure). To define the schema of the table, you must create and add DataColumn objects to the **Columns** collection of the table. You can also define a primary key column for the table, and create and add **Constraint** objects to the **Constraints** collection of the table. After you have defined the schema for a **DataTable**, you can add rows of data to the table by adding **DataRow** objects to the **Rows** collection of the table.

You are not required to supply a value for the TableName property when you create a **DataTable**; you can specify the property at another time, or you can leave it empty. However, when you add a table without a **TableName** value to a **DataSet**, the table will be given an incremental default name of Table*N*, starting with "Table" for Table0.

> **NOTE**
> We recommend that you avoid the "Table*N*" naming convention when you supply a **TableName** value, because the name you supply may conflict with an existing default table name in the **DataSet**. If the supplied name already exists, an exception is thrown.

The following example creates an instance of a **DataTable** object and assigns it the name "Customers."

```
Dim workTable as DataTable = New DataTable("Customers")
```

```
DataTable workTable = new DataTable("Customers");
```

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

```
Dim customers As DataSet = New DataSet
Dim customersTable As DataTable = _
   customers.Tables.Add("CustomersTable")
```

```
DataSet customers = new DataSet();
DataTable customersTable = customers.Tables.Add("CustomersTable");
```

See Also

DataTable

DataTableCollection
DataTables
Populating a DataSet from a DataAdapter
Loading a DataSet from XML
Loading DataSet Schema Information from XML
ADO.NET Managed Providers and DataSet Developer Center

DataTable Schema Definition

8/31/2018 • 2 minutes to read • Edit Online

The schema, or structure, of a table is represented by columns and constraints. You define the schema of a DataTable using DataColumn objects as well as ForeignKeyConstraint and UniqueConstraint objects. The columns in a table can map to columns in a data source, contain calculated values from expressions, automatically increment their values, or contain primary key values.

References by name to columns, relations, and constraints in a table are case-sensitive. Two or more columns, relations, or constraints can therefore exist in a table that have the same name, but that differ in case. For example, you can have **Col1** and **col1**. In such as case, a reference to one of the columns by name must match the case of the column name exactly; otherwise an exception is thrown. For example, if the table **myTable** contains the columns **Col1** and **col1**, you would reference **Col1** by name as **myTable.Columns["Col1"]**, and **col1** as **myTable.Columns["col1"]**. Attempting to reference either of the columns as **myTable.Columns["COL1"]** would generate an exception.

The case-sensitivity rule does not apply if only one column, relation, or constraint with a particular name exists. That is, if no other column, relation, or constraint object in the table matches the name of that particular column, relation, or constraint object, you may reference the object by name using any case, and no exception is thrown. For example, if the table has only **Col1**, you can reference it using **my.Columns["COL1"]**.

> **NOTE**
>
> The CaseSensitive property of the **DataTable** does not affect this behavior. The **CaseSensitive** property applies to the data in a table and affects sorting, searching, filtering, enforcing constraints, and so on, but not to references to the columns, relations, and constraints.

In This Section

Adding Columns to a DataTable
Describes how to define the columns of a table using **DataColumn** objects.

Creating Expression Columns
Explains how the **Expression** property of a column can be used to calculate values based on the values from other columns in the row.

Creating AutoIncrement Columns
Describes how a column can be set to automatically increment numerical values to ensure a unique column value per row.

Defining Primary Keys
Describes how to specify the primary key of a table from one or more **DataColumn** objects.

DataTable Constraints
Describes how to define foreign key and unique constraints for columns in a table.

See Also

DataTables
ADO.NET Managed Providers and DataSet Developer Center

Adding Columns to a DataTable

8/31/2018 • 2 minutes to read • Edit Online

A DataTable contains a collection of DataColumn objects referenced by the **Columns** property of the table. This collection of columns, along with any constraints, defines the schema, or structure, of the table.

You create **DataColumn** objects within a table by using the **DataColumn** constructor, or by calling the **Add** method of the **Columns** property of the table, which is a DataColumnCollection. The **Add** method accepts optional **ColumnName**, **DataType**, and **Expression** arguments and creates a new **DataColumn** as a member of the collection. It also accepts an existing **DataColumn** object and adds it to the collection, and returns a reference to the added **DataColumn** if requested. Because **DataTable** objects are not specific to any data source, .NET Framework types are used when specifying the data type of a **DataColumn**.

The following example adds four columns to a **DataTable**.

```
Dim workTable As DataTable = New DataTable("Customers")

Dim workCol As DataColumn = workTable.Columns.Add( _
    "CustID", Type.GetType("System.Int32"))
workCol.AllowDBNull = false
workCol.Unique = true

workTable.Columns.Add("CustLName", Type.GetType("System.String"))
workTable.Columns.Add("CustFName", Type.GetType("System.String"))
workTable.Columns.Add("Purchases", Type.GetType("System.Double"))
```

```
DataTable workTable = new DataTable("Customers");

DataColumn workCol = workTable.Columns.Add("CustID", typeof(Int32));
workCol.AllowDBNull = false;
workCol.Unique = true;

workTable.Columns.Add("CustLName", typeof(String));
workTable.Columns.Add("CustFName", typeof(String));
workTable.Columns.Add("Purchases", typeof(Double));
```

In the example, notice that the properties for the **CustID** column are set to not allow **DBNull** values and to constrain values to be unique. However, if you define the **CustID** column as the primary key column of the table, the **AllowDBNull** property will automatically be set to **false** and the **Unique** property will automatically be set to **true**. For more information, see Defining Primary Keys.

Caution

If a column name is not supplied for a column, the column is given an incremental default name of Column*N*, starting with "Column1", when it is added to the **DataColumnCollection**. We recommend that you avoid the naming convention of "Column*N*" when you supply a column name, because the name you supply may conflict with an existing default column name in the **DataColumnCollection**. If the supplied name already exists, an exception is thrown.

If you are using XElement as the DataType of a DataColumn in the DataTable, XML serialization will not work when you read in data. For example, if you write out a XmlDocument by using the `DataTable.WriteXml` method, upon serialization to XML there is an additional parent node in the XElement. To work around this problem, use the SqlXml type instead of XElement. `ReadXml` and `WriteXml` work correctly with SqlXml.

See Also

DataColumn
DataColumnCollection
DataTable
DataTable Schema Definition
DataTables
ADO.NET Managed Providers and DataSet Developer Center

Creating Expression Columns

8/31/2018 • 2 minutes to read • Edit Online

You can define an expression for a column, enabling it to contain a value calculated from other column values in the same row or from the column values of multiple rows in the table. To define the expression to be evaluated, use the Expression property of the target column, and use the ColumnName property to refer to other columns in the expression. The DataType for the expression column must be appropriate for the value that the expression returns.

The following table lists several possible uses for expression columns in a table.

EXPRESSION TYPE	EXAMPLE
Comparison	"Total >= 500"
Computation	"UnitPrice * Quantity"
Aggregation	Sum(Price)

You can set the **Expression** property on an existing **DataColumn** object, or you can include the property as the third argument passed to the DataColumn constructor, as shown in the following example.

```
workTable.Columns.Add("Total",Type.GetType("System.Double"))
workTable.Columns.Add("SalesTax", Type.GetType("System.Double"), _
  "Total * 0.086")
```

```
workTable.Columns.Add("Total", typeof(Double));
workTable.Columns.Add("SalesTax", typeof(Double), "Total * 0.086");
```

Expressions can reference other expression columns; however, a circular reference, in which two expressions reference each other, will generate an exception. For rules about writing expressions, see the Expression property of the **DataColumn** class.

See Also

DataColumn
DataSet
DataTable
DataTable Schema Definition
DataTables
ADO.NET Managed Providers and DataSet Developer Center

Creating AutoIncrement Columns

8/31/2018 • 2 minutes to read • Edit Online

To ensure unique column values, you can set the column values to increment automatically when new rows are added to the table. To create an auto-incrementing DataColumn, set the AutoIncrement property of the column to **true**. The DataColumn then starts with the value defined in the AutoIncrementSeed property, and with each row added the value of the **AutoIncrement** column increases by the value defined in the AutoIncrementStep property of the column.

For **AutoIncrement** columns, we recommend that the ReadOnly property of the **DataColumn** be set to **true**.

The following example demonstrates how to create a column that starts with a value of 200 and adds incrementally in steps of 3.

```
Dim workColumn As DataColumn = workTable.Columns.Add( _
    "CustomerID", typeof(Int32))
workColumn.AutoIncrement = true
workColumn.AutoIncrementSeed = 200
workColumn.AutoIncrementStep = 3
```

```
DataColumn workColumn = workTable.Columns.Add(
    "CustomerID", typeof(Int32));
workColumn.AutoIncrement = true;
workColumn.AutoIncrementSeed = 200;
workColumn.AutoIncrementStep = 3;
```

See Also

DataColumn
DataTable Schema Definition
DataTables
ADO.NET Managed Providers and DataSet Developer Center

Defining Primary Keys

8/31/2018 • 2 minutes to read • Edit Online

A database table commonly has a column or group of columns that uniquely identifies each row in the table. This identifying column or group of columns is called the primary key.

When you identify a single DataColumn as the PrimaryKey for a DataTable, the table automatically sets the AllowDBNull property of the column to **false** and the Unique property to **true**. For multiple-column primary keys, only the **AllowDBNull** property is automatically set to **false**.

The **PrimaryKey** property of a DataTable receives as its value an array of one or more **DataColumn** objects, as shown in the following examples. The first example defines a single column as the primary key.

```
workTable.PrimaryKey = New DataColumn() {workTable.Columns("CustID")}

' Or

Dim columns(1) As DataColumn
columns(0) = workTable.Columns("CustID")
workTable.PrimaryKey = columns
```

```
workTable.PrimaryKey = new DataColumn[] {workTable.Columns["CustID"]};

// Or

DataColumn[] columns = new DataColumn[1];
columns[0] = workTable.Columns["CustID"];
workTable.PrimaryKey = columns;
```

The following example defines two columns as a primary key.

```
workTable.PrimaryKey = New DataColumn() {workTable.Columns("CustLName"), _
                                         workTable.Columns("CustFName")}

' Or

Dim keyColumn(2) As DataColumn
keyColumn(0) = workTable.Columns("CustLName")
keyColumn(1) = workTable.Columns("CustFName")
workTable.PrimaryKey = keyColumn
```

```
workTable.PrimaryKey = new DataColumn[] {workTable.Columns["CustLName"],
                                         workTable.Columns["CustFName"]};

// Or

DataColumn[] keyColumn = new DataColumn[2];
keyColumn[0] = workTable.Columns["CustLName"];
keyColumn[1] = workTable.Columns["CustFName"];
workTable.PrimaryKey = keyColumn;
```

See Also

DataTable
DataTable Schema Definition
DataTables
ADO.NET Managed Providers and DataSet Developer Center

DataTable Constraints

8/31/2018 • 5 minutes to read • Edit Online

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 `System.Data.DataSet.EnforceConstraints` property of the DataSet is **true**. For a code example that shows how to set the `EnforceConstraints` property, see the EnforceConstraints reference topic.

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.

ForeignKeyConstraint

A **ForeignKeyConstraint** enforces rules about how updates and deletes to related tables are propagated. For example, if a value in a row of one table is updated or deleted, and that same value is also used in one or more related tables, a **ForeignKeyConstraint** determines what happens in the related tables.

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 following table describes the different settings available for the **DeleteRule** and **UpdateRule** properties of the **ForeignKeyConstraint**.

RULE SETTING	DESCRIPTION
Cascade	Delete or update 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.

A **ForeignKeyConstraint** can restrict, as well as propagate, changes to related columns. Depending on the properties set for the **ForeignKeyConstraint** of a column, if the **EnforceConstraints** property of the **DataSet** is **true**, performing certain operations on the parent row will result in an exception. For example, if the **DeleteRule** property of the **ForeignKeyConstraint** is **None**, a parent row cannot be deleted if it has any child rows.

You can create a foreign key constraint between single columns or between an array of columns by using the **ForeignKeyConstraint** constructor. Pass the resulting **ForeignKeyConstraint** object to the **Add** method of the table's **Constraints** property, which is a **ConstraintCollection**. You can also pass constructor arguments to several overloads of the **Add** method of a **ConstraintCollection** to create a **ForeignKeyConstraint**.

When creating a **ForeignKeyConstraint**, you can pass the **DeleteRule** and **UpdateRule** values to the constructor as arguments, or you can set them as properties as in the following example (where the **DeleteRule** value is set to **None**).

```
Dim custOrderFK As ForeignKeyConstraint = New ForeignKeyConstraint("CustOrderFK", _
  custDS.Tables("CustTable").Columns("CustomerID"), _
  custDS.Tables("OrdersTable").Columns("CustomerID"))
custOrderFK.DeleteRule = Rule.None
' Cannot delete a customer value that has associated existing orders.
custDS.Tables("OrdersTable").Constraints.Add(custOrderFK)
```

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

AcceptRejectRule

Changes to rows can be accepted using the **AcceptChanges** method or canceled using the **RejectChanges** method of the **DataSet**, **DataTable**, or **DataRow**. When a **DataSet** contains **ForeignKeyConstraints**, invoking the **AcceptChanges** or **RejectChanges** methods enforces the **AcceptRejectRule**. The **AcceptRejectRule** property of the **ForeignKeyConstraint** determines which action will be taken on the child rows when **AcceptChanges** or **RejectChanges** is called on the parent row.

The following table lists the available settings for the **AcceptRejectRule**.

RULE SETTING	DESCRIPTION
Cascade	Accept or reject changes to child rows.
None	Take no action on child rows. This is the default.

Example

The following example creates a ForeignKeyConstraint, sets several of its properties, including the AcceptRejectRule, and adds it to the ConstraintCollection of a DataTable object.

```
private void CreateConstraint(DataSet dataSet,
    string table1, string table2,string column1, string column2)
{
   // Declare parent column and child column variables.
   DataColumn parentColumn;
   DataColumn childColumn;
   ForeignKeyConstraint foreignKeyConstraint;

   // Set parent and child column variables.
   parentColumn = dataSet.Tables[table1].Columns[column1];
   childColumn = dataSet.Tables[table2].Columns[column2];
   foreignKeyConstraint = new ForeignKeyConstraint
      ("SupplierForeignKeyConstraint",  parentColumn, childColumn);

   // Set null values when a value is deleted.
   foreignKeyConstraint.DeleteRule = Rule.SetNull;
   foreignKeyConstraint.UpdateRule = Rule.Cascade;
   foreignKeyConstraint.AcceptRejectRule = AcceptRejectRule.None;

   // Add the constraint, and set EnforceConstraints to true.
   dataSet.Tables[table1].Constraints.Add(foreignKeyConstraint);
   dataSet.EnforceConstraints = true;
}
```

```
Private Sub CreateConstraint(dataSet As DataSet, _
   table1 As String, table2 As String, _
   column1 As String, column2 As String)

   ' Declare parent column and child column variables.
   Dim parentColumn As DataColumn
   Dim childColumn As DataColumn
   Dim foreignKeyConstraint As ForeignKeyConstraint

   ' Set parent and child column variables.
   parentColumn = dataSet.Tables(table1).Columns(column1)
   childColumn = dataSet.Tables(table2).Columns(column2)
   foreignKeyConstraint = New ForeignKeyConstraint _
      ("SupplierForeignKeyConstraint", parentColumn, childColumn)

   ' Set null values when a value is deleted.
   foreignKeyConstraint.DeleteRule = Rule.SetNull
   foreignKeyConstraint.UpdateRule = Rule.Cascade
   foreignKeyConstraint.AcceptRejectRule = AcceptRejectRule.None

   ' Add the constraint, and set EnforceConstraints to true.
   dataSet.Tables(table1).Constraints.Add(foreignKeyConstraint)
   dataSet.EnforceConstraints = True
End Sub
```

UniqueConstraint

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 **UniqueConstraint** constructor. Pass the resulting **UniqueConstraint** object to the **Add** method of the table's **Constraints** property, which is a **ConstraintCollection**. You can also pass constructor arguments to several overloads of the **Add** method of a **ConstraintCollection** to create a **UniqueConstraint**. 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.

The following example creates a **UniqueConstraint** for two columns of a **DataTable**.

```
Dim custTable As DataTable = custDS.Tables("Customers")
Dim custUnique As UniqueConstraint = _
    New UniqueConstraint(New DataColumn()   {custTable.Columns("CustomerID"), _
    custTable.Columns("CompanyName")})
custDS.Tables("Customers").Constraints.Add(custUnique)
```

```
DataTable custTable = custDS.Tables["Customers"];
UniqueConstraint custUnique = new UniqueConstraint(new DataColumn[]
    {custTable.Columns["CustomerID"],
    custTable.Columns["CompanyName"]});
custDS.Tables["Customers"].Constraints.Add(custUnique);
```

See Also

DataRelation
DataTable
ForeignKeyConstraint
UniqueConstraint
DataTable Schema Definition
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Manipulating Data in a DataTable

8/31/2018 • 2 minutes to read • Edit Online

After creating a DataTable in a DataSet, you can perform the same activities that you would when using a table in a database. You can add, view, edit, and delete data in the table; you can monitor errors and events; and you can query the data in the table. When modifying data in a **DataTable**, you can also verify whether the changes are accurate, and determine whether to programmatically accept or reject the changes.

In This Section

Adding Data to a DataTable
Explains how to create new rows and add them to a table.

Viewing Data in a DataTable
Describes how to access the data in a row, including original and current versions of the data.

The Load Method
Describes the use of the **Load** method to fill a **DataTable** with rows.

DataTable Edits
Explains how to modify the data in a row, including suspending the changes to a row until the proposed changes are verified and accepted.

Row States and Row Versions
Provides information about the different states of a row.

DataRow Deletion
Describes how to remove a row from a table.

Row Error Information
Explains how to insert error information per row, to help resolve problems with the data within an application.

AcceptChanges and RejectChanges
Explains how to accept or reject the changes made to a row.

See Also

DataTables
Handling DataTable Events
ADO.NET Managed Providers and DataSet Developer Center

Adding Data to a DataTable

8/31/2018 • 2 minutes to read • Edit Online

After you create a DataTable and define its structure using columns and constraints, you can add new rows of data to the table. To add a new row, declare a new variable as type DataRow. A new **DataRow** object is returned when you call the NewRow method. The **DataTable** then creates the **DataRow** object based on the structure of the table, as defined by the DataColumnCollection.

The following example demonstrates how to create a new row by calling the **NewRow** method.

```
Dim workRow As DataRow = workTable.NewRow()
```

```
DataRow workRow = workTable.NewRow();
```

You then can manipulate the newly added row using an index or the column name, as shown in the following example.

```
workRow("CustLName") = "Smith"
workRow(1) = "Smith"
```

```
workRow["CustLName"] = "Smith";
workRow[1] = "Smith";
```

After data is inserted into the new row, the **Add** method is used to add the row to the DataRowCollection, shown in the following code.

```
workTable.Rows.Add(workRow)
```

```
workTable.Rows.Add(workRow);
```

You can also call the **Add** method to add a new row by passing in an array of values, typed as Object, as shown in the following example.

```
workTable.Rows.Add(new Object() {1, "Smith"})
```

```
workTable.Rows.Add(new Object[] {1, "Smith"});
```

Passing an array of values, typed as **Object**, to the **Add** method creates a new row inside the table and sets its column values to the values in the object array. Note that values in the array are matched sequentially to the columns, based on the order in which they appear in the table.

The following example adds 10 rows to the newly created **Customers** table.

```
Dim workRow As DataRow
Dim i As Integer

For i = 0 To 9
  workRow = workTable.NewRow()
  workRow(0) = i
  workRow(1) = "CustName" & I.ToString()
  workTable.Rows.Add(workRow)
Next
```

```
DataRow workRow;

for (int i = 0; i <= 9; i++)
{
  workRow = workTable.NewRow();
  workRow[0] = i;
  workRow[1] = "CustName" + i.ToString();
  workTable.Rows.Add(workRow);
}
```

See Also

DataColumnCollection
DataRow
DataRowCollection
DataTable
Manipulating Data in a DataTable
ADO.NET Managed Providers and DataSet Developer Center

Viewing Data in a DataTable

8/31/2018 • 2 minutes to read • Edit Online

You can access the contents of a DataTable by using the **Rows** and **Columns** collections of the **DataTable**. You can also use the Select method to return subsets of the data in a **DataTable** according to criteria including search criteria, sort order, and row state. Additionally, you can use the Find method of the **DataRowCollection** when searching for a particular row using a primary key value.

The **Select** method of the **DataTable** object returns a set of DataRow objects that match the specified criteria. **Select** takes optional arguments of a filter expression, sort expression, and **DataViewRowState**. The filter expression identifies which rows to return based on **DataColumn** values, such as `LastName = 'Smith'`. The sort expression follows standard SQL conventions for ordering columns, for example `LastName ASC, FirstName ASC`. For rules about writing expressions, see the Expression property of the **DataColumn** class.

> **TIP**
> If you are performing a number of calls to the **Select** method of a **DataTable**, you can increase performance by first creating a DataView for the **DataTable**. Creating the **DataView** indexes the rows of the table. The **Select** method then usees that index, significantly reducing the time to generate the query result. For information about creating a **DataView** for a **DataTable**, see DataViews.

The **Select** method determines which version of the rows to view or manipulate based on a DataViewRowState. The following table describes the possible **DataViewRowState** enumeration values.

DATAVIEWROWSTATE VALUE	DESCRIPTION
CurrentRows	Current rows including unchanged, added, and modified rows.
Deleted	A deleted row.
ModifiedCurrent	A current version, which is a modified version of original data. (See **ModifiedOriginal**.)
ModifiedOriginal	The original version of all modified rows. The current version is available using **ModifiedCurrent**.
Added	A new row.
None	None.
OriginalRows	Original rows, including unchanged and deleted rows.
Unchanged	An unchanged row.

In the following example, the **DataSet** object is filtered so that you are only working with rows whose **DataViewRowState** is set to **CurrentRows**.

```
Dim column As DataColumn
Dim row As DataRow

Dim currentRows() As DataRow = _
    workTable.Select(Nothing, Nothing, DataViewRowState.CurrentRows)

If (currentRows.Length < 1 ) Then
  Console.WriteLine("No Current Rows Found")
Else
  For Each column in workTable.Columns
    Console.Write(vbTab & column.ColumnName)
  Next

  Console.WriteLine(vbTab & "RowState")

  For Each row In currentRows
    For Each column In workTable.Columns
      Console.Write(vbTab & row(column).ToString())
    Next

    Dim rowState As String = _
        System.Enum.GetName(row.RowState.GetType(), row.RowState)
    Console.WriteLine(vbTab & rowState)
  Next
End If
```

```
DataRow[] currentRows = workTable.Select(
    null, null, DataViewRowState.CurrentRows);

if (currentRows.Length < 1 )
  Console.WriteLine("No Current Rows Found");
else
{
  foreach (DataColumn column in workTable.Columns)
    Console.Write("\t{0}", column.ColumnName);

  Console.WriteLine("\tRowState");

  foreach (DataRow row in currentRows)
  {
    foreach (DataColumn column in workTable.Columns)
      Console.Write("\t{0}", row[column]);

    Console.WriteLine("\t" + row.RowState);
  }
}
```

The **Select** method can be used to return rows with differing **RowState** values or field values. The following example returns a **DataRow** array that references all rows that have been deleted, and returns another **DataRow** array that references all rows, ordered by **CustLName**, where the **CustID** column is greater than 5. For information about how to view the information in the **Deleted** row, see Row States and Row Versions.

```
' Retrieve all deleted rows.
Dim deletedRows() As DataRow = workTable.Select(Nothing, Nothing, DataViewRowState.Deleted)

' Retrieve rows where CustID > 5, and order by CustLName.
Dim custRows() As DataRow = workTable.Select( _
    "CustID > 5", "CustLName ASC")
```

```
// Retrieve all deleted rows.
DataRow[] deletedRows = workTable.Select(
    null, null, DataViewRowState.Deleted);

// Retrieve rows where CustID > 5, and order by CustLName.
DataRow[] custRows = workTable.Select("CustID > 5", "CustLName ASC");
```

See Also

DataRow
DataSet
DataTable
DataViewRowState
Manipulating Data in a DataTable
Row States and Row Versions
ADO.NET Managed Providers and DataSet Developer Center

The Load Method

8/31/2018 • 3 minutes to read • Edit Online

You can use the Load method to load a DataTable with rows from a data source. This is an overloaded method which, in its simplest form, accepts a single parameter, a **DataReader**. In this form, it simply loads the **DataTable** with rows. Optionally, you can specify the **LoadOption** parameter to control how data is added to the **DataTable**.

The **LoadOption** parameter is particularly useful in cases where the **DataTable** already contains rows of data, because it describes how incoming data from the data source will be combined with the data already in the table. For example, **PreserveCurrentValues** (the default) specifies that in cases where a row is marked as **Added** in the **DataTable**, the **Original** value or each column is set to the contents of the matching row from the data source. The **Current** value will retain the values assigned when the row was added, and the **RowState** of the row will be set to **Changed**.

The following table gives a short description of the LoadOption enumeration values.

LOADOPTION VALUE	DESCRIPTION
OverwriteRow	If incoming rows have the same **PrimaryKey** value as a row already in the **DataTable**, the **Original** and **Current** values of each column are replaced with the values in the incoming row, and the **RowState** property is set to **Unchanged**. Rows from the data source that do not already exist in the **DataTable** are added with a **RowState** value of **Unchanged**. This option in effect refreshes the contents of the **DataTable** so that it matches the contents of the data source.
PreserveCurrentValues (default)	If incoming rows have the same **PrimaryKey** value as a row already in the **DataTable**, the **Original** value is set to the contents of the incoming row, and the **Current** value is not changed. If the **RowState** is **Added** or **Modified**, it is set to **Modified**. If the **RowState** was **Deleted**, it remains **Deleted**. Rows from the data source that do not already exist in the **DataTable** are added, and the **RowState** is set to **Unchanged**.
UpdateCurrentValues	If incoming rows have the same **PrimaryKey** value as the row already in the **DataTable**, the **Current** value is copied to the **Original** value, and the **Current** value is then set to the contents of the incoming row. If the **RowState** in the **DataTable** was **Added**, the **RowState** remains **Added**. For rows marked as **Modified** or **Deleted**, the **RowState** is **Modified**. Rows from the data source that do not already exist in the **DataTable** are added, and the **RowState** is set to **Added**.

The following sample uses the **Load** method to display a list of birthdays for the employees in the **Northwind** database.

```
Private Sub LoadBirthdays(ByVal connectionString As String)
    ' Assumes that connectionString is a valid connection string
    ' to the Northwind database on SQL Server.
    Dim queryString As String = _
    "SELECT LastName, FirstName, BirthDate " & _
      " FROM dbo.Employees " & _
      "ORDER BY BirthDate, LastName, FirstName"

    ' Open and fill a DataSet.
    Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
        queryString, connectionString)
    Dim employees As New DataSet
    adapter.Fill(employees, "Employees")

    ' Create a SqlDataReader for use with the Load Method.
    Dim reader As DataTableReader = employees.GetDataReader()

    ' Create an instance of DataTable and assign the first
    ' DataTable in the DataSet.Tables collection to it.
    Dim dataTableEmp As DataTable = employees.Tables(0)

    ' Fill the DataTable with data by calling Load and
    ' passing the SqlDataReader.
    dataTableEmp.Load(reader, LoadOption.OverwriteRow)

    ' Loop through the rows collection and display the values
    ' in the console window.
    Dim employeeRow As DataRow
    For Each employeeRow In dataTableEmp.Rows
        Console.WriteLine("{0:MM\\dd\\yyyy}" & ControlChars.Tab & _
          "{1}, {2}", _
          employeeRow("BirthDate"), _
          employeeRow("LastName"), _
          employeeRow("FirstName"))
    Next employeeRow

    ' Keep the window opened to view the contents.
    Console.ReadLine()
End Sub
```

See Also

Manipulating Data in a DataTable
ADO.NET Managed Providers and DataSet Developer Center

DataTable Edits

8/31/2018 • 2 minutes to read • Edit Online

When you make changes to column values in a DataRow, the changes are immediately placed in the current state of the row. The DataRowState is then set to **Modified**, and the changes are accepted or rejected using the AcceptChanges or RejectChanges methods of the **DataRow**. The **DataRow** also provides three methods that you can use to suspend the state of the row while you are editing it. These methods are BeginEdit, EndEdit, and CancelEdit.

When you modify column values in a **DataRow** directly, the **DataRow** manages the column values using the **Current**, **Default**, and **Original** row versions. In addition to these row versions, the **BeginEdit**, **EndEdit**, and **CancelEdit** methods use a fourth row version: **Proposed**. For more information about row versions, see Row States and Row Versions.

The **Proposed** row version exists during an edit operation that begins by calling **BeginEdit** and that ends either by using **EndEdit** or **CancelEdit,** or by calling **AcceptChanges** or **RejectChanges**.

During the edit operation, you can apply validation logic to individual columns by evaluating the **ProposedValue** in the **ColumnChanged** event of the **DataTable**. The **ColumnChanged** event holds **DataColumnChangeEventArgs** that keep a reference to the column that is changing and to the **ProposedValue**. After you evaluate the proposed value, you can either modify it or cancel the edit. When the edit is ended, the row moves out of the **Proposed** state.

You can confirm edits by calling **EndEdit**, or you can cancel them by calling **CancelEdit**. Note that while **EndEdit** does confirm your edits, the **DataSet** does not actually accept the changes until **AcceptChanges** is called. Note also that if you call **AcceptChanges** before you have ended the edit with **EndEdit** or **CancelEdit**, the edit is ended and the **Proposed** row values are accepted for both the **Current** and **Original** row versions. In the same manner, calling **RejectChanges** ends the edit and discards the **Current** and **Proposed** row versions. Calling **EndEdit** or **CancelEdit** after calling **AcceptChanges** or **RejectChanges** has no effect because the edit has already ended.

The following example demonstrates how to use **BeginEdit** with **EndEdit** and **CancelEdit**. The example also checks the **ProposedValue** in the **ColumnChanged** event and decides whether to cancel the edit.

```
Dim workTable As DataTable = New DataTable
workTable.Columns.Add("LastName", Type.GetType("System.String"))

AddHandler workTable.ColumnChanged, _
  New DataColumnChangeEventHandler(AddressOf OnColumnChanged)

Dim workRow As DataRow = workTable.NewRow()
workRow(0) = "Smith"
workTable.Rows.Add(workRow)

workRow.BeginEdit()
' Causes the ColumnChanged event to write a message and cancel the edit.
workRow(0) = ""
workRow.EndEdit()

' Displays "Smith, New".
Console.WriteLine("{0}, {1}", workRow(0), workRow.RowState)

Private Shared Sub OnColumnChanged( _
  sender As Object, args As DataColumnChangeEventArgs)
  If args.Column.ColumnName = "LastName" Then
    If args.ProposedValue.ToString() = "" Then
      Console.WriteLine("Last Name cannot be blank.  Edit canceled.")
      args.Row.CancelEdit()
    End If
  End If
End Sub
```

```
DataTable workTable  = new DataTable();
workTable.Columns.Add("LastName", typeof(String));

workTable.ColumnChanged +=
  new DataColumnChangeEventHandler(OnColumnChanged);

DataRow workRow = workTable.NewRow();
workRow[0] = "Smith";
workTable.Rows.Add(workRow);

workRow.BeginEdit();
// Causes the ColumnChanged event to write a message and cancel the edit.
workRow[0] = "";
workRow.EndEdit();

// Displays "Smith, New".
Console.WriteLine("{0}, {1}", workRow[0], workRow.RowState);

protected static void OnColumnChanged(
  Object sender, DataColumnChangeEventArgs args)
{
  if (args.Column.ColumnName == "LastName")
    if (args.ProposedValue.ToString() == "")
    {
      Console.WriteLine("Last Name cannot be blank. Edit canceled.");
      args.Row.CancelEdit();
    }
}
```

See Also

DataRow
DataTable
DataRowVersion
Manipulating Data in a DataTable

Handling DataTable Events
ADO.NET Managed Providers and DataSet Developer Center

Row States and Row Versions

8/31/2018 • 3 minutes to read • Edit Online

ADO.NET manages rows in tables using row states and versions. A row state indicates the status of a row; row versions maintain the values stored in a row as it is modified, including current, original, and default values. For example, after you have made a modification to a column in a row, the row will have a row state of `Modified`, and two row versions: `Current`, which contains the current row values, and `Original`, which contains the row values before the column was modified.

Each DataRow object has a RowState property that you can examine to determine the current state of the row. The following table gives a brief description of each `RowState` enumeration value.

ROWSTATE VALUE	DESCRIPTION
Unchanged	No changes have been made since the last call to `AcceptChanges` or since the row was created by `DataAdapter.Fill`.
Added	The row has been added to the table, but `AcceptChanges` has not been called.
Modified	Some element of the row has been changed.
Deleted	The row has been deleted from a table, and `AcceptChanges` has not been called.
Detached	The row is not part of any `DataRowCollection`. The `RowState` of a newly created row is set to `Detached`. After the new `DataRow` is added to the `DataRowCollection` by calling the `Add` method, the value of the `RowState` property is set to `Added`. `Detached` is also set for a row that has been removed from a `DataRowCollection` using the `Remove` method, or by the `Delete` method followed by the `AcceptChanges` method.

When `AcceptChanges` is called on a DataSet, DataTable , or DataRow, all rows with a row state of `Deleted` are removed. The remaining rows are given a row state of `Unchanged`, and the values in the `Original` row version are overwritten with the `Current` row version values. When `RejectChanges` is called, all rows with a row state of `Added` are removed. The remaining rows are given a row state of `Unchanged`, and the values in the `Current` row version are overwritten with the `Original` row version values.

You can view the different row versions of a row by passing a DataRowVersion parameter with the column reference, as shown in the following example.

```
Dim custRow As DataRow = custTable.Rows(0)
Dim custID As String = custRow("CustomerID", DataRowVersion.Original).ToString()
```

```
DataRow custRow = custTable.Rows[0];
string custID = custRow["CustomerID", DataRowVersion.Original].ToString();
```

The following table gives a brief description of each `DataRowVersion` enumeration value.

DATAROWVERSION VALUE	DESCRIPTION
Current	The current values for the row. This row version does not exist for rows with a `RowState` of `Deleted`.
Default	The default row version for a particular row. The default row version for an `Added`, `Modified`, or `Deleted` row is `Current`. The default row version for a `Detached` row is `Proposed`.
Original	The original values for the row. This row version does not exist for rows with a `RowState` of `Added`.
Proposed	The proposed values for the row. This row version exists during an edit operation on a row, or for a row that is not part of a `DataRowCollection`.

You can test whether a `DataRow` has a particular row version by calling the HasVersion method and passing a `DataRowVersion` as an argument. For example, `DataRow.HasVersion(DataRowVersion.Original)` will return `false` for newly added rows before `AcceptChanges` has been called.

The following code example displays the values in all the deleted rows of a table. `Deleted` rows do not have a `Current` row version, so you must pass `DataRowVersion.Original` when accessing the column values.

```
Dim catTable As DataTable = catDS.Tables("Categories")

Dim delRows() As DataRow = catTable.Select(Nothing, Nothing, DataViewRowState.Deleted)

Console.WriteLine("Deleted rows:" & vbCrLf)

Dim catCol As DataColumn
Dim delRow As DataRow

For Each catCol In catTable.Columns
  Console.Write(catCol.ColumnName & vbTab)
Next
Console.WriteLine()

For Each delRow In delRows
  For Each catCol In catTable.Columns
    Console.Write(delRow(catCol, DataRowVersion.Original) & vbTab)
  Next
  Console.WriteLine()
Next
```

```
DataTable catTable = catDS.Tables["Categories"];

DataRow[] delRows = catTable.Select(null, null, DataViewRowState.Deleted);

Console.WriteLine("Deleted rows:\n");

foreach (DataColumn catCol in catTable.Columns)
  Console.Write(catCol.ColumnName + "\t");
Console.WriteLine();

foreach (DataRow delRow in delRows)
{
  foreach (DataColumn catCol in catTable.Columns)
    Console.Write(delRow[catCol, DataRowVersion.Original] + "\t");
  Console.WriteLine();
}
```

See Also

Manipulating Data in a DataTable
DataSets, DataTables, and DataViews
DataAdapters and DataReaders
ADO.NET Managed Providers and DataSet Developer Center

DataRow Deletion

8/31/2018 • 2 minutes to read • Edit Online

There are two methods you can use to delete a DataRow object from a DataTable object: the **Remove** method of the DataRowCollection object, and the Delete method of the **DataRow** object. Whereas the Remove method deletes a **DataRow** from the **DataRowCollection**, the Delete method only marks the row for deletion. The actual removal occurs when the application calls the **AcceptChanges** method. By using Delete, you can programmatically check which rows are marked for deletion before actually removing them. When a row is marked for deletion, its RowState property is set to Delete.

Neither Delete nor Remove should be called in a foreach loop while iterating through a DataRowCollection object. Delete nor Remove modify the state of the collection.

When using a DataSet or **DataTable** in conjunction with a **DataAdapter** and a relational data source, use the **Delete** method of the **DataRow** to remove the row. The **Delete** method marks the row as **Deleted** in the **DataSet** or **DataTable** but does not remove it. Instead, when the **DataAdapter** encounters a row marked as **Deleted**, it executes its **DeleteCommand** method to delete the row at the data source. The row can then be permanently removed using the **AcceptChanges** method. If you use **Remove** to delete the row, the row is removed entirely from the table, but the **DataAdapter** will not delete the row at the data source.

The **Remove** method of the **DataRowCollection** takes a **DataRow** as an argument and removes it from the collection, as shown in the following example.

```
workTable.Rows.Remove(workRow)
```

```
workTable.Rows.Remove(workRow);
```

In contrast, the following example demonstrates how to call the **Delete** method on a **DataRow** to change its **RowState** to **Deleted**.

```
workRow.Delete
```

```
workRow.Delete();
```

If a row is marked for deletion and you call the **AcceptChanges** method of the **DataTable** object, the row is removed from the **DataTable**. In contrast, if you call **RejectChanges**, the **RowState** of the row reverts to what it was before being marked as **Deleted**.

> **NOTE**
>
> If the **RowState** of a **DataRow** is **Added**, meaning it has just been added to the table, and it is then marked as **Deleted**, it is removed from the table.

See Also

DataRow
DataRowCollection

DataTable

Manipulating Data in a DataTable

ADO.NET Managed Providers and DataSet Developer Center

Row Error Information

8/31/2018 • 2 minutes to read • Edit Online

To avoid having to respond to row errors while editing values in a DataTable, you can add the error information to the row for later use. The DataRow object provides a RowError property on each row for this purpose. Adding data to the **RowError** property of a **DataRow** sets the HasErrors property of the **DataRow** to **true**. If the **DataRow** is part of a **DataTable**, and **DataRow.HasErrors** is **true**, the **DataTable.HasErrors** property is also **true**. This applies as well to the **DataSet** to which the **DataTable** belongs. When testing for errors, you can check the **HasErrors** property to determine if error information has been added to any rows. If **HasErrors** is **true**, you can use the GetErrors method of the **DataTable** to return and examine only the rows with errors, as shown in the following example.

```
Dim workTable As DataTable = New DataTable("Customers")
workTable.Columns.Add("CustID", Type.GetType("System.Int32"))
workTable.Columns.Add("Total", Type.GetType("System.Double"))

AddHandler workTable.RowChanged, New DataRowChangeEventHandler(AddressOf OnRowChanged)

Dim i  As Int32

For i  = 0 To 10
  workTable.Rows.Add(New Object() {i , i *100})
Next

If workTable.HasErrors Then
  Console.WriteLine("Errors in Table " & workTable.TableName)

  Dim myRow As DataRow

  For Each myRow In workTable.GetErrors()
    Console.WriteLine("CustID = " & myRow("CustID").ToString())
    Console.WriteLine(" Error = " & myRow.RowError & vbCrLf)
  Next
End If

Private Shared Sub OnRowChanged( _
    sender As Object, args As DataRowChangeEventArgs)
  ' Check for zero values.
  If CDbl(args.Row("Total")) = 0 Then args.Row.RowError = _
      "Total cannot be 0."
End Sub
```

```
DataTable  workTable = new DataTable("Customers");
workTable.Columns.Add("CustID", typeof(Int32));
workTable.Columns.Add("Total", typeof(Double));

workTable.RowChanged += new DataRowChangeEventHandler(OnRowChanged);

for (int i = 0; i < 10; i++)
  workTable.Rows.Add(new Object[] {i, i*100});

if (workTable.HasErrors)
{
  Console.WriteLine("Errors in Table " + workTable.TableName);

  foreach (DataRow myRow in workTable.GetErrors())
  {
    Console.WriteLine("CustID = " + myRow["CustID"]);
    Console.WriteLine(" Error = " + myRow.RowError + "\n");
  }
}

protected static void OnRowChanged(
    Object sender, DataRowChangeEventArgs args)
{
  // Check for zero values.
  if (args.Row["Total"].Equals(0D))
    args.Row.RowError = "Total cannot be 0.";
}
```

See Also

DataColumnCollection
DataRow
DataTable
Manipulating Data in a DataTable
ADO.NET Managed Providers and DataSet Developer Center

AcceptChanges and RejectChanges

8/31/2018 • 2 minutes to read • Edit Online

After verifying the accuracy of changes made to data in a DataTable, you can accept the changes using the AcceptChanges method of the DataRow, DataTable, or DataSet, which will set the **Current** row values to be the **Original** values and will set the **RowState** property to **Unchanged**. Accepting or rejecting changes clears out any **RowError** information and sets the **HasErrors** property to **false**. Accepting or rejecting changes can also affect updating data in the data source. For more information, see Updating Data Sources with DataAdapters.

If foreign key constraints exist on the **DataTable**, changes accepted or rejected using **AcceptChanges** and **RejectChanges** are propagated to child rows of the **DataRow** according to the **ForeignKeyConstraint.AcceptRejectRule**. For more information, see DataTable Constraints.

The following example checks for rows with errors, resolves the errors where applicable, and rejects the rows where the error cannot be resolved. Note that, for resolved errors, the **RowError** value is reset to an empty string, causing the **HasErrors** property to be set to **false**. When all the rows with errors have been resolved or rejected, **AcceptChanges** is called to accept all changes for the entire **DataTable**.

```
If workTable.HasErrors Then
  Dim errRow As DataRow

  For Each errRow in workTable.GetErrors()

    If errRow.RowError = "Total cannot exceed 1000." Then
      errRow("Total") = 1000
      errRow.RowError = ""    ' Clear the error.
    Else
      errRow.RejectChanges()
    End If
  Next
End If

workTable.AcceptChanges()
```

```
if (workTable.HasErrors)
{

  foreach (DataRow errRow in workTable.GetErrors())
  {
    if (errRow.RowError == "Total cannot exceed 1000.")
    {
      errRow["Total"] = 1000;
      errRow.RowError = "";    // Clear the error.
    }
    else
      errRow.RejectChanges();
  }
}

workTable.AcceptChanges();
```

See Also

DataRow
DataSet

DataTable

Manipulating Data in a DataTable

ADO.NET Managed Providers and DataSet Developer Center

Handling DataTable Events

8/31/2018 • 6 minutes to read • Edit Online

The DataTable object provides a series of events that can be processed by an application. The following table describes `DataTable` events.

EVENT	DESCRIPTION
Initialized	Occurs after the EndInit method of a `DataTable` is called. This event is intended primarily to support design-time scenarios.
ColumnChanged	Occurs after a value has been successfully changed in a DataColumn.
ColumnChanging	Occurs when a value has been submitted for a `DataColumn`.
RowChanged	Occurs after a `DataColumn` value or the RowState of a DataRow in the `DataTable` has been changed successfully.
RowChanging	Occurs when a change has been submitted for a `DataColumn` value or the `RowState` of a `DataRow` in the `DataTable`.
RowDeleted	Occurs after a `DataRow` in the `DataTable` has been marked as `Deleted`.
RowDeleting	Occurs before a `DataRow` in the `DataTable` is marked as `Deleted`.
TableCleared	Occurs after a call to the Clear method of the `DataTable` has successfully cleared every `DataRow`.
TableClearing	Occurs after the `Clear` method is called but before the `Clear` operation begins.
TableNewRow	Occurs after a new `DataRow` is created by a call to the `NewRow` method of the `DataTable`.
Disposed	Occurs when the `DataTable` is `Disposed`. Inherited from MarshalByValueComponent.

NOTE

Most operations that add or delete rows do not raise the `ColumnChanged` and `ColumnChanging` events. However, the `ReadXml` method does raise `ColumnChanged` and `ColumnChanging` events, unless the `XmlReadMode` is set to `DiffGram` or is set to `Auto` when the XML document being read is a `DiffGram`.

WARNING

Data corruption can occur if data is modified in a `DataSet` from which the `RowChanged` event is raised. No exception will be raised if such data corruption occurs.

Additional Related Events

The Constraints property holds a ConstraintCollection instance. The ConstraintCollection class exposes a CollectionChanged event. This event fires when a constraint is added, modified, or removed from the `ConstraintCollection`.

The Columns property holds a DataColumnCollection instance. The `DataColumnCollection` class exposes a CollectionChanged event. This event fires when a `DataColumn` is added, modified, or removed from the `DataColumnCollection`. Modifications that cause the event to fire include changes to the name, type, expression or ordinal position of a column.

The Tables property of a DataSet holds a DataTableCollection instance. The `DataTableCollection` class exposes both a `CollectionChanged` and a `CollectionChanging` event. These events fire when a `DataTable` is added to or removed from the `DataSet`.

Changes to `DataRows` can also trigger events for an associated DataView. The `DataView` class exposes a ListChanged event that fires when a `DataColumn` value changes or when the composition or sort order of the view changes. The DataRowView class exposes a PropertyChanged event that fires when an associated `DataColumn` value changes.

Sequence of Operations

Here is the sequence of operations that occur when a `DataRow` is added, modified, or deleted:

1. Create the proposed record and apply any changes.
2. Check constraints for non-expression columns.
3. Raise the `RowChanging` or `RowDeleting` events as applicable.
4. Set the proposed record to be the current record.
5. Update any associated indexes.
6. Raise `ListChanged` events for associated `DataView` objects and `PropertyChanged` events for associated `DataRowView` objects.
7. Evaluate all expression columns, but delay checking any constraints on these columns.
8. Raise `ListChanged` events for associated `DataView` objects and `PropertyChanged` events for associated `DataRowView` objects affected by the expression column evaluations.
9. Raise `RowChanged` or `RowDeleted` events as applicable.
10. Check constraints on expression columns.

NOTE

Changes to expression columns never raise `DataTable` events. Changes to expression columns only raise `DataView` and `DataRowView` events. Expression columns can have dependencies on multiple other columns, and can be evaluated multiple times during a single `DataRow` operation. Each expression evaluation raises events, and a single `DataRow` operation can raise multiple `ListChanged` and `PropertyChanged` events when expression columns are affected, possibly including multiple events for the same expression column.

WARNING

Do not throw a NullReferenceException within the `RowChanged` event handler. If a NullReferenceException is thrown within the `RowChanged` event of a `DataTable`, then the `DataTable` will be corrupted.

Example

The following example demonstrates how to create event handlers for the `RowChanged`, `RowChanging`, `RowDeleted`, `RowDeleting`, `ColumnChanged`, `ColumnChanging`, `TableNewRow`, `TableCleared`, and `TableClearing` events. Each event handler displays output in the console window when it is fired.

```
static void DataTableEvents()
{
    DataTable table = new DataTable("Customers");
    // Add two columns, id and name.
    table.Columns.Add("id", typeof(int));
    table.Columns.Add("name", typeof(string));

    // Set the primary key.
    table.Columns["id"].Unique = true;
    table.PrimaryKey = new DataColumn[] { table.Columns["id"] };

    // Add a RowChanged event handler.
    table.RowChanged += new DataRowChangeEventHandler(Row_Changed);

    // Add a RowChanging event handler.
    table.RowChanging += new DataRowChangeEventHandler(Row_Changing);

    // Add a RowDeleted event handler.
    table.RowDeleted += new DataRowChangeEventHandler(Row_Deleted);

    // Add a RowDeleting event handler.
    table.RowDeleting += new DataRowChangeEventHandler(Row_Deleting);

    // Add a ColumnChanged event handler.
    table.ColumnChanged += new
        DataColumnChangeEventHandler(Column_Changed);

    // Add a ColumnChanging event handler.
    table.ColumnChanging += new
        DataColumnChangeEventHandler(Column_Changing);

    // Add a TableNewRow event handler.
    table.TableNewRow += new
        DataTableNewRowEventHandler(Table_NewRow);

    // Add a TableCleared event handler.
    table.TableCleared += new
        DataTableClearEventHandler(Table_Cleared);

    // Add a TableClearing event handler.
    table.TableClearing += new
        DataTableClearEventHandler(Table_Clearing);
```

```
    // Add a customer.
    DataRow row = table.NewRow();
    row["id"] = 1;
    row["name"] = "Customer1";
    table.Rows.Add(row);

    table.AcceptChanges();

    // Change the customer name.
    table.Rows[0]["name"] = "ChangedCustomer1";

    // Delete the row.
    table.Rows[0].Delete();

    // Clear the table.
    table.Clear();
}


private static void Row_Changed(object sender, DataRowChangeEventArgs e)
{
    Console.WriteLine("Row_Changed Event: name={0}; action={1}",
        e.Row["name"], e.Action);
}

private static void Row_Changing(object sender, DataRowChangeEventArgs e)
{
    Console.WriteLine("Row_Changing Event: name={0}; action={1}",
        e.Row["name"], e.Action);
}

private static void Row_Deleted(object sender, DataRowChangeEventArgs e)
{
    Console.WriteLine("Row_Deleted Event: name={0}; action={1}",
        e.Row["name", DataRowVersion.Original], e.Action);
}

private static void Row_Deleting(object sender,
DataRowChangeEventArgs e)
{
    Console.WriteLine("Row_Deleting Event: name={0}; action={1}",
        e.Row["name"], e.Action);
}

private static void Column_Changed(object sender, DataColumnChangeEventArgs e)
{
    Console.WriteLine("Column_Changed Event: ColumnName={0}; RowState={1}",
        e.Column.ColumnName, e.Row.RowState);
}

private static void Column_Changing(object sender, DataColumnChangeEventArgs e)
{
    Console.WriteLine("Column_Changing Event: ColumnName={0}; RowState={1}",
        e.Column.ColumnName, e.Row.RowState);
}

private static void Table_NewRow(object sender,
    DataTableNewRowEventArgs e)
{
    Console.WriteLine("Table_NewRow Event: RowState={0}",
        e.Row.RowState.ToString());
}

private static void Table_Cleared(object sender, DataTableClearEventArgs e)
{
    Console.WriteLine("Table_Cleared Event: TableName={0}; Rows={1}",
        e.TableName, e.Table.Rows.Count.ToString());
}
```

```
private static void Table_Clearing(object sender, DataTableClearEventArgs e)
{
    Console.WriteLine("Table_Clearing Event: TableName={0}; Rows={1}",
        e.TableName, e.Table.Rows.Count.ToString());
}
```

```
Private Sub DataTableEvents()

    Dim table As DataTable = New DataTable("Customers")
    ' Add two columns, id and name.
    table.Columns.Add("id", Type.GetType("System.Int32"))
    table.Columns.Add("name", Type.GetType("System.String"))

    ' Set the primary key.
    table.Columns("id").Unique = True
    table.PrimaryKey = New DataColumn() {table.Columns("id")}

    ' Add a RowChanged event handler.
    AddHandler table.RowChanged, _
           New DataRowChangeEventHandler(AddressOf Row_Changed)

    ' Add a RowChanging event handler.
    AddHandler table.RowChanging, _
           New DataRowChangeEventHandler(AddressOf Row_Changing)

    ' Add a RowDeleted event handler.
    AddHandler table.RowDeleted, New _
           DataRowChangeEventHandler(AddressOf Row_Deleted)

    ' Add a RowDeleting event handler.
    AddHandler table.RowDeleting, New _
           DataRowChangeEventHandler(AddressOf Row_Deleting)

    ' Add a ColumnChanged event handler.
    AddHandler table.ColumnChanged, _
           New DataColumnChangeEventHandler(AddressOf Column_Changed)

    ' Add a ColumnChanging event handler for the table.
    AddHandler table.ColumnChanging, New _
           DataColumnChangeEventHandler(AddressOf Column_Changing)

    ' Add a TableNewRow event handler.
    AddHandler table.TableNewRow, New _
           DataTableNewRowEventHandler(AddressOf Table_NewRow)

    ' Add a TableCleared event handler.
    AddHandler table.TableCleared, New _
           DataTableClearEventHandler(AddressOf Table_Cleared)

    ' Add a TableClearing event handler.
    AddHandler table.TableClearing, New _
           DataTableClearEventHandler(AddressOf Table_Clearing)

    ' Add a customer.
    Dim row As DataRow = table.NewRow()
    row("id") = 1
    row("name") = "Customer1"
    table.Rows.Add(row)

    table.AcceptChanges()

    ' Change the customer name.
    table.Rows(0).Item("name") = "ChangedCustomer1"

    ' Delete the row.
    table.Rows(0).Delete()
```

```
    ' Clear the table.
    table.Clear()
End Sub


Private Sub Row_Changed(ByVal sender As Object, _
    ByVal e As DataRowChangeEventArgs)
    Console.WriteLine("Row_Changed Event: name={0}; action={1}", _
     e.Row("name"), e.Action)
End Sub

Private Sub Row_Changing(ByVal sender As Object, _
    ByVal e As DataRowChangeEventArgs)
    Console.WriteLine("Row_Changing Event: name={0}; action={1}", _
     e.Row("name"), e.Action)
End Sub

Private Sub Row_Deleted(ByVal sender As Object, _
    ByVal e As DataRowChangeEventArgs)
    Console.WriteLine("Row_Deleted Event: name={0}; action={1}", _
     e.Row("name", DataRowVersion.Original), e.Action)
End Sub

Private Sub Row_Deleting(ByVal sender As Object, _
    ByVal e As DataRowChangeEventArgs)
    Console.WriteLine("Row_Deleting Event: name={0}; action={1}", _
       e.Row("name"), e.Action)
End Sub

Private Sub Column_Changed(ByVal sender As Object, _
    ByVal e As DataColumnChangeEventArgs)
    Console.WriteLine("Column_Changed Event: ColumnName={0}; RowState={1}", _
       e.Column.ColumnName, e.Row.RowState)
End Sub

Private Sub Column_Changing(ByVal sender As Object, _
    ByVal e As DataColumnChangeEventArgs)
    Console.WriteLine("Column_Changing Event: ColumnName={0}; RowState={1}", _
       e.Column.ColumnName, e.Row.RowState)
End Sub

Private Sub Table_NewRow(ByVal sender As Object, _
ByVal e As DataTableNewRowEventArgs)
    Console.WriteLine("Table_NewRow Event: RowState={0}", _
       e.Row.RowState.ToString())
End Sub

Private Sub Table_Cleared(ByVal sender As Object, _
    ByVal e As DataTableClearEventArgs)
    Console.WriteLine("Table_Cleared Event: TableName={0}; Rows={1}", _
       e.TableName, e.Table.Rows.Count.ToString())
End Sub

Private Sub Table_Clearing(ByVal sender As Object, _
    ByVal e As DataTableClearEventArgs)
    Console.WriteLine("Table_Clearing Event: TableName={0}; Rows={1}", _
       e.TableName, e.Table.Rows.Count.ToString())
End Sub
```

See Also

Manipulating Data in a DataTable
Handling DataAdapter Events
Handling DataSet Events
ADO.NET Managed Providers and DataSet Developer Center

DataTableReaders

8/31/2018 • 2 minutes to read • Edit Online

The DataTableReader presents the contents of a DataTable or a DataSet in the form of one or more read-only, forward-only result sets.

When you create a **DataTableReader** from a **DataTable**, the resulting **DataTableReader** object contains one result set with the same data as the **DataTable** from which it was created, except for any rows that have been marked as deleted. The columns appear in the same order as in the original **DataTable**.

A **DataTableReader** may contain multiple result sets if it was created by calling CreateDataReader. The results are in the same order as the **DataTables** in the **DataSet** object's Tables collection.

In This Section

Creating a DataReader
Discusses how to create a **DataTableReader** object.

Navigating DataTables
Describes the use of the **Read** method to move through the contents of a **DataTableReader**.

See Also

Retrieving and Modifying Data in ADO.NET
ADO.NET Managed Providers and DataSet Developer Center

Creating a DataReader

8/31/2018 • 2 minutes to read • Edit Online

The DataTable and DataSet classes have a CreateDataReader method that returns the contents of the DataTable or the contents of the DataSet object's Tables collection as one or more read-only, forward-only result sets.

Example

The following console application creates a DataTable instance. The example then passes the filled DataTable to a procedure that calls the CreateDataReader method, which iterates through the results contained within the DataTableReader.

```
static void Main()
{
    TestCreateDataReader(GetCustomers());
    Console.WriteLine("Press any key to continue.");
    Console.ReadKey();
}

private static void TestCreateDataReader(DataTable dt)
{
    // Given a DataTable, retrieve a DataTableReader
    // allowing access to all the tables' data:
    using (DataTableReader reader = dt.CreateDataReader())
    {
        do
        {
            if (!reader.HasRows)
            {
                Console.WriteLine("Empty DataTableReader");
            }
            else
            {
                PrintColumns(reader);
            }
            Console.WriteLine("========================");
        } while (reader.NextResult());
    }
}

private static DataTable GetCustomers()
{
    // Create sample Customers table, in order
    // to demonstrate the behavior of the DataTableReader.
    DataTable table = new DataTable();

    // Create two columns, ID and Name.
    DataColumn idColumn = table.Columns.Add("ID", typeof(int));
    table.Columns.Add("Name", typeof(string));

    // Set the ID column as the primary key column.
    table.PrimaryKey = new DataColumn[] { idColumn };

    table.Rows.Add(new object[] { 1, "Mary" });
    table.Rows.Add(new object[] { 2, "Andy" });
    table.Rows.Add(new object[] { 3, "Peter" });
    table.Rows.Add(new object[] { 4, "Russ" });
    return table;
}

private static void PrintColumns(DataTableReader reader)
{
    // Loop through all the rows in the DataTableReader
    while (reader.Read())
    {
        for (int i = 0; i < reader.FieldCount; i++)
        {
            Console.Write(reader[i] + " ");
        }
        Console.WriteLine();
    }
}
```

```
Sub Main()
  TestCreateDataReader(GetCustomers())
  Console.WriteLine("Press any key to continue.")
  Console.ReadKey()
End Sub

Private Sub TestCreateDataReader(ByVal dt As DataTable)
  ' Given a DataTable, retrieve a DataTableReader
  ' allowing access to all the tables's data:
  Using reader As DataTableReader = dt.CreateDataReader()
    Do
      If Not reader.HasRows Then
        Console.WriteLine("Empty DataTableReader")
      Else
        PrintColumns(reader)
      End If
      Console.WriteLine("========================")
    Loop While reader.NextResult()
  End Using
End Sub

Private Function GetCustomers() As DataTable
  ' Create sample Customers table, in order
  ' to demonstrate the behavior of the DataTableReader.
  Dim table As New DataTable

   ' Create two columns, ID and Name.
  Dim idColumn As DataColumn = table.Columns.Add("ID", GetType(Integer))
  table.Columns.Add("Name", GetType(String))

  ' Set the ID column as the primary key column.
  table.PrimaryKey = New DataColumn() {idColumn}

  table.Rows.Add(New Object() {1, "Mary"})
  table.Rows.Add(New Object() {2, "Andy"})
  table.Rows.Add(New Object() {3, "Peter"})
  table.Rows.Add(New Object() {4, "Russ"})
  Return table
End Function

Private Sub PrintColumns( _
   ByVal reader As DataTableReader)

  ' Loop through all the rows in the DataTableReader.
  Do While reader.Read()
    For i As Integer = 0 To reader.FieldCount - 1
      Console.Write(reader(i).ToString() & " ")
    Next
    Console.WriteLine()
  Loop
End Sub
```

The example displays the following output in the console window:

```
1 Mary
2 Andy
3 Peter
4 Russ
```

See Also

CreateDataReader

CreateDataReader

Navigating DataTables

8/31/2018 • 3 minutes to read • Edit Online

The DataTableReader obtains the contents of one or more DataTable objects in the form of one or more read-only, forward-only result sets.

A DataTableReader may contain multiple result sets if it is created by using the CreateDataReader method. When there is more than one result set, the NextResult method advances the cursor to the next result set. This is a forward-only process. It is not possible to return to a previous result set.

Example

In the following example, the `TestConstructor` method creates two DataTable instances. In order to demonstrate this constructor for the DataTableReader class, the sample creates a new **DataTableReader** based on an array that contains the two **DataTables**, and performs a simple operation, printing the contents from the first few columns to the console window.

```
private static void TestConstructor()
{
    // Create two data adapters, one for each of the two
    // DataTables to be filled.
    DataTable customerDataTable = GetCustomers();
    DataTable productDataTable = GetProducts();

    // Create the new DataTableReader.
    using (DataTableReader reader = new DataTableReader(
                new DataTable[] { customerDataTable, productDataTable }))
    {
        // Print the contents of each of the result sets.
        do
        {
            PrintColumns(reader);
        } while (reader.NextResult());
    }

    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static DataTable GetCustomers()
{
    // Create sample Customers table, in order
    // to demonstrate the behavior of the DataTableReader.
    DataTable table = new DataTable();

    // Create two columns, ID and Name.
    DataColumn idColumn = table.Columns.Add("ID", typeof(int));
    table.Columns.Add("Name", typeof(string ));

    // Set the ID column as the primary key column.
    table.PrimaryKey = new DataColumn[] { idColumn };

    table.Rows.Add(new object[] { 1, "Mary" });
    table.Rows.Add(new object[] { 2, "Andy" });
    table.Rows.Add(new object[] { 3, "Peter" });
    table.Rows.Add(new object[] { 4, "Russ" });
    return table;
}
```

```
private static DataTable GetProducts()
{
    // Create sample Products table, in order
    // to demonstrate the behavior of the DataTableReader.
    DataTable table = new DataTable();

    // Create two columns, ID and Name.
    DataColumn idColumn = table.Columns.Add("ID", typeof(int));
    table.Columns.Add("Name", typeof(string ));

    // Set the ID column as the primary key column.
    table.PrimaryKey = new DataColumn[] { idColumn };

    table.Rows.Add(new object[] { 1, "Wireless Network Card" });
    table.Rows.Add(new object[] { 2, "Hard Drive" });
    table.Rows.Add(new object[] { 3, "Monitor" });
    table.Rows.Add(new object[] { 4, "CPU" });
    return table;
}

private static void PrintColumns(DataTableReader reader)
{
    // Loop through all the rows in the DataTableReader
    while (reader.Read())
    {
        for (int i = 0; i < reader.FieldCount; i++)
        {
            Console.Write(reader[i] + " ");
        }
        Console.WriteLine();
    }
}
```

```
Private Sub TestConstructor()
   ' Create two data adapters, one for each of the two
   ' DataTables to be filled.
   Dim customerDataTable As DataTable = GetCustomers()
   Dim productDataTable As DataTable = GetProducts()

   ' Create the new DataTableReader.
   Using reader As New DataTableReader( _
      New DataTable() {customerDataTable, productDataTable})

      ' Print the contents of each of the result sets.
      Do
         PrintColumns(reader)
      Loop While reader.NextResult()
   End Using

   Console.WriteLine("Press Enter to finish.")
   Console.ReadLine()

End Sub

Private Function GetCustomers() As DataTable
   ' Create sample Customers table, in order
   ' to demonstrate the behavior of the DataTableReader.
   Dim table As New DataTable

   ' Create two columns, ID and Name.
   Dim idColumn As DataColumn = table.Columns.Add("ID", GetType(Integer))
   table.Columns.Add("Name", GetType(String))

   ' Set the ID column as the primary key column.
   table.PrimaryKey = New DataColumn() {idColumn}

   table.Rows.Add(New Object() {1, "Mary"})
```

```
    table.Rows.Add(New Object() {2, "Andy"})
    table.Rows.Add(New Object() {3, "Peter"})
    table.Rows.Add(New Object() {4, "Russ"})
    Return table
End Function

Private Function GetProducts() As DataTable
    ' Create sample Products table, in order
    ' to demonstrate the behavior of the DataTableReader.
    Dim table As New DataTable

    ' Create two columns, ID and Name.
    Dim idColumn As DataColumn = table.Columns.Add("ID", GetType(Integer))
    table.Columns.Add("Name", GetType(String))

    ' Set the ID column as the primary key column.
    table.PrimaryKey = New DataColumn() {idColumn}

    table.Rows.Add(New Object() {1, "Wireless Network Card"})
    table.Rows.Add(New Object() {2, "Hard Drive"})
    table.Rows.Add(New Object() {3, "Monitor"})
    table.Rows.Add(New Object() {4, "CPU"})
    Return table
End Function

Private Sub PrintColumns( _
    ByVal reader As DataTableReader)

    ' Loop through all the rows in the DataTableReader.
    Do While reader.Read()
        For i As Integer = 0 To reader.FieldCount - 1
            Console.Write(reader(i).ToString() & " ")
        Next
        Console.WriteLine()
    Loop
End Sub
```

See Also

DataTableReaders

ADO.NET Managed Providers and DataSet Developer Center

DataViews

8/31/2018 • 2 minutes to read • Edit Online

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 a dynamic view of data in the underlying **DataTable**: the content, ordering, and membership reflect changes as they occur. This behavior differs from the **Select** method of the **DataTable**, which returns a DataRow array from a table based on a particular filter and/or sort order: thiscontent reflects changes to the underlying table, but its membership and ordering remain static. The dynamic capabilities of the **DataView** make it ideal for data-binding applications.

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. Unlike a database view, however, a **DataView** cannot be treated as a table and cannot provide a view of joined tables. You also cannot exclude columns that exist in the source table, nor can you append columns, such as computational columns, that do not exist in the source table.

You can use a DataViewManager to manage view settings for all the tables in a **DataSet**. The **DataViewManager** provides you with a convenient way to manage default view settings for each table. When binding a control to more than one table of a **DataSet**, binding to a **DataViewManager** is the ideal choice.

In This Section

Creating a DataView
Describes how to create a **DataView** for a **DataTable**.

Sorting and Filtering Data
Describes how to set the properties of a **DataView** to return subsets of data rows meeting specific filter criteria, or to return data in a particular sort order.

DataRows and DataRowViews
Describes how to access the data exposed by the **DataView**.

Finding Rows
Describes how to find a particular row in a **DataView**.

ChildViews and Relations
Describes how to create views of data from a parent-child relationship using a **DataView**.

Modifying DataViews
Describes how to modify the data in the underlying **DataTable** via the **DataView**, including enabling or disabling updates.

Handling DataView Events
Describes how to use the **ListChanged** event to receive notification when the contents or order of a **DataView** is being updated.

Managing DataViews
Describes how to use a **DataViewManager** to manage **DataView** settings for each table in a **DataSet**.

Related Sections

ASP.NET Web Applications

Provides overviews and detailed, step-by-step procedures for creating ASP.NET applications, Web Forms, and Web Services.

Windows Applications

Provides detailed information about working with Windows Forms and console applications.

DataSets, DataTables, and DataViews

Describes the **DataSet** object and how you can use it to manage application data.

DataTables

Describes the **DataTable** object and how you can use it to manage application data by itself or as part of a **DataSet**.

ADO.NET

Describes the ADO.NET architecture and components, and how to use ADO.NET to access existing data sources and manage application data.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Creating a DataView

8/31/2018 • 2 minutes to read • Edit Online

There are two ways to create a DataView. You 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. For more information about the additional arguments available for use with the **DataView**, see Sorting and Filtering Data.

Because the index for a **DataView** is built both when the **DataView** is created, and when any of the **Sort**, **RowFilter**, or **RowStateFilter** properties are modified, you achieve best performance by supplying any initial sort order or filtering criteria as constructor arguments when you create the **DataView**. Creating a **DataView** without specifying sort or filter criteria and then setting the **Sort**, **RowFilter**, or **RowStateFilter** properties later causes the index to be built at least twice: once when the **DataView** is created, and again when any of the sort or filter properties are modified.

Note that if you create a **DataView** using the constructor that does not take any arguments, you will not be able to use the **DataView** until you have set the **Table** property.

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**.

```
Dim custDV As DataView = New DataView(custDS.Tables("Customers"), _
    "Country = 'USA'", _
    "ContactName", _
    DataViewRowState.CurrentRows)
```

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

The following code example demonstrates how to obtain a reference to the default **DataView** of a **DataTable** using the **DefaultView** property of the table.

```
Dim custDV As DataView = custDS.Tables("Customers").DefaultView
```

```
DataView custDV = custDS.Tables["Customers"].DefaultView;
```

See Also

DataTable
DataView
DataViews
Sorting and Filtering Data
DataTables
ADO.NET Managed Providers and DataSet Developer Center

Sorting and Filtering Data

8/31/2018 • 2 minutes to read • Edit Online

The DataView provides several ways of sorting and filtering data in a DataTable:

- You can use the Sort property to specify single or multiple column sort orders and include ASC (ascending) and DESC (descending) parameters.
- You can use the ApplyDefaultSort property to automatically create a sort order, in ascending order, based on the primary key column or columns of the table. ApplyDefaultSort only applies when the **Sort** property is a null reference or an empty string, and when the table has a primary key defined.
- You can use the RowFilter property to specify subsets of rows based on their column values. For details about valid expressions for the **RowFilter** property, see the reference information for the Expression property of the DataColumn class.

  If you want to return the results of a particular query on the data, as opposed to providing a dynamic view of a subset of the data, use the Find or FindRows methods of the **DataView** to achieve best performance rather than setting the **RowFilter** property. Setting the **RowFilter** property rebuilds the index for the data, adding overhead to your application and decreasing performance. The **RowFilter** property is best used in a data-bound application where a bound control displays filtered results. The **Find** and **FindRows** methods leverage the current index without requiring the index to be rebuilt. For more information about the **Find** and **FindRows** methods, see Finding Rows.
- You can use the RowStateFilter property to specify which row versions to view. The **DataView** implicitly manages which row version to expose depending upon the **RowState** of the underlying row. For example, if the **RowStateFilter** is set to **DataViewRowState.Deleted**, the **DataView** exposes the **Original** row version of all **Deleted** rows because there is no **Current** row version. You can determine which row version of a row is being exposed by using the **RowVersion** property of the **DataRowView**.

  The following table shows the options for **DataViewRowState**.

DATAVIEWROWSTATE OPTIONS	DESCRIPTION
CurrentRows	The **Current** row version of all **Unchanged**, **Added**, and **Modified** rows. This is the default.
Added	The **Current** row version of all **Added** rows.
Deleted	The **Original** row version of all **Deleted** rows.
ModifiedCurrent	The **Current** row version of all **Modified** rows.
ModifiedOriginal	The **Original** row version of all **Modified** rows.
None	No rows.
OriginalRows	The **Original** row version of all **Unchanged**, **Modified**, and **Deleted** rows.
Unchanged	The **Current** row version of all **Unchanged** rows.

For more information about row states and row versions, see Row States and Row Versions.

The following code example creates a view that shows all the products where the number of units in stock is less than or equal to the reorder level, sorted first by supplier ID and then by product name.

```
Dim prodView As DataView = New DataView(prodDS.Tables("Products"), _
   "UnitsInStock <= ReorderLevel", _
   "SupplierID, ProductName", _
   DataViewRowState.CurrentRows)
```

```
DataView prodView = new DataView(prodDS.Tables["Products"],
   "UnitsInStock <= ReorderLevel",
   "SupplierID, ProductName",
   DataViewRowState.CurrentRows);
```

See Also

DataViewRowState
DataColumn.Expression
DataTable
DataView
DataViews
ADO.NET Managed Providers and DataSet Developer Center

DataRows and DataRowViews

8/31/2018 • 2 minutes to read • Edit Online

A DataView exposes an enumerable collection of DataRowView objects. The **DataRowView** objects expose values as object arrays that are indexed by either the name or the ordinal reference of the column in the underlying table. You can access the DataRow that is exposed by the **DataRowView** by using the Row property of the **DataRowView**.

When you view values by using a **DataRowView**, the RowStateFilter property of the **DataView** determines which row version of the underlying **DataRow** is exposed. For information about accessing different row versions using a **DataRow**, see Row States and Row Versions.

The following code example displays all the current and original values in a table.

```
Dim catView As DataView = New DataView(catDS.Tables("Categories"))
Console.WriteLine("Current Values:")
WriteView(catView)
Console.WriteLine("Original Values:")
catView.RowStateFilter = DataViewRowState.ModifiedOriginal
WriteView(catView)

Public Shared Sub WriteView(thisDataView As DataView)
  Dim rowView As DataRowView
  Dim i As Integer

  For Each rowView In thisDataView
    For i = 0 To thisDataView.Table.Columns.Count - 1
      Console.Write(rowView(i) & vbTab)
    Next
    Console.WriteLine()
  Next
End Sub
```

```
DataView catView = new DataView(catDS.Tables["Categories"]);
Console.WriteLine("Current Values:");
WriteView(catView);
Console.WriteLine("Original Values:");
catView.RowStateFilter = DataViewRowState.ModifiedOriginal;
WriteView(catView);

public static void WriteView(DataView thisDataView)
{
  foreach (DataRowView rowView in thisDataView)
  {
    for (int i = 0; i < thisDataView.Table.Columns.Count; i++)
      Console.Write(rowView[i] + "\t");
    Console.WriteLine();
  }
}
```

See Also

DataRowVersion
DataViewRowState
DataView

DataRowView
DataViews
ADO.NET Managed Providers and DataSet Developer Center

Finding Rows

8/31/2018 • 2 minutes to read • Edit Online

You can search for rows according to their sort key values by using the Find and FindRows methods of the DataView. The case sensitivity of search values in the **Find** and **FindRows** methods is determined by the **CaseSensitive** property of the underlying DataTable. Search values must match existing sort key values in their entirety in order to return a result.

The **Find** method returns an integer with the index of the DataRowView that matches the search criteria. If more than one row matches the search criteria, only the index of the first matching **DataRowView** is returned. If no matches are found, **Find** returns -1.

To return search results that match multiple rows, use the **FindRows** method. **FindRows** works just like the **Find** method, except that it returns a **DataRowView** array that references all matching rows in the **DataView**. If no matches are found, the **DataRowView** array will be empty.

To use the **Find** or **FindRows** methods you must specify a sort order either by setting **ApplyDefaultSort** to **true** or by using the **Sort** property. If no sort order is specified, an exception is thrown.

The **Find** and **FindRows** methods take an array of values as input whose length matches the number of columns in the sort order. In the case of a sort on a single column, you can pass a single value. For sort orders containing multiple columns, you pass an array of objects. Note that for a sort on multiple columns, the values in the object array must match the order of the columns specified in the **Sort** property of the **DataView**.

The following code example shows the **Find** method being called against a **DataView** with a single column sort order.

```
Dim custView As DataView = _
  New DataView(custDS.Tables("Customers"), "", _
  "CompanyName", DataViewRowState.CurrentRows)

Dim rowIndex As Integer = custView.Find("The Cracker Box")

If rowIndex = -1 Then
  Console.WriteLine("No match found.")
Else
  Console.WriteLine("{0}, {1}", _
    custView(rowIndex)("CustomerID").ToString(), _
    custView(rowIndex)("CompanyName").ToString())
End If
```

```
DataView custView = new DataView(custDS.Tables["Customers"], "",
  "CompanyName", DataViewRowState.CurrentRows);

int rowIndex = custView.Find("The Cracker Box");

if (rowIndex == -1)
  Console.WriteLine("No match found.");
else
  Console.WriteLine("{0}, {1}",
    custView[rowIndex]["CustomerID"].ToString(),
    custView[rowIndex]["CompanyName"].ToString());
```

If your **Sort** property specifies multiple columns, you must pass an object array with the search values for each column in the order specified by the **Sort** property, as in the following code example.

```
Dim custView As DataView = _
  New DataView(custDS.Tables("Customers"), "", _
  "CompanyName, ContactName", _
  DataViewRowState.CurrentRows)

Dim foundRows() As DataRowView = _
  custView.FindRows(New object() {"The Cracker Box", "Liu Wong"})

If foundRows.Length = 0 Then
  Console.WriteLine("No match found.")
Else
  Dim myDRV As DataRowView
  For Each myDRV In foundRows
    Console.WriteLine("{0}, {1}", _
      myDRV("CompanyName").ToString(), myDRV("ContactName").ToString())
  Next
End If
```

```
DataView custView = new DataView(custDS.Tables["Customers"], "",
  "CompanyName, ContactName",
  DataViewRowState.CurrentRows);

DataRowView[] foundRows =
  custView.FindRows(new object[] {"The Cracker Box", "Liu Wong"});

if (foundRows.Length == 0)
  Console.WriteLine("No match found.");
else
  foreach (DataRowView myDRV in foundRows)
    Console.WriteLine("{0}, {1}", myDRV["CompanyName"].ToString(),
      myDRV["ContactName"].ToString());
```

See Also

DataTable
DataView
DataViews
ADO.NET Managed Providers and DataSet Developer Center

ChildViews and Relations

8/31/2018 • 2 minutes to read • Edit Online

If a relationship exists between tables in a DataSet, you can create a DataView containing rows from the related child table by using the CreateChildView method of the DataRowView for the rows in the parent table. For example, the following code displays **Categories** and their related **Products** in alphabetical order sorted by **CategoryName** and **ProductName**.

```
Dim catTable As DataTable = catDS.Tables("Categories")
Dim prodTable As DataTable = catDS.Tables("Products")

' Create a relation between the Categories and Products tables.
Dim relation As DataRelation = catDS.Relations.Add("CatProdRel", _
  catTable.Columns("CategoryID"), _
  prodTable.Columns("CategoryID"))

' Create DataViews for the Categories and Products tables.
Dim catView As DataView = New DataView(catTable, "", _
  "CategoryName", DataViewRowState.CurrentRows)
Dim prodView As DataView

' Iterate through the Categories table.
Dim catDRV, prodDRV As DataRowView

For Each catDRV In catView
  Console.WriteLine(catDRV("CategoryName"))

  ' Create a DataView of the child product records.
  prodView = catDRV.CreateChildView(relation)
  prodView.Sort = "ProductName"

  For Each prodDRV In prodView
    Console.WriteLine(vbTab & prodDRV("ProductName"))
  Next
Next
```

```
DataTable catTable = catDS.Tables["Categories"];
DataTable prodTable = catDS.Tables["Products"];

// Create a relation between the Categories and Products tables.
DataRelation relation = catDS.Relations.Add("CatProdRel",
  catTable.Columns["CategoryID"],
                                                          prodTable.Columns["CategoryID"]);

// Create DataViews for the Categories and Products tables.
DataView catView = new DataView(catTable, "", "CategoryName",
  DataViewRowState.CurrentRows);
DataView prodView;

// Iterate through the Categories table.
foreach (DataRowView catDRV in catView)
{
  Console.WriteLine(catDRV["CategoryName"]);

  // Create a DataView of the child product records.
  prodView = catDRV.CreateChildView(relation);
  prodView.Sort = "ProductName";

  foreach (DataRowView prodDRV in prodView)
    Console.WriteLine("\t" + prodDRV["ProductName"]);
}
```

See Also

DataSet
DataView
DataRowView
DataViews
ADO.NET Managed Providers and DataSet Developer Center

Modifying DataViews

8/31/2018 • 2 minutes to read • Edit Online

You can use the DataView to add, delete, or modify rows of data in the underlying table. The ability to use the **DataView** to modify data in the underlying table is controlled by setting one of three Boolean properties of the **DataView**. These properties are AllowNew, AllowEdit, and AllowDelete. They are set to **true** by default.

If **AllowNew** is **true**, you can use the AddNew method of the **DataView** to create a new DataRowView. Note that a new row is not actually added to the underlying DataTable until the EndEdit method of the **DataRowView** is called. If the CancelEdit method of the **DataRowView** is called, the new row is discarded. Note also that you can edit only one **DataRowView** at a time. If you call the **AddNew** or **BeginEdit** method of the **DataRowView** while a pending row exists, **EndEdit** is implicitly called on the pending row. When **EndEdit** is called, the changes are applied to the underlying **DataTable** and can later be committed or rejected using the **AcceptChanges** or **RejectChanges** methods of the **DataTable**, **DataSet**, or **DataRow** object. If **AllowNew** is **false**, an exception is thrown if you call the **AddNew** method of the **DataRowView**.

If **AllowEdit** is **true**, you can modify the contents of a **DataRow** via the **DataRowView**. You can confirm changes to the underlying row using **DataRowView.EndEdit** or reject the changes using **DataRowView.CancelEdit**. Note that only one row can be edited at a time. If you call the **AddNew** or **BeginEdit** methods of the **DataRowView** while a pending row exists, **EndEdit** is implicitly called on the pending row. When **EndEdit** is called, proposed changes are placed in the **Current** row version of the underlying **DataRow** and can later be committed or rejected using the **AcceptChanges** or **RejectChanges** methods of the **DataTable**, **DataSet**, or **DataRow** object. If **AllowEdit** is **false**, an exception is thrown if you attempt to modify a value in the **DataView**.

When an existing **DataRowView** is being edited, events of the underlying **DataTable** will still be raised with the proposed changes. Note that if you call **EndEdit** or **CancelEdit** on the underlying **DataRow**, pending changes will be applied or canceled regardless of whether **EndEdit** or **CancelEdit** is called on the **DataRowView**.

If **AllowDelete** is **true**, you can delete rows from the **DataView** by using the **Delete** method of the **DataView** or **DataRowView** object, and the rows are deleted from the underlying **DataTable**. You can later commit or reject the deletes using **AcceptChanges** or **RejectChanges** respectively. If **AllowDelete** is **false**, an exception is thrown if you call the **Delete** method of the **DataView** or **DataRowView**.

The following code example disables using the **DataView** to delete rows and adds a new row to the underlying table using the **DataView**.

```
Dim custTable As DataTable = custDS.Tables("Customers")
Dim custView As DataView = custTable.DefaultView
custView.Sort = "CompanyName"

custView.AllowDelete = False

Dim newDRV As DataRowView = custView.AddNew()
newDRV("CustomerID") = "ABCDE"
newDRV("CompanyName") = "ABC Products"
newDRV.EndEdit()
```

```
DataTable custTable = custDS.Tables["Customers"];
DataView custView = custTable.DefaultView;
custView.Sort = "CompanyName";

custView.AllowDelete = false;

DataRowView newDRV = custView.AddNew();
newDRV["CustomerID"] = "ABCDE";
newDRV["CompanyName"] = "ABC Products";
newDRV.EndEdit();
```

See Also

DataTable
DataView
DataRowView
DataViews
ADO.NET Managed Providers and DataSet Developer Center

Handling DataView Events

8/31/2018 • 2 minutes to read • Edit Online

You can use the ListChanged event of the DataView to determine if a view has been updated. Updates that raise the event include adding, deleting, or modifying a row in the underlying table; adding or deleting a column to the schema of the underlying table; and a change in a parent or child relationship. The **ListChanged** event also notifies you if the list of rows you are viewing has changed significantly due to the application of a new sort order or a filter.

The **ListChanged** event implements the **ListChangedEventHandler** delegate of the System.ComponentModel namespace and takes as input a ListChangedEventArgs object. You can determine what type of change has occurred using the ListChangedType enumeration value in the **ListChangedType** property of the **ListChangedEventArgs** object. For changes that involve adding, deleting, or moving rows, the new index of the added or moved row and the previous index of the deleted row can be accessed using the **NewIndex** property of the **ListChangedEventArgs** object. In the case of a moved row, the previous index of the moved row can be accessed using the **OldIndex** property of the **ListChangedEventArgs** object.

The **DataViewManager** also exposes a **ListChanged** event to notify you if a table has been added or removed, or if a change has been made to the **Relations** collection of the underlying **DataSet**.

The following code example shows how to add a **ListChanged** event handler.

```
AddHandler custView.ListChanged, _
  New System.ComponentModel.ListChangedEventHandler( _
  AddressOf OnListChanged)

Private Shared Sub OnListChanged( _
  sender As Object, args As System.ComponentModel.ListChangedEventArgs)
  Console.WriteLine("ListChanged:")
  Console.WriteLine(vbTab & "    Type = " & _
    System.Enum.GetName(args.ListChangedType.GetType(), _
    args.ListChangedType))
  Console.WriteLine(vbTab & "OldIndex = " & args.OldIndex)
  Console.WriteLine(vbTab & "NewIndex = " & args.NewIndex)
End Sub
```

```
custView.ListChanged  += new
  System.ComponentModel.ListChangedEventHandler(OnListChanged);

protected static void OnListChanged(object sender,
  System.ComponentModel.ListChangedEventArgs args)
{
  Console.WriteLine("ListChanged:");
  Console.WriteLine("\t    Type = " + args.ListChangedType);
  Console.WriteLine("\tOldIndex = " + args.OldIndex);
  Console.WriteLine("\tNewIndex = " + args.NewIndex);
}
```

See Also

DataView
ListChangedEventHandler
DataViews

Managing DataViews

8/31/2018 • 2 minutes to read • Edit Online

You can use a DataViewManager to manage view settings for all the tables in a DataView. If you have a control that you want to bind to multiple tables, such as a grid that navigates relationships, a **DataViewManager** is ideal.

The **DataViewManager** contains a collection of DataViewSetting objects that are used to set the view setting of the tables in the DataSet. The DataViewSettingCollection contains one DataViewSetting object for each table in a **DataSet**. You can set the default **ApplyDefaultSort**, **Sort**, **RowFilter**, and **RowStateFilter** properties of the referenced table by using its **DataViewSetting**. You can reference the **DataViewSetting** for a particular table by name or ordinal reference, or by passing a reference to that specific table object. You can access the collection of **DataViewSetting** objects in a **DataViewManager** by using the **DataViewSettings** property.

The following code example fills a **DataSet** with the SQL Server **Northwind** database tables **Customers**, **Orders**, and **Order Details**, creates the relationships between the tables, uses a **DataViewManager** to set default **DataView** settings, and binds a **DataGrid** to the **DataViewManager**. The example sets the default **DataView** settings for all tables in the **DataSet** to sort by the primary key of the table (**ApplyDefaultSort** = **true**), and then modifies the sort order of the **Customers** table to sort by **CompanyName**.

```
' Assumes connection is a valid SqlConnection to Northwind.
' Create a Connection, DataAdapters, and a DataSet.
Dim custDA As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT CustomerID, CompanyName FROM Customers", connection)
Dim orderDA As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT OrderID, CustomerID FROM Orders", connection)
Dim ordDetDA As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT OrderID, ProductID, Quantity FROM [Order Details]", connection)

Dim custDS As DataSet = New DataSet()

' Open the Connection.
connection.Open()

    ' Fill the DataSet with schema information and data.
    custDA.MissingSchemaAction = MissingSchemaAction.AddWithKey
    orderDA.MissingSchemaAction = MissingSchemaAction.AddWithKey
    ordDetDA.MissingSchemaAction = MissingSchemaAction.AddWithKey

    custDA.Fill(custDS, "Customers")
    orderDA.Fill(custDS, "Orders")
    ordDetDA.Fill(custDS, "OrderDetails")

    ' Close the Connection.
    connection.Close()

    ' Create relationships.
    custDS.Relations.Add("CustomerOrders", _
          custDS.Tables("Customers").Columns("CustomerID"), _
          custDS.Tables("Orders").Columns("CustomerID"))

    custDS.Relations.Add("OrderDetails", _
          custDS.Tables("Orders").Columns("OrderID"), _
          custDS.Tables("OrderDetails").Columns("OrderID"))

' Create default DataView settings.
Dim viewManager As DataViewManager = New DataViewManager(custDS)

Dim viewSetting As DataViewSetting
For Each viewSetting In viewManager.DataViewSettings
  viewSetting.ApplyDefaultSort = True
Next

viewManager.DataViewSettings("Customers").Sort = "CompanyName"

' Bind to a DataGrid.
Dim grid As System.Windows.Forms.DataGrid = New System.Windows.Forms.DataGrid()
grid.SetDataBinding(viewManager, "Customers")
```

```
// Assumes connection is a valid SqlConnection to Northwind.
// Create a Connection, DataAdapters, and a DataSet.
SqlDataAdapter custDA = new SqlDataAdapter(
  "SELECT CustomerID, CompanyName FROM Customers", connection);
SqlDataAdapter orderDA = new SqlDataAdapter(
  "SELECT OrderID, CustomerID FROM Orders", connection);
SqlDataAdapter ordDetDA = new SqlDataAdapter(
  "SELECT OrderID, ProductID, Quantity FROM [Order Details]", connection);

DataSet custDS = new DataSet();

// Open the Connection.
connection.Open();

    // Fill the DataSet with schema information and data.
    custDA.MissingSchemaAction = MissingSchemaAction.AddWithKey;
    orderDA.MissingSchemaAction = MissingSchemaAction.AddWithKey;
    ordDetDA.MissingSchemaAction = MissingSchemaAction.AddWithKey;

    custDA.Fill(custDS, "Customers");
    orderDA.Fill(custDS, "Orders");
    ordDetDA.Fill(custDS, "OrderDetails");

    // Close the Connection.
    connection.Close();

    // Create relationships.
    custDS.Relations.Add("CustomerOrders",
          custDS.Tables["Customers"].Columns["CustomerID"],
          custDS.Tables["Orders"].Columns["CustomerID"]);

    custDS.Relations.Add("OrderDetails",
          custDS.Tables["Orders"].Columns["OrderID"],
          custDS.Tables["OrderDetails"].Columns["OrderID"]);

// Create default DataView settings.
DataViewManager viewManager = new DataViewManager(custDS);

foreach (DataViewSetting viewSetting in viewManager.DataViewSettings)
  viewSetting.ApplyDefaultSort = true;

viewManager.DataViewSettings["Customers"].Sort = "CompanyName";

// Bind to a DataGrid.
System.Windows.Forms.DataGrid grid = new System.Windows.Forms.DataGrid();
grid.SetDataBinding(viewManager, "Customers");
```

See Also

DataSet
DataViewManager
DataViewSetting
DataViewSettingCollection
DataViews
ADO.NET Managed Providers and DataSet Developer Center

Creating a DataTable from a DataView

8/31/2018 • 4 minutes to read • Edit Online

Once you have retrieved data from a data source, and have filled a DataTable with the data, you may want to sort, filter, or otherwise limit the returned data without retrieving it again. The DataView class makes this possible. In addition, if you need to create a new DataTable from the DataView, you can use the ToTable method to copy all the rows and columns, or a subset of the data into a new DataTable. The ToTable method provides overloads to:

- Create a DataTable containing columns that are a subset of the columns in the DataView.
- Create a DataTable that includes only distinct rows from the DataView, analogously to the DISTINCT keyword in Transact-SQL.

Example

The following console application example creates a DataTable that contains data from the **Person.Contact** table in the **AdventureWorks** sample database. Next, the example creates a sorted and filtered DataView based on the DataTable. After displaying the contents of the DataTable and the DataView, the example creates a new DataTable from the DataView by calling the ToTable method, selecting only a subset of the available columns. Finally, the example displays the contents of the new DataTable.

```
Private Sub DemonstrateDataView()
    ' Retrieve a DataTable from the AdventureWorks sample database.
    ' connectionString is assumed to be a valid connection string.
    Dim adapter As New SqlDataAdapter( _
       "SELECT FirstName, LastName, EmailAddress FROM Person.Contact WHERE FirstName LIKE 'Mich%'",
connectionString)
    Dim table As New DataTable

    adapter.Fill(table)
    Console.WriteLine("Original table name: " & table.TableName)
    ' Print current table values.
    PrintTableOrView(table, "Current Values in Table")

    ' Now create a DataView based on the DataTable.
    ' Sort and filter the data.
    Dim view As DataView = table.DefaultView
    view.Sort = "LastName, FirstName"
    view.RowFilter = "LastName > 'M'"
    PrintTableOrView(view, "Current Values in View")

    ' Create a new DataTable based on the DataView,
    ' requesting only two columns with distinct values
    ' in the columns.
    Dim newTable As DataTable = view.ToTable("UniqueLastNames", True, "FirstName", "LastName")
    PrintTableOrView(newTable, "Table created from DataView")
    Console.WriteLine("New table name: " & newTable.TableName)

    Console.WriteLine("Press any key to continue.")
    Console.ReadKey()
    End Sub

Private Sub PrintTableOrView(ByVal dv As DataView, ByVal label As String)
    Dim sw As System.IO.StringWriter
    Dim output As String
    Dim table As DataTable = dv.Table

    Console.WriteLine(label)
```

```
    ' Loop through each row in the view.
    For Each rowView As DataRowView In dv
        sw = New System.IO.StringWriter

        ' Loop through each column.
        For Each col As DataColumn In table.Columns
            ' Output the value of each column's data.
            sw.Write(rowView(col.ColumnName).ToString() & ", ")
        Next
        output = sw.ToString
        ' Trim off the trailing ", ", so the output looks correct.
        If output.Length > 2 Then
            output = output.Substring(0, output.Length - 2)
        End If
        ' Display the row in the console window.
        Console.WriteLine(output)
    Next
    Console.WriteLine()
End Sub

Private Sub PrintTableOrView(ByVal table As DataTable, ByVal label As String)
    Dim sw As System.IO.StringWriter
    Dim output As String

    Console.WriteLine(label)

    ' Loop through each row in the table.
    For Each row As DataRow In table.Rows
        sw = New System.IO.StringWriter
        ' Loop through each column.
        For Each col As DataColumn In table.Columns
            ' Output the value of each column's data.
            sw.Write(row(col).ToString() & ", ")
        Next
        output = sw.ToString
        ' Trim off the trailing ", ", so the output looks correct.
        If output.Length > 2 Then
            output = output.Substring(0, output.Length - 2)
        End If
        ' Display the row in the console window.
        Console.WriteLine(output)
    Next
    Console.WriteLine()
    End Sub
End Module
```

```
private static void DemonstrateDataView()
{
// Retrieve a DataTable from the AdventureWorks sample database.
// connectionString is assumed to be a valid connection string.
SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT FirstName, LastName, EmailAddress " +
    "FROM Person.Contact WHERE FirstName LIKE 'Mich%'",
       GetConnectionString());
DataTable table = new DataTable();

adapter.Fill(table);
Console.WriteLine("Original table name: " + table.TableName);
// Print current table values.
PrintTableOrView(table, "Current Values in Table");

// Now create a DataView based on the DataTable.
// Sort and filter the data.
DataView view = table.DefaultView;
view.Sort = "LastName, FirstName";
view.RowFilter = "LastName > 'M'";
PrintTableOrView(view, "Current Values in View");
```

```
PrintTableOrView(view, "Current values in view");

// Create a new DataTable based on the DataView,
// requesting only two columns with distinct values
// in the columns.
DataTable newTable = view.ToTable("UniqueLastNames",
     true, "FirstName", "LastName");
PrintTableOrView(newTable, "Table created from DataView");
Console.WriteLine("New table name: " + newTable.TableName);

Console.WriteLine("Press any key to continue.");
Console.ReadKey();
}

private static void PrintTableOrView(DataView dv, string label)
{
System.IO.StringWriter sw;
string output;
DataTable table = dv.Table;

Console.WriteLine(label);

// Loop through each row in the view.
foreach (DataRowView rowView in dv)
{
    sw = new System.IO.StringWriter();

    // Loop through each column.
    foreach (DataColumn col in table.Columns)
    {
        // Output the value of each column's data.
        sw.Write(rowView[col.ColumnName].ToString() + ", ");
    }
    output = sw.ToString();
    // Trim off the trailing ", ", so the output looks correct.
    if (output.Length > 2)
    {
        output = output.Substring(0, output.Length - 2);
    }
    // Display the row in the console window.
    Console.WriteLine(output);
}
Console.WriteLine();
}

private static void PrintTableOrView(DataTable table, string label)
{
System.IO.StringWriter sw;
string output;

Console.WriteLine(label);

// Loop through each row in the table.
foreach (DataRow row in table.Rows)
{
    sw = new System.IO.StringWriter();
    // Loop through each column.
    foreach (DataColumn col in table.Columns)
    {
        // Output the value of each column's data.
        sw.Write(row[col].ToString() + ", ");
    }
    output = sw.ToString();
    // Trim off the trailing ", ", so the output looks correct.
    if (output.Length > 2)
    {
        output = output.Substring(0, output.Length - 2);
    }
    // Display the row in the console window.
    Console.WriteLine(output);
```

```
        Console.WriteLine(output);
    } //
    Console.WriteLine();
    }

}
```

See Also

ToTable
DataViews
ADO.NET Managed Providers and DataSet Developer Center

Using XML in a DataSet

8/31/2018 • 2 minutes to read • Edit Online

With ADO.NET you can fill a DataSet from an XML stream or document. You can use the XML stream or document to supply to the DataSet either data, schema information, or both. The information supplied from the XML stream or document can be combined with existing data or schema information already present in the DataSet.

ADO.NET also allows you to create an XML representation of a DataSet, with or without its schema, in order to transport the DataSet across HTTP for use by another application or XML-enabled platform. In an XML representation of a DataSet, the data is written in XML and the schema, if it is included inline in the representation, is written using the XML Schema definition language (XSD). XML and XML Schema provide a convenient format for transferring the contents of a DataSet to and from remote clients.

In This Section

DiffGrams
Provides details on the DiffGram, an XML format used to read and write the contents of a DataSet.

Loading a DataSet from XML
Discusses different options to consider when loading the contents of a DataSet from an XML document.

Writing DataSet Contents as XML Data
Discusses how to generate the contents of a DataSet as XML data, and the different XML format options you can use.

Loading DataSet Schema Information from XML
Discusses the DataSet methods used to load the schema of a DataSet from XML.

Writing DataSet Schema Information as XSD
Discusses the uses for an XML Schema and how to generate one from a DataSet.

DataSet and XmlDataDocument Synchronization
Discusses the capability available in the .NET Framework of synchronous access to both relational and hierarchical views of a single set of data, and shows how to create a synchronous relationship between a DataSet and an XmlDataDocument.

Nesting DataRelations
Discusses the importance of nested DataRelation objects when representing the contents of a DataSet as XML data, and describes how to create them.

Deriving DataSet Relational Structure from XML Schema (XSD)
Describes the relational structure, or schema, of a DataSet that is created from XML Schema.

Inferring DataSet Relational Structure from XML
Describes the resulting relational structure, or schema, of a DataSet that is created when inferred from XML elements.

Related Sections

ADO.NET Overview
Describes the ADO.NET architecture and components, and how to use them to access existing data sources as well as to manage application data.

See Also

DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

DiffGrams

8/31/2018 • 5 minutes to read • Edit Online

A DiffGram is an XML format that identifies current and original versions of data elements. The DataSet uses the DiffGram format to load and persist its contents, and to serialize its contents for transport across a network connection. When a DataSet is written as a DiffGram, it populates the DiffGram with all the necessary information to accurately recreate the contents, though not the schema, of the DataSet, including column values from both the **Original** and **Current** row versions, row error information, and row order.

When sending and retrieving a DataSet from an XML Web service, the DiffGram format is implicitly used. Additionally, when loading the contents of a DataSet from XML using the **ReadXml** method, or when writing the contents of a DataSet in XML using the **WriteXml** method, you can specify that the contents be read or written as a DiffGram. For more information, see Loading a DataSet from XML and Writing DataSet Contents as XML Data.

While the DiffGram format is primarily used by the .NET Framework as a serialization format for the contents of a DataSet, you can also use DiffGrams to modify data in tables in a Microsoft SQL Server database.

A Diffgram is generated by writing the contents of all tables to a **<diffgram>** element.

To generate a Diffgram

1. Generate a list of Root tables (that is, tables without any parent).
2. For each table and its descendants in the list, write out the current version of all rows in the first Diffgram section.
3. For each table in the DataSet, write out the original version of all rows, if any, in the **<before>** section of the Diffgram.
4. For rows that have errors, write the error content in the **<errors>** section of the Diffgram.

A Diffgram is processed in order from beginning of the XML file to the end.

To process a Diffgram

1. Process the first section of the Diffgram that contains the current version of the rows.
2. Process the second or the **<before>** section that contains the original row version of modified and deleted rows.

   > **NOTE**
   > If a row is marked deleted, the delete operation can delete the row's descendants as well, depending on the `Cascade` property of the current DataSet.

3. Process the **<errors>** section. Set the error information for the specified row and column for each item in this section.

> **NOTE**
> If you set the XmlWriteMode to Diffgram, the content of the target DataSet and the original DataSet may differ.

DiffGram Format

The DiffGram format is divided into three sections: the current data, the original (or "before") data, and an errors section, as shown in the following example.

```
<?xml version="1.0"?>
<diffgr:diffgram
         xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
         xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema">

   <DataInstance>
   </DataInstance>

  <diffgr:before>
  </diffgr:before>

  <diffgr:errors>
  </diffgr:errors>
</diffgr:diffgram>
```

The DiffGram format consists of the following blocks of data:

< *DataInstance* >
The name of this element, ***DataInstance***, is used for explanation purposes in this documentation. A ***DataInstance*** element represents a DataSet or a row of a DataTable. Instead of *DataInstance*, the element would contain the name of the DataSet or DataTable. This block of the DiffGram format contains the current data, whether it has been modified or not. An element, or row, that has been modified is identified with the **diffgr:hasChanges** annotation.

<diffgr:before>
This block of the DiffGram format contains the original version of a row. Elements in this block are matched to elements in the ***DataInstance*** block using the **diffgr:id** annotation.

<diffgr:errors>
This block of the DiffGram format contains error information for a particular row in the ***DataInstance*** block. Elements in this block are matched to elements in the ***DataInstance*** block using the **diffgr:id** annotation.

DiffGram Annotations

DiffGrams use several annotations to relate elements from the different DiffGram blocks that represent different row versions or error information in the DataSet.

The following table describes the DiffGram annotations that are defined in the DiffGram namespace **urn:schemas-microsoft-com:xml-diffgram-v1**.

ANNOTATION	DESCRIPTION
id	Used to pair the elements in the **<diffgr:before>** and **<diffgr:errors>** blocks to elements in the **< *DataInstance* >** block. Values with the **diffgr:id** annotation are in the form *[TableName][RowIdentifier]*. For example: `<Customers diffgr:id="Customers1">`.
parentId	Identifies which element from the **< *DataInstance* >** block is the parent element of the current element. Values with the **diffgr:parentId** annotation are in the form *[TableName][RowIdentifier]*. For example: `<Orders diffgr:parentId="Customers1">`.

ANNOTATION	DESCRIPTION
hasChanges	Identifies a row in the **< *DataInstance* >** block as modified. The **hasChanges** annotation can have one of the following two values: **inserted** Identifies an **Added** row. **modified** Identifies a **Modified** row that contains an **Original** row version in the **<diffgr:before>** block. Note that **Deleted** rows will have an **Original** row version in the **<diffgr:before>** block, but there will be no annotated element in the **< *DataInstance* >** block.
hasErrors	Identifies a row in the **< *DataInstance* >** block with a **RowError**. The error element is placed in the **<diffgr:errors>** block.
Error	Contains the text of the **RowError** for a particular element in the **<diffgr:errors>** block.

The DataSet includes additional annotations when reading or writing its contents as a DiffGram. The following table describes these additional annotations, which are defined in the namespace **urn:schemas-microsoft-com:xml-msdata**.

ANNOTATION	DESCRIPTION
RowOrder	Preserves the row order of the original data and identifies the index of a row in a particular DataTable.
Hidden	Identifies a column as having a **ColumnMapping** property set to **MappingType.Hidden**. The attribute is written in the format **msdata:hidden** *[ColumnName]*="*value*". For example: `<Customers diffgr:id="Customers1" msdata:hiddenContactTitle="Owner">` . Note that hidden columns are only written as a DiffGram attribute if they contain data. Otherwise, they are ignored.

Sample DiffGram

An example of the DiffGram format is shown below. This example shows the result of an update to a row in a table before the changes have been committed. The row with a CustomerID of "ALFKI" has been modified, but not updated. As a result, there is a **Current** row with a **diffgr:id** of "Customers1" in the **< *DataInstance* >** block, and an **Original** row with a **diffgr:id** of "Customers1" in the **<diffgr:before>** block. The row with a CustomerID of "ANATR" includes a **RowError**, so it is annotated with `diffgr:hasErrors="true"` and there is a related element in the **<diffgr:errors>** block.

```
<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata" xmlns:diffgr="urn:schemas-microsoft-
com:xml-diffgram-v1">
  <CustomerDataSet>
    <Customers diffgr:id="Customers1" msdata:rowOrder="0" diffgr:hasChanges="modified">
      <CustomerID>ALFKI</CustomerID>
      <CompanyName>New Company</CompanyName>
    </Customers>
    <Customers diffgr:id="Customers2" msdata:rowOrder="1" diffgram:hasErrors="true">
      <CustomerID>ANATR</CustomerID>
      <CompanyName>Ana Trujillo Emparedados y Helados</CompanyName>
    </Customers>
    <Customers diffgr:id="Customers3" msdata:rowOrder="2">
      <CustomerID>ANTON</CustomerID>
      <CompanyName>Antonio Moreno Taquera</CompanyName>
    </Customers>
    <Customers diffgr:id="Customers4" msdata:rowOrder="3">
      <CustomerID>AROUT</CustomerID>
      <CompanyName>Around the Horn</CompanyName>
    </Customers>
  </CustomerDataSet>
  <diffgr:before>
    <Customers diffgr:id="Customers1" msdata:rowOrder="0">
      <CustomerID>ALFKI</CustomerID>
      <CompanyName>Alfreds Futterkiste</CompanyName>
    </Customers>
  </diffgr:before>
  <diffgr:errors>
    <Customers diffgr:id="Customers2" diffgr:Error="An optimistic concurrency violation has occurred for this
row."/>
  </diffgr:errors>
</diffgr:diffgram>
```

See Also

Using XML in a DataSet
Loading a DataSet from XML
Writing DataSet Contents as XML Data
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Loading a DataSet from XML

8/31/2018 • 5 minutes to read • Edit Online

The contents of an ADO.NET DataSet can be created from an XML stream or document. In addition, with the .NET Framework you have great flexibility over what information is loaded from XML, and how the schema or relational structure of the DataSet is created.

To fill a DataSet with data from XML, use the **ReadXml** method of the DataSet object. The **ReadXml** method reads from a file, a stream, or an **XmlReader**, and takes as arguments the source of the XML plus an optional **XmlReadMode** argument. (For more information about the **XmlReader**, see NIB: Reading XML Data with XmlTextReader.) The **ReadXml** method reads the contents of the XML stream or document and loads the DataSet with data. It will also create the relational schema of the DataSet depending on the **XmlReadMode** specified and whether or not a relational schema already exists.

The following table describes the options for the **XmlReadMode** argument.

OPTION	DESCRIPTION
Auto	This is the default. Examines the XML and chooses the most appropriate option in the following order: - If the XML is a DiffGram, **DiffGram** is used. - If the DataSet contains a schema or the XML contains an inline schema, **ReadSchema** is used. - If the DataSet does not contain a schema and the XML does not contain an inline schema, **InferSchema** is used. If you know the format of the XML being read, for best performance it is recommended that you set an explicit **XmlReadMode**, rather than accept the **Auto** default.
ReadSchema	Reads any inline schema and loads the data and schema. If the DataSet already contains a schema, new tables are added from the inline schema to the existing schema in the DataSet. If any tables in the inline schema already exist in the DataSet, an exception is thrown. You will not be able to modify the schema of an existing table using **XmlReadMode.ReadSchema**. If the DataSet does not contain a schema, and there is no inline schema, no data is read. Inline schema can be defined using XML Schema definition language (XSD) schema. For details about writing inline schema as XML Schema, see Deriving DataSet Relational Structure from XML Schema (XSD).
IgnoreSchema	Ignores any inline schema and loads the data into the existing DataSet schema. Any data that does not match the existing schema is discarded. If no schema exists in the DataSet, no data is loaded. If the data is a DiffGram, **IgnoreSchema** has the same functionality as **DiffGram** .

OPTION	DESCRIPTION
InferSchema	Ignores any inline schema and infers the schema per the structure of the XML data, then loads the data. If the DataSet already contains a schema, the current schema is extended by adding columns to existing tables. Extra tables will not be added if there are not existing tables. An exception is thrown if an inferred table already exists with a different namespace, or if any inferred columns conflict with existing columns. For details about how **ReadXmlSchema** infers a schema from an XML document, see Inferring DataSet Relational Structure from XML.
DiffGram	Reads a DiffGram and adds the data to the current schema. **DiffGram** merges new rows with existing rows where the unique identifier values match. See "Merging Data from XML" at the end of this topic. For more information about DiffGrams, see DiffGrams.
Fragment	Continues reading multiple XML fragments until the end of the stream is reached. Fragments that match the DataSet schema are appended to the appropriate tables. Fragments that do not match the DataSet schema are discarded.

> **NOTE**
> If you pass an **XmlReader** to **ReadXml** that is positioned part of the way into an XML document, **ReadXml** will read to the next element node and will treat that as the root element, reading until the end of the element node only. This does not apply if you specify **XmlReadMode.Fragment**.

DTD Entities

If your XML contains entities defined in a document type definition (DTD) schema, an exception will be thrown if you attempt to load a DataSet by passing a file name, stream, or non-validating **XmlReader** to **ReadXml**. Instead, you must create an **XmlValidatingReader**, with **EntityHandling** set to **EntityHandling.ExpandEntities**, and pass your **XmlValidatingReader** to **ReadXml**. The **XmlValidatingReader** will expand the entities prior to being read by the DataSet.

The following code examples show how to load a DataSet from an XML stream. The first example shows a file name being passed to the **ReadXml** method. The second example shows a string that contains XML being loaded using a StringReader.

```
Dim dataSet As DataSet = New DataSet
dataSet.ReadXml("input.xml", XmlReadMode.ReadSchema)
```

```
DataSet dataSet = new DataSet();
dataSet.ReadXml("input.xml", XmlReadMode.ReadSchema);
```

```
Dim dataSet As DataSet = New DataSet
Dim dataTable As DataTable = New DataTable("table1")
dataTable.Columns.Add("col1", Type.GetType("System.String"))
dataSet.Tables.Add(dataTable)

Dim xmlData As String = "<XmlDS><table1><col1>Value1</col1></table1><table1><col1>Value2</col1></table1>
</XmlDS>"

Dim xmlSR As System.IO.StringReader = New System.IO.StringReader(xmlData)

dataSet.ReadXml(xmlSR, XmlReadMode.IgnoreSchema)
```

```
DataSet dataSet = new DataSet();
DataTable dataTable = new DataTable("table1");
dataTable.Columns.Add("col1", typeof(string));
dataSet.Tables.Add(dataTable);

string xmlData = "<XmlDS><table1><col1>Value1</col1></table1><table1><col1>Value2</col1></table1></XmlDS>";

System.IO.StringReader xmlSR = new System.IO.StringReader(xmlData);

dataSet.ReadXml(xmlSR, XmlReadMode.IgnoreSchema);
```

NOTE

If you call **ReadXml** to load a very large file, you may encounter slow performance. To ensure best performance for **ReadXml**, on a large file, call the BeginLoadData method for each table in the DataSet, and then call **ReadXml**. Finally, call EndLoadData for each table in the DataSet, as shown in the following example.

```
Dim dataTable As DataTable

For Each dataTable In dataSet.Tables
   dataTable.BeginLoadData()
Next

dataSet.ReadXml("file.xml")

For Each dataTable in dataSet.Tables
   dataTable.EndLoadData()
Next
```

```
foreach (DataTable dataTable in dataSet.Tables)
   dataTable.BeginLoadData();

dataSet.ReadXml("file.xml");

foreach (DataTable dataTable in dataSet.Tables)
   dataTable.EndLoadData();
```

NOTE

If the XSD schema for your DataSet includes a **targetNamespace**, data may not be read, and you may encounter exceptions, when calling **ReadXml** to load the DataSet with XML that contains elements with no qualifying namespace. To read unqualified elements in this case, set **elementFormDefault** equal to "qualified" in your XSD schema. For example:

```
<xsd:schema id="customDataSet"
  elementFormDefault="qualified"
  targetNamespace="http://www.tempuri.org/customDataSet.xsd"
  xmlns="http://www.tempuri.org/customDataSet.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
</xsd:schema>
```

Merging Data from XML

If the DataSet already contains data, the new data from the XML is added to the data already present in the DataSet. **ReadXml** does not merge from the XML into the DataSet any row information with matching primary keys. To overwrite existing row information with new information from XML, use **ReadXml** to create a new DataSet, and then Merge the new DataSet into the existing DataSet. Note that loading a DiffGram using **ReadXML** with an **XmlReadMode** of **DiffGram** will merge rows that have the same unique identifier.

See Also

DataSet.Merge
Using XML in a DataSet
DiffGrams
Deriving DataSet Relational Structure from XML Schema (XSD)
Inferring DataSet Relational Structure from XML
Loading DataSet Schema Information from XML
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Writing DataSet Contents as XML Data

8/31/2018 • 3 minutes to read • Edit Online

In ADO.NET you can write an XML representation of a DataSet, with or without its schema. If schema information is included inline with the XML, it is written using the XML Schema definition language (XSD). The schema contains the table definitions of the DataSet as well as the relation and constraint definitions.

When a DataSet is written as XML data, the rows in the DataSet are written in their current versions. However, the DataSet can also be written as a DiffGram so that both the current and the original values of the rows will be included.

The XML representation of the DataSet can be written to a file, a stream, an **XmlWriter**, or a string. These choices provide great flexibility for how you transport the XML representation of the DataSet. To obtain the XML representation of the DataSet as a string, use the **GetXml** method as shown in the following example.

```
Dim xmlDS As String = custDS.GetXml()
```

```
string xmlDS = custDS.GetXml();
```

GetXml returns the XML representation of the DataSet without schema information. To write the schema information from the DataSet (as XML Schema) to a string, use **GetXmlSchema**.

To write a DataSet to a file, stream, or **XmlWriter**, use the **WriteXml** method. The first parameter you pass to **WriteXml** is the destination of the XML output. For example, pass a string containing a file name, a **System.IO.TextWriter** object, and so on. You can pass an optional second parameter of an **XmlWriteMode** to specify how the XML output is to be written.

The following table shows the options for **XmlWriteMode**.

XMLWRITEMODE OPTION	DESCRIPTION
IgnoreSchema	Writes the current contents of the DataSet as XML data, without an XML Schema. This is the default.
WriteSchema	Writes the current contents of the DataSet as XML data with the relational structure as inline XML Schema.
DiffGram	Writes the entire DataSet as a DiffGram, including original and current values. For more information, see DiffGrams.

When writing an XML representation of a DataSet that contains **DataRelation** objects, you will most likely want the resulting XML to have the child rows of each relation nested within their related parent elements. To accomplish this, set the **Nested** property of the **DataRelation** to **true** when you add the **DataRelation** to the DataSet. For more information, see Nesting DataRelations.

Following are two examples of how to write the XML representation of a DataSet to a file. The first example passes the file name for the resulting XML as a string to **WriteXml**. The second example passes a **System.IO.StreamWriter** object.

```
custDS.WriteXml("Customers.xml", XmlWriteMode.WriteSchema)
```

```
custDS.WriteXml("Customers.xml", XmlWriteMode.WriteSchema);
```

```
Dim xmlSW As System.IO.StreamWriter = New System.IO.StreamWriter("Customers.xml")
custDS.WriteXml(xmlSW, XmlWriteMode.WriteSchema)
xmlSW.Close()
```

```
System.IO.StreamWriter xmlSW = new System.IO.StreamWriter("Customers.xml");
custDS.WriteXml(xmlSW, XmlWriteMode.WriteSchema);
xmlSW.Close();
```

Mapping Columns to XML Elements, Attributes, and Text

You can specify how a column of a table is represented in XML using the **ColumnMapping** property of the **DataColumn** object. The following table shows the different **MappingType** values for the **ColumnMapping** property of a table column, and the resulting XML.

MAPPINGTYPE VALUE	DESCRIPTION
Element	This is the default. The column is written as an XML element where the ColumnName is the name of the element and the contents of the column are written as the text of the element. For example: `<ColumnName>Column Contents</ColumnName>`
Attribute	The column is written as an XML attribute of the XML element for the current row where the ColumnName is the name of the attribute and the contents of the column are written as the value of the attribute. For example: `<RowElement ColumnName="Column Contents" />`
SimpleContent	The contents of the column are written as text in the XML element for the current row. For example: `<RowElement>Column Contents</RowElement>` Note that **SimpleContent** cannot be set for a column of a table that has **Element** columns or nested relations.
Hidden	The column is not written in the XML output.

See Also

Using XML in a DataSet
DiffGrams
Nesting DataRelations
Writing DataSet Schema Information as XSD
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Loading DataSet Schema Information from XML

8/31/2018 • 4 minutes to read • Edit Online

The schema of a DataSet (its tables, columns, relations, and constraints) can be defined programmatically, created by the **Fill** or **FillSchema** methods of a DataAdapter, or loaded from an XML document. To load **DataSet** schema information from an XML document, you can use either the **ReadXmlSchema** or the **InferXmlSchema** method of the **DataSet**. **ReadXmlSchema** allows you to load or infer **DataSet** schema information from the document containing XML Schema definition language (XSD) schema, or an XML document with inline XML Schema. **InferXmlSchema** allows you to infer the schema from the XML document while ignoring certain XML namespaces that you specify.

> **NOTE**
>
> Table ordering in a **DataSet** might not be preserved when you use Web services or XML serialization to transfer a **DataSet** that was created in-memory by using XSD constructs (such as nested relations). Therefore, the recipient of the **DataSet** should not depend on table ordering in this case. However, table ordering is always preserved if the schema of the **DataSet** being transferred was read from XSD files, instead of being created in-memory.

ReadXmlSchema

To load the schema of a **DataSet** from an XML document without loading any data, you can use the **ReadXmlSchema** method of the **DataSet**. **ReadXmlSchema** creates **DataSet** schema defined using XML Schema definition language (XSD) schema.

The **ReadXmlSchema** method takes a single argument of a file name, a stream, or an **XmlReader** containing the XML document to be loaded. The XML document can contain only schema, or can contain schema inline with XML elements containing data. For details about writing inline schema as XML Schema, see Deriving DataSet Relational Structure from XML Schema (XSD).

If the XML document passed to **ReadXmlSchema** contains no inline schema information, **ReadXmlSchema** will infer the schema from the elements in the XML document. If the **DataSet** already contains a schema, the current schema will be extended by adding new tables if they do not already exist. New columns will not be added to added to existing tables. If a column being added already exists in the **DataSet** but has an incompatible type with the column found in the XML, an exception is thrown. For details about how **ReadXmlSchema** infers a schema from an XML document, see Inferring DataSet Relational Structure from XML.

Although **ReadXmlSchema** loads or infers only the schema of a **DataSet**, the **ReadXml** method of the **DataSet** loads or infers both the schema and the data contained in the XML document. For more information, see Loading a DataSet from XML.

The following code examples show how to load a **DataSet** schema from an XML document or stream. The first example shows an XML Schema file name being passed to the **ReadXmlSchema** method. The second example shows a **System.IO.StreamReader**.

```
Dim dataSet As DataSet = New DataSet
dataSet.ReadXmlSchema("schema.xsd")
```

```
DataSet dataSet = new DataSet();
dataSet.ReadXmlSchema("schema.xsd");
```

```
Dim xmlStream As System.IO.StreamReader = New System.IO.StreamReader ("schema.xsd");
Dim dataSet As DataSet = New DataSet
dataSet.ReadXmlSchema(xmlStream)
xmlStream.Close()
```

```
System.IO.StreamReader xmlStream = new System.IO.StreamReader("schema.xsd");
DataSet dataSet = new DataSet();
dataSet.ReadXmlSchema(xmlStream);
xmlStream.Close();
```

InferXmlSchema

You can also instruct the **DataSet** to infer its schema from an XML document using the **InferXmlSchema** method of the **DataSet**. **InferXmlSchema** functions the same as do both **ReadXml** with an **XmlReadMode** of **InferSchema** (loads data as well as infers schema), and **ReadXmlSchema** if the document being read contains no inline schema. However, **InferXmlSchema** provides the additional capability of allowing you to specify particular XML namespaces to be ignored when the schema is inferred. **InferXmlSchema** takes two required arguments: the location of the XML document, specified by a file name, a stream, or an **XmlReader**; and a string array of XML namespaces to be ignored by the operation.

For example, consider the following XML:

```
<NewDataSet xmlns:od="urn:schemas-microsoft-com:officedata">
<Categories>
  <CategoryID od:adotype="3">1</CategoryID>
  <CategoryName od:maxLength="15" od:adotype="130">Beverages</CategoryName>
  <Description od:adotype="203">Soft drinks and teas</Description>
</Categories>
<Products>
  <ProductID od:adotype="20">1</ProductID>
  <ReorderLevel od:adotype="3">10</ReorderLevel>
  <Discontinued od:adotype="11">0</Discontinued>
</Products>
</NewDataSet>
```

Because of the attributes specified for the elements in the preceding XML document, both the **ReadXmlSchema** method and the **ReadXml** method with an **XmlReadMode** of **InferSchema** would create tables for every element in the document: **Categories**, **CategoryID**, **CategoryName**, **Description**, **Products**, **ProductID**, **ReorderLevel**, and **Discontinued**. (For more information, see Inferring DataSet Relational Structure from XML.) However, a more appropriate structure would be to create only the **Categories** and **Products** tables, and then to create **CategoryID**, **CategoryName**, and **Description** columns in the **Categories** table, and **ProductID**, **ReorderLevel**, and **Discontinued** columns in the **Products** table. To ensure that the inferred schema ignores the attributes specified in the XML elements, use the **InferXmlSchema** method and specify the XML namespace for **officedata** to be ignored, as shown in the following example.

```
Dim dataSet As DataSet = New DataSet
dataSet.InferXmlSchema("input_od.xml", New String() {"urn:schemas-microsoft-com:officedata"})
```

```
DataSet dataSet = new DataSet();
dataSet.InferXmlSchema("input_od.xml", new string[] "urn:schemas-microsoft-com:officedata");
```

See Also

Using XML in a DataSet
Deriving DataSet Relational Structure from XML Schema (XSD)
Inferring DataSet Relational Structure from XML
Loading a DataSet from XML
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Writing DataSet Schema Information as XSD

8/31/2018 • 2 minutes to read • Edit Online

You can write the schema of a DataSet as XML Schema definition language (XSD) schema, so that you can transport it, with or without related data, in an XML document. XML Schema can be written to a file, a stream, an XmlWriter, or a string; it is useful for generating a strongly typed **DataSet**. For more information about strongly typed **DataSet** objects, see Typed DataSets.

You can specify how a column of a table is represented in XML Schema using the **ColumnMapping** property of the DataColumn object. For more information, see "Mapping Columns to XML Elements, Attributes, and Text" in Writing DataSet Contents as XML Data.

To write the schema of a **DataSet** as XML Schema, to a file, stream, or **XmlWriter**, use the **WriteXmlSchema** method of the **DataSet**. **WriteXmlSchema** takes one parameter that specifies the destination of the resulting XML Schema. The following code examples demonstrate how to write the XML Schema of a **DataSet** to a file by passing a string containing a file name and a StreamWriter object.

```
dataSet.WriteXmlSchema("Customers.xsd")
```

```
dataSet.WriteXmlSchema("Customers.xsd");
```

```
Dim writer As System.IO.StreamWriter = New System.IO.StreamWriter("Customers.xsd")
dataSet.WriteXmlSchema(writer)
writer.Close()
```

```
System.IO.StreamWriter writer = new System.IO.StreamWriter("Customers.xsd");
dataSet.WriteXmlSchema(writer);
writer.Close();
```

To obtain the schema of a **DataSet** and write it as an XML Schema string, use the **GetXmlSchema** method, as shown in the following example.

```
Dim schemaString As String = dataSet.GetXmlSchema()
```

```
string schemaString = dataSet.GetXmlSchema();
```

See Also

Using XML in a DataSet
Writing DataSet Contents as XML Data
Typed DataSets
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

DataSet and XmlDataDocument Synchronization

8/31/2018 • 4 minutes to read • Edit Online

The ADO.NET DataSet provides you with a relational representation of data. For hierarchical data access, you can use the XML classes available in the .NET Framework. Historically, these two representations of data have been used separately. However, the .NET Framework enables real-time, synchronous access to both the relational and hierarchical representations of data through the **DataSet** object and the XmlDataDocument object, respectively.

When a **DataSet** is synchronized with an **XmlDataDocument**, both objects are working with a single set of data. This means that if a change is made to the **DataSet**, the change will be reflected in the **XmlDataDocument**, and vice versa. The relationship between the **DataSet** and the **XmlDataDocument** creates great flexibility by allowing a single application, using a single set of data, to access the entire suite of services built around the **DataSet** (such as Web Forms and Windows Forms controls, and Visual Studio .NET designers), as well as the suite of XML services including Extensible Stylesheet Language (XSL), XSL Transformations (XSLT), and XML Path Language (XPath). You do not have to choose which set of services to target with the application; both are available.

There are several ways that you can synchronize a **DataSet** with an **XmlDataDocument**. You can:

- Populate a **DataSet** with schema (that is, a relational structure) and data and then synchronize it with a new **XmlDataDocument**. This provides a hierarchical view of existing relational data. For example:

```
Dim dataSet As DataSet = New DataSet

' Add code here to populate the DataSet with schema and data.

Dim xmlDoc As XmlDataDocument = New XmlDataDocument(dataSet)
```

```
DataSet dataSet = new DataSet();

// Add code here to populate the DataSet with schema and data.

XmlDataDocument xmlDoc = new XmlDataDocument(dataSet);
```

- Populate a **DataSet** with schema only (such as a strongly typed **DataSet**), synchronize it with an **XmlDataDocument**, and then load the **XmlDataDocument** from an XML document. This provides a relational view of existing hierarchical data. The table names and column names in your **DataSet** schema must match the names of the XML elements that you want them synchronized with. This matching is case-sensitive.

  Note that the schema of the **DataSet** only needs to match the XML elements that you want to expose in your relational view. This way, you can have a very large XML document and a very small relational "window" on that document. The **XmlDataDocument** preserves the entire XML document even though the **DataSet** only exposes a small portion of it. (For a detailed example of this, see Synchronizing a DataSet with an XmlDataDocument.)

  The following code example shows the steps for creating a **DataSet** and populating its schema, then synchronizing it with an **XmlDataDocument**. Note that the **DataSet** schema only needs to match the elements from the **XmlDataDocument** that you want to expose using the **DataSet**.

```
Dim dataSet As DataSet = New DataSet

' Add code here to populate the DataSet with schema, but not data.

Dim xmlDoc As XmlDataDocument = New XmlDataDocument(dataSet)
xmlDoc.Load("XMLDocument.xml")
```

```
DataSet dataSet = new DataSet();

// Add code here to populate the DataSet with schema, but not data.

XmlDataDocument xmlDoc = new XmlDataDocument(dataSet);
xmlDoc.Load("XMLDocument.xml");
```

You cannot load an **XmlDataDocument** if it is synchronized with a **DataSet** that contains data. An exception will be thrown.

- Create a new **XmlDataDocument** and load it from an XML document, and then access the relational view of the data using the **DataSet** property of the **XmlDataDocument**. You need to set the schema of the **DataSet** before you can view any of the data in the **XmlDataDocument** using the **DataSet**. Again, the table names and column names in your **DataSet** schema must match the names of the XML elements that you want them synchronized with. This matching is case-sensitive.

  The following code example shows how to access the relational view of the data in an **XmlDataDocument**.

```
Dim xmlDoc As XmlDataDocument = New XmlDataDocument
Dim dataSet As DataSet = xmlDoc.DataSet

' Add code here to create the schema of the DataSet to view the data.

xmlDoc.Load("XMLDocument.xml")
```

```
XmlDataDocument xmlDoc = new XmlDataDocument();
DataSet dataSet = xmlDoc.DataSet;

// Add code here to create the schema of the DataSet to view the data.

xmlDoc.Load("XMLDocument.xml");
```

Another advantage of synchronizing an **XmlDataDocument** with a **DataSet** is that the fidelity of an XML document is preserved. If the **DataSet** is populated from an XML document using **ReadXml**, when the data is written back as an XML document using **WriteXml** it may differ dramatically from the original XML document. This is because the **DataSet** does not maintain formatting, such as white space, or hierarchical information, such as element order, from the XML document. The **DataSet** also does not contain elements from the XML document that were ignored because they did not match the schema of the **Dataset**. Synchronizing an **XmlDataDocument** with a **DataSet** allows the formatting and hierarchical element structure of the original XML document to be maintained in the **XmlDataDocument**, while the **DataSet** contains only data and schema information appropriate to the **DataSet**.

When synchronizing a **DataSet** with an **XmlDataDocument**, results may differ depending on whether or not your DataRelation objects are nested. For more information, see Nesting DataRelations.

In This Section

Synchronizing a DataSet with an XmlDataDocument

Demonstrates synchronizing a strongly typed **DataSet**, with minimal schema, with an **XmlDataDocument**.

Performing an XPath Query on a DataSet
Demonstrates performing an XPath query on the contents of a **DataSet**.

Applying an XSLT Transform to a DataSet
Demonstrates applying an XSLT transform to the contents of a **DataSet**.

Related Sections

Using XML in a DataSet
Describes how the **DataSet** interacts with XML as a data source, including loading and persisting the contents of a **DataSet** as XML data.

Nesting DataRelations
Discusses the importance of nested **DataRelation** objects when representing the contents of a **DataSet** as XML data, and describes how to create these relations.

DataSets, DataTables, and DataViews
Describes the **DataSet** and how to use it to manage application data and to interact with data sources including relational databases and XML.

XmlDataDocument
Contains reference information about the **XmlDataDocument** class.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Synchronizing a DataSet with an XmlDataDocument

8/31/2018 • 4 minutes to read • Edit Online

This section demonstrates one step in the processing of a purchase order, using a strongly typed DataSet synchronized with an XmlDataDocument. The examples that follow create a **DataSet** with a minimized schema that matches only a portion of the source XML document. The examples use an **XmlDataDocument** to preserve the fidelity of the source XML document, enabling the **DataSet** to be used to expose a subset of the XML document.

The following XML document contains all the information pertaining to a purchase order: customer information, items ordered, shipping information, and so on.

```
<?xml version="1.0" standalone="yes"?>
<PurchaseOrder>
  <Customers>
    <CustomerID>CHOPS</CustomerID>
    <Orders>
      <OrderID>10966</OrderID>
      <OrderDetails>
        <OrderID>10966</OrderID>
        <ProductID>37</ProductID>
        <UnitPrice>26</UnitPrice>
        <Quantity>8</Quantity>
        <Discount>0</Discount>
      </OrderDetails>
      <OrderDetails>
        <OrderID>10966</OrderID>
        <ProductID>56</ProductID>
        <UnitPrice>38</UnitPrice>
        <Quantity>12</Quantity>
        <Discount>0.15</Discount>
      </OrderDetails>
      <OrderDetails>
        <OrderID>10966</OrderID>
        <ProductID>62</ProductID>
        <UnitPrice>49.3</UnitPrice>
        <Quantity>12</Quantity>
        <Discount>0.15</Discount>
      </OrderDetails>
      <CustomerID>CHOPS</CustomerID>
      <EmployeeID>4</EmployeeID>
      <OrderDate>1998-03-20T00:00:00.0000000</OrderDate>
      <RequiredDate>1998-04-17T00:00:00.0000000</RequiredDate>
      <ShippedDate>1998-04-08T00:00:00.0000000</ShippedDate>
      <ShipVia>1</ShipVia>
      <Freight>27.19</Freight>
      <ShipName>Chop-suey Chinese</ShipName>
      <ShipAddress>Hauptstr. 31</ShipAddress>
      <ShipCity>Bern</ShipCity>
      <ShipPostalCode>3012</ShipPostalCode>
      <ShipCountry>Switzerland</ShipCountry>
    </Orders>
    <CompanyName>Chop-suey Chinese</CompanyName>
    <ContactName>Yang Wang</ContactName>
    <ContactTitle>Owner</ContactTitle>
    <Address>Hauptstr. 29</Address>
    <City>Bern</City>
    <PostalCode>3012</PostalCode>
    <Country>Switzerland</Country>
    <Phone>0452-076545</Phone>
  </Customers>
```

```
  </Customers>
  <Shippers>
    <ShipperID>1</ShipperID>
    <CompanyName>Speedy Express</CompanyName>
    <Phone>(503) 555-0100</Phone>
  </Shippers>
  <Shippers>
    <ShipperID>2</ShipperID>
    <CompanyName>United Package</CompanyName>
    <Phone>(503) 555-0101</Phone>
  </Shippers>
  <Shippers>
    <ShipperID>3</ShipperID>
    <CompanyName>Federal Shipping</CompanyName>
    <Phone>(503) 555-0102</Phone>
  </Shippers>
  <Products>
    <ProductID>37</ProductID>
    <ProductName>Gravad lax</ProductName>
    <QuantityPerUnit>12 - 500 g pkgs.</QuantityPerUnit>
    <UnitsInStock>11</UnitsInStock>
    <UnitsOnOrder>50</UnitsOnOrder>
    <ReorderLevel>25</ReorderLevel>
  </Products>
  <Products>
    <ProductID>56</ProductID>
    <ProductName>Gnocchi di nonna Alice</ProductName>
    <QuantityPerUnit>24 - 250 g pkgs.</QuantityPerUnit>
    <UnitsInStock>21</UnitsInStock>
    <UnitsOnOrder>10</UnitsOnOrder>
    <ReorderLevel>30</ReorderLevel>
  </Products>
  <Products>
    <ProductID>62</ProductID>
    <ProductName>Tarte au sucre</ProductName>
    <QuantityPerUnit>48 pies</QuantityPerUnit>
    <UnitsInStock>17</UnitsInStock>
    <UnitsOnOrder>0</UnitsOnOrder>
    <ReorderLevel>0</ReorderLevel>
  </Products>
</PurchaseOrder>
```

One step in processing the purchase order information contained in the preceding XML document is for the order to be filled from the company's current inventory. The employee responsible for filling the order from the company's warehouse does not need to see the entire contents of the purchase order; they only need to see the product information for the order. To expose only the product information from the XML document, create a strongly typed **DataSet** with a schema, written as XML Schema definition language (XSD) schema, that maps to the products and quantities ordered. For more information about strongly typed **DataSet** objects, see Typed DataSets.

The following code shows the schema from which the strongly typed **DataSet** is generated for this sample.

```
<?xml version="1.0" standalone="yes"?>
<xs:schema id="OrderDetail" xmlns=""
                            xmlns:xs="http://www.w3.org/2001/XMLSchema"
                            xmlns:codegen="urn:schemas-microsoft-com:xml-msprop"
                            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="OrderDetail" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="OrderDetails" codegen:typedName="LineItem" codegen:typedPlural="LineItems">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="OrderID" type="xs:int" minOccurs="0" codegen:typedName="OrderID"/>
              <xs:element name="Quantity" type="xs:short" minOccurs="0" codegen:typedName="Quantity"/>
              <xs:element name="ProductID" type="xs:int" minOccurs="0" codegen:typedName="ProductID"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Products" codegen:typedName="Product" codegen:typedPlural="Products">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ProductID" type="xs:int" minOccurs="0" codegen:typedName="ProductID"/>
              <xs:element name="ProductName" type="xs:string" minOccurs="0" codegen:typedName="ProductName"/>
              <xs:element name="QuantityPerUnit" type="xs:string" minOccurs="0"
codegen:typedName="QuantityPerUnit"/>
              <xs:element name="UnitsInStock" type="xs:short" minOccurs="0" codegen:typedName="UnitsInStock"/>
              <xs:element name="UnitsOnOrder" type="xs:short" minOccurs="0" codegen:typedName="UnitsOnOrder"/>
              <xs:element name="ReorderLevel" type="xs:short" minOccurs="0" codegen:typedName="ReorderLevel"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
    <xs:unique name="Constraint1">
      <xs:selector xpath=".//Products" />
      <xs:field xpath="ProductID" />
    </xs:unique>
    <xs:keyref name="Relation1" refer="Constraint1" codegen:typedChildren="GetLineItems"
codegen:typedParent="Product">
      <xs:selector xpath=".//OrderDetails" />
      <xs:field xpath="ProductID" />
    </xs:keyref>
  </xs:element>
</xs:schema>
```

Notice that only information from the **OrderDetails** and **Products** elements of the original XML document are included in the schema for the **DataSet**. Synchronizing the **DataSet** with an **XmlDataDocument** ensures that the elements not included in the **DataSet** will persist with the XML document.

With the strongly typed **DataSet** generated from the XML Schema (with a namespace of **Northwind.FillOrder**), a portion of the original XML document can be exposed by synchronizing the **DataSet** with the **XmlDataDocument** loaded from the source XML document. Notice that the **DataSet** generated from the schema contains structure but no data. The data is filled in when you load the XML into the **XmlDataDocument**. If you attempt to load an **XmlDataDocument** that has been synchronized with a **DataSet** that already contains data, an exception will be thrown.

After the **DataSet** (and the **XmlDataDocument**) has been updated, the **XmlDataDocument** can then write out the modified XML document with the elements ignored by the **DataSet** still intact, as shown below. In the purchase order scenario, after the order items have been filled, the modified XML document can then be passed on to the next step in the order process, perhaps to the company's shipping department.

```
Imports System
Imports System.Data
Imports System.Xml
Imports Northwind.FillOrder

Public class Sample
  Public Shared Sub Main()

    Dim orderDS As OrderDetail = New OrderDetail

    Dim xmlDocument As XmlDataDocument = New XmlDataDocument(orderDS)

    xmlDocument.Load("Order.xml")

    Dim orderItem As OrderDetail.LineItem
    Dim product As OrderDetail.Product

    For Each orderItem In orderDS.LineItems
      product = orderItem.Product

      ' Remove quantity from the current stock.
      product.UnitsInStock = CType(product.UnitsInStock - orderItem.Quantity, Short)

      ' If the remaining stock is less than the reorder level, order more.
      If ((product.UnitsInStock + product.UnitsOnOrder) < product.ReorderLevel) Then
        product.UnitsOnOrder = CType(product.UnitsOnOrder + product.ReorderLevel, Short)
      End If
    Next

    xmlDocument.Save("Order_out.xml")
  End Sub
End Class
```

```
using System;
using System.Data;
using System.Xml;
using Northwind.FillOrder;

public class Sample
{
  public static void Main()
  {
    OrderDetail orderDS = new OrderDetail();

    XmlDataDocument xmlDocument = new XmlDataDocument(orderDS);

    xmlDocument.Load("Order.xml");

    foreach (OrderDetail.LineItem orderItem in orderDS.LineItems)
    {
      OrderDetail.Product product = orderItem.Product;

      // Remove quantity from the current stock.
      product.UnitsInStock = (short)(product.UnitsInStock - orderItem.Quantity);

      // If the remaining stock is less than the reorder level, order more.
      if ((product.UnitsInStock + product.UnitsOnOrder) < product.ReorderLevel)
        product.UnitsOnOrder = (short)(product.UnitsOnOrder + product.ReorderLevel);
    }

    xmlDocument.Save("Order_out.xml");
  }
}
```

See Also

DataSet and XmlDataDocument Synchronization
ADO.NET Managed Providers and DataSet Developer Center

Performing an XPath Query on a DataSet

8/31/2018 • 2 minutes to read • Edit Online

The relationship between a synchronized DataSet and XmlDataDocument allows you to make use of XML services, such as the XML Path Language (XPath) query, that access the **XmlDataDocument** and can perform certain functionality more conveniently than accessing the **DataSet** directly. For example, rather than using the **Select** method of a DataTable to navigate relationships to other tables in a **DataSet**, you can perform an XPath query on an **XmlDataDocument** that is synchronized with the **DataSet**, to get a list of XML elements in the form of an XmlNodeList. The nodes in the **XmlNodeList**, cast as XmlElement nodes, can then be passed to the **GetRowFromElement** method of the **XmlDataDocument**, to return matching DataRow references to the rows of the table in the synchronized **DataSet**.

For example, the following code sample performs a "grandchild" XPath query. The **DataSet** is filled with three tables: **Customers**, **Orders**, and **OrderDetails**. In the sample, a parent-child relation is first created between the **Customers** and **Orders** tables, and between the **Orders** and **OrderDetails** tables. An XPath query is then performed to return an **XmlNodeList** of **Customers** nodes where a grandchild **OrderDetails** node has a **ProductID** node with the value of 43. In essence, the sample is using the XPath query to determine which customers have ordered the product that has the **ProductID** of 43.

```
' Assumes that connection is a valid SqlConnection.
connection.Open()
Dim dataSet As DataSet = New DataSet("CustomerOrders")
Dim customerAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM Customers", connection)
customerAdapter.Fill(dataSet, "Customers")

Dim orderAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM Orders", connection)
orderAdapter.Fill(dataSet, "Orders")

Dim detailAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM [Order Details]", connection)
detailAdapter.Fill(dataSet, "OrderDetails")

connection.Close()

dataSet.Relations.Add("CustOrders", _
dataSet.Tables("Customers").Columns("CustomerID"), _
dataSet.Tables("Orders").Columns("CustomerID")).Nested = true

dataSet.Relations.Add("OrderDetail", _
  dataSet.Tables("Orders").Columns("OrderID"), _
dataSet.Tables("OrderDetails").Columns("OrderID"), false).Nested = true

Dim xmlDoc As XmlDataDocument = New XmlDataDocument(dataSet)

Dim nodeList As XmlNodeList = xmlDoc.DocumentElement.SelectNodes( _
  "descendant::Customers[*/OrderDetails/ProductID=43]")

Dim dataRow As DataRow
Dim xmlNode As XmlNode

For Each xmlNode In nodeList
  dataRow = xmlDoc.GetRowFromElement(CType(xmlNode, XmlElement))

  If Not dataRow Is Nothing then Console.WriteLine(xmlRow(0).ToString())
Next
```

```
// Assumes that connection is a valid SqlConnection.
connection.Open();

DataSet dataSet = new DataSet("CustomerOrders");

SqlDataAdapter customerAdapter = new SqlDataAdapter(
  "SELECT * FROM Customers", connection);
customerAdapter.Fill(dataSet, "Customers");

SqlDataAdapter orderAdapter = new SqlDataAdapter(
  "SELECT * FROM Orders", connection);
orderAdapter.Fill(dataSet, "Orders");

SqlDataAdapter detailAdapter = new SqlDataAdapter(
  "SELECT * FROM [Order Details]", connection);
detailAdapter.Fill(dataSet, "OrderDetails");

connection.Close();

dataSet.Relations.Add("CustOrders",
  dataSet.Tables["Customers"].Columns["CustomerID"],
 dataSet.Tables["Orders"].Columns["CustomerID"]).Nested = true;

dataSet.Relations.Add("OrderDetail",
  dataSet.Tables["Orders"].Columns["OrderID"],
  dataSet.Tables["OrderDetails"].Columns["OrderID"],
  false).Nested = true;

XmlDataDocument xmlDoc = new XmlDataDocument(dataSet);

XmlNodeList nodeList = xmlDoc.DocumentElement.SelectNodes(
  "descendant::Customers[*/OrderDetails/ProductID=43]");

DataRow dataRow;
foreach (XmlNode xmlNode in nodeList)
{
  dataRow = xmlDoc.GetRowFromElement((XmlElement)xmlNode);
  if (dataRow != null)
    Console.WriteLine(dataRow[0]);
}
```

See Also

DataSet and XmlDataDocument Synchronization

ADO.NET Managed Providers and DataSet Developer Center

Applying an XSLT Transform to a DataSet

8/31/2018 • 2 minutes to read • Edit Online

The **WriteXml** method of the DataSet enables you to write the contents of a **DataSet** as XML data. A common task is to then transform that XML to another format using XSL transformations (XSLT). However, synchronizing a **DataSet** with an XmlDataDocument enables you to apply an XSLT stylesheet to the contents of a **DataSet** without having to first write the contents of the **DataSet** as XML data using **WriteXml**.

The following example populates a **DataSet** with tables and relationships, synchronizes the **DataSet** with an **XmlDataDocument**, and writes a portion of the **DataSet** as an HTML file using an XSLT stylesheet. Following are the contents of the XSLT stylesheet.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="CustomerOrders">
  <HTML>
  <STYLE>
  BODY {font-family:verdana;font-size:9pt}
  TD   {font-size:8pt}
  </STYLE>
    <BODY>
    <TABLE BORDER="1">
      <xsl:apply-templates select="Customers"/>
    </TABLE>
    </BODY>
  </HTML>
</xsl:template>

<xsl:template match="Customers">
    <TR><TD>
      <xsl:value-of select="ContactName"/>, <xsl:value-of select="Phone"/><BR/>
    </TD></TR>
      <xsl:apply-templates select="Orders"/>
</xsl:template>

<xsl:template match="Orders">
  <TABLE BORDER="1">
    <TR><TD valign="top"><B>Order:</B></TD><TD valign="top"><xsl:value-of select="OrderID"/></TD></TR>
    <TR><TD valign="top"><B>Date:</B></TD><TD valign="top"><xsl:value-of select="OrderDate"/></TD></TR>
    <TR><TD valign="top"><B>Ship To:</B></TD>
        <TD valign="top"><xsl:value-of select="ShipName"/><BR/>
        <xsl:value-of select="ShipAddress"/><BR/>
        <xsl:value-of select="ShipCity"/>, <xsl:value-of select="ShipRegion"/>  <xsl:value-of
select="ShipPostalCode"/><BR/>
        <xsl:value-of select="ShipCountry"/></TD></TR>
  </TABLE>
</xsl:template>

</xsl:stylesheet>
```

The following code fills the **DataSet** and applies the XSLT style sheet.

NOTE

If you are applying an XSLT style sheet to a **DataSet** that contains relations, you achieve best performance if you set the **Nested** property of the DataRelation to **true** for each nested relation. This allows you to use XSLT style sheets that implement natural top-down processing to navigate the hierarchy and transform the data, as opposed to using performance-intensive XPath location axes (for example, preceding-sibling and following-sibling in style sheet node test expressions) to navigate it. For more information on nested relations, see Nesting DataRelations.

```
' Assumes connection is a valid SqlConnection.
Dim dataSet As DataSet = New DataSet("CustomerOrders")

Dim customerAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM Customers", connection)
customerAdapter.Fill(dataSet, "Customers")

Dim orderAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT * FROM Orders", connection)
orderAdapter.Fill(dataSet, "Orders")

connection.Close()

dataSet.Relations.Add("CustOrders", _
dataSet.Tables("Customers").Columns("CustomerID"), _
dataSet.Tables("Orders").Columns("CustomerID")).Nested = true

Dim xmlDoc As XmlDataDocument = New XmlDataDocument(dataSet)

Dim xslTran As XslTransform = New XslTransform
xslTran.Load("transform.xsl")

Dim writer As XmlTextWriter = New XmlTextWriter( _
  "xslt_output.html", System.Text.Encoding.UTF8)

xslTran.Transform(xmlDoc, Nothing, writer)
writer.Close()
```

```
// Assumes connection is a valid SqlConnection.
connection.Open();

DataSet custDS = new DataSet("CustomerDataSet");

SqlDataAdapter customerAdapter = new SqlDataAdapter(
  "SELECT * FROM Customers", connection);
customerAdapter.Fill(custDS, "Customers");

SqlDataAdapter orderAdapter = new SqlDataAdapter(
  "SELECT * FROM Orders", connection);
orderAdapter.Fill(custDS, "Orders");

connection.Close();

custDS.Relations.Add("CustOrders",
  custDS.Tables["Customers"].Columns["CustomerID"],
                    custDS.Tables["Orders"].Columns["CustomerID"]).Nested = true;

XmlDataDocument xmlDoc = new XmlDataDocument(custDS);

XslTransform xslTran = new XslTransform();
xslTran.Load("transform.xsl");

XmlTextWriter writer = new XmlTextWriter("xslt_output.html",
  System.Text.Encoding.UTF8);

xslTran.Transform(xmlDoc, null, writer);
writer.Close();
```

See Also

DataSet and XmlDataDocument Synchronization
ADO.NET Managed Providers and DataSet Developer Center

Nesting DataRelations

8/31/2018 • 2 minutes to read • Edit Online

In a relational representation of data, individual tables contain rows that are related to one another using a column or set of columns. In the ADO.NET DataSet, the relationship between tables is implemented using a DataRelation. When you create a **DataRelation**, the parent-child relationships of the columns are managed only through the relation. The tables and columns are separate entities. In the hierarchical representation of data that XML provides, the parent-child relationships are represented by parent elements that contain nested child elements.

To facilitate the nesting of child objects when a **DataSet** is synchronized with an XmlDataDocument or written as XML data using **WriteXml**, the **DataRelation** exposes a **Nested** property. Setting the **Nested** property of a **DataRelation** to **true** causes the child rows of the relation to be nested within the parent column when written as XML data or synchronized with an **XmlDataDocument**. The **Nested** property of the **DataRelation** is **false**, by default.

For example, consider the following **DataSet**.

```
' Assumes connection is a valid SqlConnection.
Dim customerAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT CustomerID, CompanyName FROM Customers", connection)
Dim orderAdapter As SqlDataAdapter = New SqlDataAdapter( _
  "SELECT OrderID, CustomerID, OrderDate FROM Orders", connection)

connection.Open()

Dim dataSet As DataSet = New DataSet("CustomerOrders")
customerAdapter.Fill(dataSet, "Customers")
orderAdapter.Fill(dataSet, "Orders")

connection.Close()

Dim customerOrders As DataRelation = dataSet.Relations.Add( _
  "CustOrders", dataSet.Tables("Customers").Columns("CustomerID"), _
  dataSet.Tables("Orders").Columns("CustomerID"))
```

```
// Assumes connection is a valid SqlConnection.
SqlDataAdapter customerAdapter = new SqlDataAdapter(
  "SELECT CustomerID, CompanyName FROM Customers", connection);
SqlDataAdapter orderAdapter = new SqlDataAdapter(
  "SELECT OrderID, CustomerID, OrderDate FROM Orders", connection);

connection.Open();

DataSet dataSet = new DataSet("CustomerOrders");
customerAdapter.Fill(dataSet, "Customers");
orderAdapter.Fill(dataSet, "Orders");

connection.Close();

DataRelation customerOrders = dataSet.Relations.Add(
  "CustOrders", dataSet.Tables["Customers"].Columns["CustomerID"],
  dataSet.Tables["Orders"].Columns["CustomerID"]);
```

Because the **Nested** property of the **DataRelation** object is not set to **true** for this **DataSet**, the child objects are not nested within the parent elements when this **DataSet** is represented as XML data. Transforming the XML

representation of a **DataSet** that contains related **DataSet**s with non-nested data relations can cause slow performance. We recommend that you nest the data relations. To do this, set the **Nested** property to **true**. Then write code in the XSLT style sheet that uses top-down hierarchical XPath query expressions to locate and transform the data.

The following code example shows the result from calling **WriteXml** on the **DataSet**.

```
<CustomerOrders>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <CompanyName>Alfreds Futterkiste</CompanyName>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  </Customers>
  <Orders>
    <OrderID>10643</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <OrderDate>1997-08-25T00:00:00</OrderDate>
  </Orders>
  <Orders>
    <OrderID>10692</OrderID>
    <CustomerID>ALFKI</CustomerID>
    <OrderDate>1997-10-03T00:00:00</OrderDate>
  </Orders>
  <Orders>
    <OrderID>10308</OrderID>
    <CustomerID>ANATR</CustomerID>
    <OrderDate>1996-09-18T00:00:00</OrderDate>
  </Orders>
</CustomerOrders>
```

Note that the **Customers** element and the **Orders** elements are shown as sibling elements. If you wanted the **Orders** elements to show up as children of their respective parent elements, the **Nested** property of the **DataRelation** would need to be set to **true** and you would add the following:

```
customerOrders.Nested = True
```

```
customerOrders.Nested = true;
```

The following code shows what the resulting output would look like, with the **Orders** elements nested within their respective parent elements.

```
<CustomerOrders>
  <Customers>
    <CustomerID>ALFKI</CustomerID>
    <Orders>
      <OrderID>10643</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <OrderDate>1997-08-25T00:00:00</OrderDate>
    </Orders>
    <Orders>
      <OrderID>10692</OrderID>
      <CustomerID>ALFKI</CustomerID>
      <OrderDate>1997-10-03T00:00:00</OrderDate>
    </Orders>
    <CompanyName>Alfreds Futterkiste</CompanyName>
  </Customers>
  <Customers>
    <CustomerID>ANATR</CustomerID>
    <Orders>
      <OrderID>10308</OrderID>
      <CustomerID>ANATR</CustomerID>
      <OrderDate>1996-09-18T00:00:00</OrderDate>
    </Orders>
    <CompanyName>Ana Trujillo Emparedados y helados</CompanyName>
  </Customers>
</CustomerOrders>
```

See Also

Using XML in a DataSet
Adding DataRelations
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Deriving DataSet Relational Structure from XML Schema (XSD)

8/31/2018 • 2 minutes to read • Edit Online

This section provides an overview of how the relational schema of a `DataSet` is built from an XML Schema definition language (XSD) schema document. In general, for each `complexType` child element of a schema element, a table is generated in the `DataSet`. The table structure is determined by the definition of the complex type. Tables are created in the `DataSet` for top-level elements in the schema. However, a table is only created for a top-level `complexType` element when the `complexType` element is nested inside another `complexType` element, in which case the nested `complexType` element is mapped to a `DataTable` within the `DataSet`.

For more information about the XSD, see the World Wide Web Consortium (W3C) XML Schema Part 0: Primer Recommendation, the XML Schema Part 1: Structures Recommendation, and the XML Schema Part 2: Datatypes Recommendation, located at http://www.w3.org/.

The following example demonstrates an XML Schema where `customers` is the child element of the `MyDataSet` element, which is a **DataSet** element.

```
<xs:schema id="SomeID"
           xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="MyDataSet" msdata:IsDataSet="true">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="customers" >
          <xs:complexType >
            <xs:sequence>
              <xs:element name="CustomerID" type="xs:integer"
                          minOccurs="0" />
              <xs:element name="CompanyName" type="xs:string"
                          minOccurs="0" />
              <xs:element name="Phone" type="xs:string" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

In the preceding example, the element `customers` is a complex type element. Therefore, the complex type definition is parsed, and the mapping process creates the following table.

```
Customers (CustomerID , CompanyName, Phone)
```

The data type of each column in the table is derived from the XML Schema type of the corresponding element or attribute specified.

> **NOTE**
>
> If the element `customers` is of a simple XML Schema data type such as **integer**, no table is generated. Tables are only created for the top-level elements that are complex types.

In the following XML Schema, the **Schema** element has two element children, `InStateCustomers` and `OutOfStateCustomers`.

```
<xs:schema id="SomeID"
            xmlns=""
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
   <xs:element name="InStateCustomers" type="customerType" />
   <xs:element name="OutOfStateCustomers" type="customerType" />
    <xs:complexType name="customerType" >

     </xs:complexType>

   <xs:element name="MyDataSet" msdata:IsDataSet="true">
     <xs:complexType>
       <xs:choice maxOccurs="unbounded">
         <xs:element ref="customers" />
       </xs:choice>
     </xs:complexType>
   </xs:element>
 </xs:schema>
```

Both the `InStateCustomers` and the `OutOfStateCustomers` child elements are complex type elements (`customerType`). Therefore, the mapping process generates the following two identical tables in the `DataSet`.

```
InStateCustomers (CustomerID , CompanyName, Phone)
OutOfStateCustomers (CustomerID , CompanyName, Phone)
```

In This Section

Mapping XML Schema (XSD) Constraints to DataSet Constraints

Describes the XML Schema elements used to create unique and foreign key constraints in a `DataSet`.

Generating DataSet Relations from XML Schema (XSD)

Describes the XML Schema elements used to create relations between table columns in a `DataSet`.

XML Schema Constraints and Relationships

Describes how relations are created implicitly when using XML Schema elements to create constraints in a `DataSet`.

Related Sections

Using XML in a DataSet

Describes how to load and persist the relational structure and data in a `DataSet` as XML data.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Mapping XML Schema (XSD) Constraints to DataSet Constraints

8/31/2018 • 2 minutes to read • Edit Online

The XML Schema definition language (XSD) allows constraints to be specified on the elements and attributes it defines. When mapping an XML Schema to relational schema in a DataSet, XML Schema constraints are mapped to appropriate relational constraints on the tables and columns within the **DataSet**.

This section discusses the mapping of the following XML Schema constraints:

- The uniqueness constraint specified using the **unique** element.
- The key constraint specified using the **key** element.
- The keyref constraint specified using the **keyref** element.

By using a constraint on an element or attribute, you specify certain restrictions on the values of the element in any instance of the document. For example, a key constraint on a **CustomerID** child element of a **Customer** element in the schema indicates that the values of the **CustomerID** child element must be unique in any document instance, and that null values are not allowed.

Constraints can also be specified between elements and attributes in a document, in order to establish a relationship within the document. The key and keyref constraints are used in the schema to specify the constraints within the document, resulting in a relationship between document elements and attributes.

The mapping process converts these schema constraints into appropriate constraints on the tables created within the **DataSet**.

In This Section

Map unique XML Schema (XSD) Constraints to DataSet Constraints
Describes the XML Schema elements used to create unique constraints in a **DataSet**.

Map key XML Schema (XSD) Constraints to DataSet Constraints
Describes the XML Schema elements used to create key constraints (unique constraints where null values are not allowed) in a **DataSet**.

Map keyref XML Schema (XSD) Constraints to DataSet Constraints
Describes the XML Schema elements used to create keyref (foreign key) constraints in a **DataSet**.

Related Sections

Deriving DataSet Relational Structure from XML Schema (XSD)
Describes the relational structure, or schema, of a **DataSet** that is created from XSD schema.

Generating DataSet Relations from XML Schema (XSD)
Describes the XML Schema elements used to create relations between table columns in a **DataSet**.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Map unique XML Schema (XSD) Constraints to DataSet Constraints

8/31/2018 • 2 minutes to read • Edit Online

In an XML Schema definition language (XSD) schema, the **unique** element specifies the uniqueness constraint on an element or attribute. In the process of translating an XML Schema into a relational schema, the unique constraint specified on an element or attribute in the XML Schema is mapped to a unique constraint in the DataTable in the corresponding DataSet that is generated.

The following table outlines the **msdata** attributes that you can specify in the **unique** element.

ATTRIBUTE NAME	DESCRIPTION
msdata:ConstraintName	If this attribute is specified, its value is used as the constraint name. Otherwise, the **name** attribute provides the value of the constraint name.
msdata:PrimaryKey	If `PrimaryKey="true"` is present in the **unique** element, a unique constraint is created with the **IsPrimaryKey** property set to **true**.

The following example shows an XML Schema that uses the **unique** element to specify a uniqueness constraint.

```
<xs:schema id="SampleDataSet"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CustomerID" type="xs:integer"
           minOccurs="0"/>
        <xs:element name="CompanyName" type="xs:string"
           minOccurs="0"/>
       <xs:element name="Phone" type="xs:string" />
     </xs:sequence>
   </xs:complexType>
 </xs:element>

 <xs:element name="SampleDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element ref="Customers" />
    </xs:choice>
  </xs:complexType>
   <xs:unique     msdata:ConstraintName="UCustID"     name="UniqueCustIDConstr" >       <xs:selector
xpath=".//Customers" />       <xs:field xpath="CustomerID" />     </xs:unique>
</xs:element>
</xs:schema>
```

The **unique** element in the schema specifies that for all **Customers** elements in a document instance, the value of the **CustomerID** child element must be unique. In building the **DataSet**, the mapping process reads this schema and generates the following table:

```
Customers (CustomerID, CompanyName, Phone)
```

The mapping process also creates a unique constraint on the **CustomerID** column, as shown in the following **DataSet**. (For simplicity, only relevant properties are shown.)

```
      DataSetName: MyDataSet
TableName: Customers
  ColumnName: CustomerID
      AllowDBNull: True
      Unique: True
  ConstraintName: UcustID       Type: UniqueConstraint
      Table: Customers
      Columns: CustomerID
      IsPrimaryKey: False
```

In the **DataSet** that is generated, the **IsPrimaryKey** property is set to **False** for the unique constraint. The **unique** property on the column indicates that the **CustomerID** column values must be unique (but they can be a null reference, as specified by the **AllowDBNull** property of the column).

If you modify the schema and set the optional **msdata:PrimaryKey** attribute value to **True**, the unique constraint is created on the table. The **AllowDBNull** column property is set to **False**, and the **IsPrimaryKey** property of the constraint set to **True**, thus making the **CustomerID** column a primary key column.

You can specify a unique constraint on a combination of elements or attributes in the XML Schema. The following example demonstrates how to specify that a combination of **CustomerID** and **CompanyName** values must be unique for all **Customers** in any instance, by adding another **xs:field** element in the schema.

```
      <xs:unique
         msdata:ConstraintName="SomeName"
         name="UniqueCustIDConstr" >
  <xs:selector xpath=".//Customers" />
  <xs:field xpath="CustomerID" />
  <xs:field xpath="CompanyName" />
</xs:unique>
```

This is the constraint that is created in the resulting **DataSet**.

```
ConstraintName: SomeName
  Table: Customers
  Columns: CustomerID CompanyName
  IsPrimaryKey: False
```

See Also

Mapping XML Schema (XSD) Constraints to DataSet Constraints
Generating DataSet Relations from XML Schema (XSD)
ADO.NET Managed Providers and DataSet Developer Center

Map key XML Schema (XSD) Constraints to DataSet Constraints

8/31/2018 • 2 minutes to read • Edit Online

In a schema, you can specify a key constraint on an element or attribute using the **key** element. The element or attribute on which a key constraint is specified must have unique values in any schema instance, and cannot have null values.

The key constraint is similar to the unique constraint, except that the column on which a key constraint is defined cannot have null values.

The following table outlines the **msdata** attributes that you can specify in the **key** element.

ATTRIBUTE NAME	DESCRIPTION
msdata:ConstraintName	If this attribute is specified, its value is used as the constraint name. Otherwise, the **name** attribute provides the value of the constraint name.
msdata:PrimaryKey	If `PrimaryKey="true"` is present, the **IsPrimaryKey** constraint property is set to **true**, thus making it a primary key. The **AllowDBNull** column property is set to **false**, because primary keys cannot have null values.

In converting schema in which a key constraint is specified, the mapping process creates a unique constraint on the table with the **AllowDBNull** column property set to **false** for each column in the constraint. The **IsPrimaryKey** property of the unique constraint is also set to **false** unless you have specified `msdata:PrimaryKey="true"` on the **key** element. This is identical to a unique constraint in the schema in which `PrimaryKey="true"`.

In the following schema example, the **key** element specifies the key constraint on the **CustomerID** element.

```
<xs:schema id="cod"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="Customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="CustomerID" type="xs:string" minOccurs="0" />
        <xs:element name="CompanyName" type="xs:string" minOccurs="0" />
       <xs:element name="Phone" type="xs:string" />
     </xs:sequence>
   </xs:complexType>
 </xs:element>
<xs:element name="MyDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element ref="Customers" />
    </xs:choice>
  </xs:complexType>
   <xs:key  msdata:PrimaryKey="true"
       msdata:ConstraintName="KeyCustID"
          name="KeyConstCustomerID" >
     <xs:selector xpath=".//Customers" />
     <xs:field xpath="CustomerID" />
    </xs:key>
 </xs:element>
</xs:schema>
```

The **key** element specifies that the values of the **CustomerID** child element of the **Customers** element must have unique values and cannot have null values. In translating the XML Schema definition language (XSD) schema, the mapping process creates the following table:

```
Customers(CustomerID, CompanyName, Phone)
```

The XML Schema mapping also creates a **UniqueConstraint** on the **CustomerID** column, as shown in the following DataSet. (For simplicity, only relevant properties are shown.)

```
      DataSetName: MyDataSet
TableName: customers
  ColumnName: CustomerID
      AllowDBNull: False
      Unique: True
  ConstraintName: KeyCustID
      Table: customers
      Columns: CustomerID
      IsPrimaryKey: True
```

In the **DataSet** that is generated, the **IsPrimaryKey** property of the **UniqueConstraint** is set to **true** because the schema specifies `msdata:PrimaryKey="true"` in the **key** element.

The value of the **ConstraintName** property of the **UniqueConstraint** in the **DataSet** is the value of the **msdata:ConstraintName** attribute specified in the **key** element in the schema.

See Also

Mapping XML Schema (XSD) Constraints to DataSet Constraints
Generating DataSet Relations from XML Schema (XSD)
ADO.NET Managed Providers and DataSet Developer Center

Map keyref XML Schema (XSD) Constraints to DataSet Constraints

8/31/2018 • 2 minutes to read • Edit Online

The **keyref** element allows you to establish links between elements within a document. This is similar to a foreign key relationship in a relational database. If a schema specifies the **keyref** element, the element is converted during the schema mapping process to a corresponding foreign key constraint on the columns in the tables of the DataSet. By default, the **keyref** element also generates a relation, with the **ParentTable**, **ChildTable**, **ParentColumn**, and **ChildColumn** properties specified on the relation.

The following table outlines the **msdata** attributes you can specify in the **keyref** element.

ATTRIBUTE NAME	DESCRIPTION
msdata:ConstraintOnly	If **ConstraintOnly="true"** is specified on the **keyref** element in the schema, a constraint is created, but no relation is created. If this attribute is not specified (or is set to **False**), both the constraint and the relation are created in the **DataSet**.
msdata:ConstraintName	If the **ConstraintName** attribute is specified, its value is used as the name of the constraint. Otherwise, the **name** attribute of the **keyref** element in the schema provides the constraint name in the **DataSet**.
msdata:UpdateRule	If the **UpdateRule** attribute is specified in the **keyref** element in the schema, its value is assigned to the **UpdateRule** constraint property in the **DataSet**. Otherwise the **UpdateRule** property is set to **Cascade**.
msdata:DeleteRule	If the **DeleteRule** attribute is specified in the **keyref** element in the schema, its value is assigned to the **DeleteRule** constraint property in the **DataSet**. Otherwise the **DeleteRule** property is set to **Cascade**.
msdata:AcceptRejectRule	If the **AcceptRejectRule** attribute is specified in the **keyref** element in the schema, its value is assigned to the **AcceptRejectRule** constraint property in the **DataSet**. Otherwise the **AcceptRejectRule** property is set to **None**.

The following example contains a schema that specifies the **key** and **keyref** relationships between the **OrderNumber** child element of the **Order** element and the **OrderNo** child element of the **OrderDetail** element.

In the example, the **OrderNumber** child element of the **OrderDetail** element refers to the **OrderNo** key child element of the **Order** element.

```
<xs:schema id="MyDataSet" xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

 <xs:element name="MyDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="OrderDetail">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="OrderNo" type="xs:integer" />
           <xs:element name="ItemNo" type="xs:string" />
         </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="Order">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="OrderNumber" type="xs:integer" />
            <xs:element name="EmpNumber" type="xs:integer" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>

  <xs:key name="OrderNumberKey"  >
    <xs:selector xpath=".//Order" />
    <xs:field xpath="OrderNumber" />
  </xs:key>

  <xs:keyref name="OrderNoRef" refer="OrderNumberKey">
    <xs:selector xpath=".//OrderDetail" />
    <xs:field xpath="OrderNo" />
  </xs:keyref>
 </xs:element>
</xs:schema>
```

The XML Schema definition language (XSD) schema mapping process produces the following **DataSet** with two tables:

```
OrderDetail(OrderNo, ItemNo) and
Order(OrderNumber, EmpNumber)
```

In addition, the **DataSet** defines the following constraints:

- A unique constraint on the **Order** table.

  ```
            Table: Order
   Columns: OrderNumber
   ConstraintName: OrderNumberKey
   Type: UniqueConstraint
   IsPrimaryKey: False
  ```

- A relationship between the **Order** and **OrderDetail** tables. The **Nested** property is set to **False** because the two elements are not nested in the schema.

```
          ParentTable: Order
ParentColumns: OrderNumber
ChildTable: OrderDetail
ChildColumns: OrderNo
ParentKeyConstraint: OrderNumberKey
ChildKeyConstraint: OrderNoRef
RelationName: OrderNoRef
Nested: False
```

- A foreign key constraint on the **OrderDetail** table.

```
          ConstraintName: OrderNoRef
Type: ForeignKeyConstraint
Table: OrderDetail
Columns: OrderNo
RelatedTable: Order
RelatedColumns: OrderNumber
```

See Also

Mapping XML Schema (XSD) Constraints to DataSet Constraints
Generating DataSet Relations from XML Schema (XSD)
ADO.NET Managed Providers and DataSet Developer Center

Generating DataSet Relations from XML Schema (XSD)

8/31/2018 • 2 minutes to read • Edit Online

In a DataSet, you form an association between two or more columns by creating a parent-child relation. There are three ways to represent a **DataSet** relation within an XML Schema definition language (XSD) schema:

- Specify nested complex types.
- Use the **msdata:Relationship** annotation.
- Specify an **xs:keyref** without the **msdata:ConstraintOnly** annotation.

Nested Complex Types

Nested complex type definitions in a schema indicate the parent-child relationships of the elements. The following XML Schema fragment shows that **OrderDetail** is a child element of the **Order** element.

```
<xs:element name="Order">
  <xs:complexType>
     <xs:sequence>
       <xs:element name="OrderDetail" />
           <xs:complexType>
           </xs:complexType>
     </xs:sequence>
  </xs:complexType>
</xs:element>
```

The XML Schema mapping process creates tables in the **DataSet** that correspond to the nested complex types in the schema. It also creates additional columns that are used as parent-child columns for the generated tables. Note that these parent-child columns specify relationships, which is not the same as specifying primary key/foreign key constraints.

msdata:Relationship Annotation

The **msdata:Relationship** annotation allows you to explicitly specify parent-child relationships between elements in the schema that are not nested. The following example shows the structure of the **Relationship** element.

```
<msdata:Relationship name="CustOrderRelationship"
msdata:parent=""
msdata:child=""
msdata:parentkey=""
msdata:childkey="" />
```

The attributes of the **msdata:Relationship** annotation identify the elements involved in the parent-child relationship, as well as the **parentkey** and **childkey** elements and attributes involved in the relationship. The mapping process uses this information to generate tables in the **DataSet** and to create the primary key/foreign key relationship between these tables.

For example, the following schema fragment specifies **Order** and **OrderDetail** elements at the same level (not nested). The schema specifies an **msdata:Relationship** annotation, which specifies the parent-child relationship

between these two elements. In this case, an explicit relationship must be specified using the **msdata:Relationship** annotation.

```
 <xs:element name="MyDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
        <xs:element name="OrderDetail">
          <xs:complexType>

          </xs:complexType>
       </xs:element>
       <xs:element name="Order">
          <xs:complexType>

          </xs:complexType>
       </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
   <xs:annotation>
     <xs:appinfo>
       <msdata:Relationship name="OrdOrdDetailRelation"
          msdata:parent="Order"
          msdata:child="OrderDetail"
          msdata:parentkey="OrderNumber"
          msdata:childkey="OrderNo"/>
     </xs:appinfo>
  </xs:annotation>
```

The mapping process uses the **Relationship** element to create a parent-child relationship between the **OrderNumber** column in the **Order** table and the **OrderNo** column in the **OrderDetail** table in the **DataSet**. The mapping process only specifies the relationship; it does not automatically specify any constraints on the values in these columns, as do the primary key/foreign key constraints in relational databases.

In This Section

Map Implicit Relations Between Nested Schema Elements
Describes the constraints and relations that are implicitly created in a **DataSet** when nested elements are encountered in XML Schema.

Map Relations Specified for Nested Elements
Describes how to explicitly set relations in a **DataSet** for nested elements in XML Schema.

Specify Relations Between Elements with No Nesting
Describes how to create relations in a **DataSet** between XML Schema elements that are not nested.

Related Sections

Deriving DataSet Relational Structure from XML Schema (XSD)
Describes the relational structure, or schema, of a **DataSet** that is created from XML Schema definition language (XSD) schema.

Mapping XML Schema (XSD) Constraints to DataSet Constraints
Describes the XML Schema elements used to create unique and foreign key constraints in a **DataSet**.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Map Implicit Relations Between Nested Schema Elements

8/31/2018 • 2 minutes to read • Edit Online

An XML Schema definition language (XSD) schema can have complex types nested inside one another. In this case, the mapping process applies default mapping and creates the following in the DataSet:

- One table for each of the complex types (parent and child).
- If no unique constraint exists on the parent, one additional primary key column per table definition named *TableName*_Id where *TableName* is the name of the parent table.
- A primary key constraint on the parent table identifying the additional column as the primary key (by setting the **IsPrimaryKey** property to **True**). The constraint is named Constraint# where # is 1, 2, 3, and so on. For example, the default name for the first constraint is Constraint1.
- A foreign key constraint on the child table identifying the additional column as the foreign key referring to the primary key of the parent table. The constraint is named *ParentTable*_*ChildTable* where *ParentTable* is the name of the parent table and *ChildTable* is the name of the child table.
- A data relation between the parent and child tables.

The following example shows a schema where **OrderDetail** is a child element of **Order**.

```
<xs:schema id="MyDataSet" xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

 <xs:element name="MyDataSet" msdata:IsDataSet="true">
   <xs:complexType>
     <xs:choice maxOccurs="unbounded">
       <xs:element name="Order">
         <xs:complexType>
          <xs:sequence>
            <xs:element name="OrderNumber" type="xs:string" />
            <xs:element name="EmpNumber" type="xs:string" />
            <xs:element name="OrderDetail">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="OrderNo" type="xs:string" />
                  <xs:element name="ItemNo" type="xs:string" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:choice>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

The XML Schema mapping process creates the following in the **DataSet**:

- An **Order** and an **OrderDetail** table.

```
Order(OrderNumber, EmpNumber, Order_Id)
OrderDetail(OrderNo, ItemNo, Order_Id)
```

- A unique constraint on the **Order** table. Note that the **IsPrimaryKey** property is set to **True**.

```
ConstraintName: Constraint1
Type: UniqueConstraint
Table: Order
Columns: Order_Id
IsPrimaryKey: True
```

- A foreign key constraint on the **OrderDetail** table.

```
ConstraintName: Order_OrderDetail
Type: ForeignKeyConstraint
Table: OrderDetail
Columns: Order_Id
RelatedTable: Order
RelatedColumns: Order_Id
```

- A relationship between the **Order** and **OrderDetail** tables. The **Nested** property for this relationship is set to **True** because the **Order** and **OrderDetail** elements are nested in the schema.

```
ParentTable: Order
ParentColumns: Order_Id
ChildTable: OrderDetail
ChildColumns: Order_Id
ParentKeyConstraint: Constraint1
ChildKeyConstraint: Order_OrderDetail
RelationName: Order_OrderDetail
Nested: True
```

See Also

Generating DataSet Relations from XML Schema (XSD)
Mapping XML Schema (XSD) Constraints to DataSet Constraints
ADO.NET Managed Providers and DataSet Developer Center

Map Relations Specified for Nested Elements

8/31/2018 • 2 minutes to read • Edit Online

A schema can include an **msdata:Relationship** annotation to explicitly specify the mapping between any two elements in the schema. The two elements specified in **msdata:Relationship** can be nested in the schema, but do not have to be. The mapping process uses **msdata:Relationship** in the schema to generate the primary key/foreign key relationship between the two columns.

The following example shows an XML Schema in which the **OrderDetail** element is a child element of **Order**. The **msdata:Relationship** identifies this parent-child relationship and specifies that the **OrderNumber** column of the resulting **Order** table is related to the **OrderNo** column of the resulting **OrderDetail** table.

```
<xs:schema id="MyDataSet" xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
<xs:element name="MyDataSet" msdata:IsDataSet="true">
 <xs:complexType>
  <xs:choice maxOccurs="unbounded">
   <xs:element name="Order">
    <xs:complexType>
     <xs:sequence>
       <xs:element name="OrderNumber" type="xs:string" />
       <xs:element name="EmpNumber" type="xs:string" />
       <xs:element name="OrderDetail">
          <xs:annotation>
           <xs:appinfo>
            <msdata:Relationship name="OrdODRelation"
                                 msdata:parent="Order"
                                 msdata:child="OrderDetail"
                                 msdata:parentkey="OrderNumber"
                                 msdata:childkey="OrderNo"/>
           </xs:appinfo>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
             <xs:element name="OrderNo" type="xs:string" />
             <xs:element name="ItemNo" type="xs:string" />
            </xs:sequence>
         </xs:complexType>
       </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:choice>
 </xs:complexType>
</xs:element>
</xs:schema>
```

The XML Schema mapping process creates the following in the DataSet:

- An **Order** and an **OrderDetail** table.

  ```
  Order(OrderNumber, EmpNumber)
  OrderDetail(OrderNo, ItemNo)
  ```

- A relationship between the **Order** and **OrderDetail** tables. The **Nested** property for this relationship is set to **True** because the **Order** and **OrderDetail** elements are nested in the schema.

```
ParentTable: Order
ParentColumns: OrderNumber
ChildTable: OrderDetail
ChildColumns: OrderNo
RelationName: OrdODRelation
Nested: True
```

The mapping process does not create any constraints.

See Also

Generating DataSet Relations from XML Schema (XSD)
Mapping XML Schema (XSD) Constraints to DataSet Constraints
ADO.NET Managed Providers and DataSet Developer Center

Specify Relations Between Elements with No Nesting

8/31/2018 • 2 minutes to read • Edit Online

When elements are not nested, no implicit relations are created. You can, however, explicitly specify relations between elements that are not nested by using the **msdata:Relationship** annotation.

The following example shows an XML Schema in which the **msdata:Relationship** annotation is specified between the **Order** and **OrderDetail** elements, which are not nested. The **msdata:Relationship** annotation is specified as the child element of the **Schema** element.

```
<xs:schema id="MyDataSet" xmlns=""
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
 <xs:element name="MyDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="OrderDetail">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="OrderNo" type="xs:string" />
           <xs:element name="ItemNo" type="xs:string" />
         </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="Order">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="OrderNumber" type="xs:string" />
           <xs:element name="EmpNumber" type="xs:string" />
         </xs:sequence>
       </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>

  </xs:element>
   <xs:annotation>
     <xs:appinfo>
       <msdata:Relationship name="OrdOrderDetailRelation"
                            msdata:parent="Order"
                            msdata:child="OrderDetail"
                            msdata:parentkey="OrderNumber"
                            msdata:childkey="OrderNo"/>
     </xs:appinfo>
  </xs:annotation>
</xs:schema>
```

The XML Schema definition language (XSD) schema mapping process creates a DataSet with **Order** and **OrderDetail** tables and a relationship specified between these two tables, as shown below.

```
RelationName: OrdOrderDetailRelation
ParentTable: Order
ParentColumns: OrderNumber
ChildTable: OrderDetail
ChildColumns: OrderNo
Nested: False
```

See Also

Generating DataSet Relations from XML Schema (XSD)
Mapping XML Schema (XSD) Constraints to DataSet Constraints
ADO.NET Managed Providers and DataSet Developer Center

XML Schema Constraints and Relationships

8/31/2018 • 3 minutes to read • Edit Online

In an XML Schema definition language (XSD) schema, you can specify constraints (unique, key, and keyref constraints) and relationships (using the **msdata:Relationship** annotation). This topic explains how the constraints and relationships specified in an XML Schema are interpreted to generate the DataSet.

In general, in an XML Schema, you specify the **msdata:Relationship** annotation if you want to generate only relationships in the **DataSet**. For more information, see Generating DataSet Relations from XML Schema (XSD). You specify constraints (unique, key, and keyref) if you want to generate constraints in the **DataSet**. Note that the key and keyref constraints are also used to generate relationships, as explained later in this topic.

Generating a Relationship from key and keyref Constraints

Instead of specifying the **msdata:Relationship** annotation, you can specify key and keyref constraints, which are used during the XML Schema mapping process to generate not only the constraints but also the relationship in the **DataSet**. However, if you specify `msdata:ConstraintOnly="true"` in the **keyref** element, the **DataSet** will include only the constraints and will not include the relationship.

The following example shows an XML Schema that includes **Order** and **OrderDetail** elements, which are not nested. The schema also specifies key and keyref constraints.

```
<xs:schema id="MyDataSet" xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

 <xs:element name="MyDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="OrderDetail">
       <xs:complexType>
         <xs:sequence>
           <xs:element name="OrderNo" type="xs:integer" />
           <xs:element name="ItemNo" type="xs:string" />
         </xs:sequence>
       </xs:complexType>
      </xs:element>
      <xs:element name="Order">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="OrderNumber" type="xs:integer" />
            <xs:element name="EmpNumber" type="xs:integer" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>

  <xs:key name="OrderNumberKey"  >
    <xs:selector xpath=".//Order" />
    <xs:field xpath="OrderNumber" />
  </xs:key>

  <xs:keyref name="OrderNoRef" refer="OrderNumberKey">
    <xs:selector xpath=".//OrderDetail" />
    <xs:field xpath="OrderNo" />
  </xs:keyref>
 </xs:element>
</xs:schema>
```

The **DataSet** that is generated during the XML Schema mapping process includes the **Order** and **OrderDetail** tables. In addition, the **DataSet** includes relationships and constraints. The following example shows these relationships and constraints. Note that the schema does not specify the **msdata:Relationship** annotation; instead, the key and keyref constraints are used to generate the relation.

```
....ConstraintName: OrderNumberKey
....Type: UniqueConstraint
....Table: Order
....Columns: OrderNumber
....IsPrimaryKey: False

....ConstraintName: OrderNoRef
....Type: ForeignKeyConstraint
....Table: OrderDetail
....Columns: OrderNo
....RelatedTable: Order
....RelatedColumns: OrderNumber

..RelationName: OrderNoRef
..ParentTable: Order
..ParentColumns: OrderNumber
..ChildTable: OrderDetail
..ChildColumns: OrderNo
..ParentKeyConstraint: OrderNumberKey
..ChildKeyConstraint: OrderNoRef
..Nested: False
```

In the previous schema example, the **Order** and **OrderDetail** elements are not nested. In the following schema example, these elements are nested. However, no **msdata:Relationship** annotation is specified; therefore, an implicit relation is assumed. For more information, see Map Implicit Relations Between Nested Schema Elements. The schema also specifies key and keyref constraints.

```
<xs:schema id="MyDataSet" xmlns=""
           xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">

 <xs:element name="MyDataSet" msdata:IsDataSet="true">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">

      <xs:element name="Order">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="OrderNumber" type="xs:integer" />
            <xs:element name="EmpNumber" type="xs:integer" />

            <xs:element name="OrderDetail">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="OrderNo" type="xs:integer" />
                  <xs:element name="ItemNo" type="xs:string" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>

  <xs:key name="OrderNumberKey"  >
    <xs:selector xpath=".//Order" />
    <xs:field xpath="OrderNumber" />
  </xs:key>

  <xs:keyref name="OrderNoRef" refer="OrderNumberKey">
    <xs:selector xpath=".//OrderDetail" />
    <xs:field xpath="OrderNo" />
  </xs:keyref>
 </xs:element>
</xs:schema>
```

The **DataSet** resulting from the XML Schema mapping process includes two tables:

```
Order(OrderNumber, EmpNumber, Order_Id)
OrderDetail(OrderNumber, ItemNumber, Order_Id)
```

The **DataSet** also includes the two relationships (one based on the **msdata:relationship** annotation and the other based on the key and keyref constraints) and various constraints. The following example shows the relations and constraints.

```
..RelationName: Order_OrderDetail
..ParentTable: Order
..ParentColumns: Order_Id
..ChildTable: OrderDetail
..ChildColumns: Order_Id
..ParentKeyConstraint: Constraint1
..ChildKeyConstraint: Order_OrderDetail
..Nested: True

..RelationName: OrderNoRef
..ParentTable: Order
..ParentColumns: OrderNumber
..ChildTable: OrderDetail
..ChildColumns: OrderNo
..ParentKeyConstraint: OrderNumberKey
..ChildKeyConstraint: OrderNoRef
..Nested: False

..ConstraintName: OrderNumberKey
..Type: UniqueConstraint
..Table: Order
..Columns: OrderNumber
..IsPrimaryKey: False

..ConstraintName: Constraint1
..Type: UniqueConstraint
..Table: Order
..Columns: Order_Id
..IsPrimaryKey: True

..ConstraintName: Order_OrderDetail
..Type: ForeignKeyConstraint
..Table: OrderDetail
..Columns: Order_Id
..RelatedTable: Order
..RelatedColumns: Order_Id

..ConstraintName: OrderNoRef
..Type: ForeignKeyConstraint
..Table: OrderDetail
..Columns: OrderNo
..RelatedTable: Order
..RelatedColumns: OrderNumber
```

If a keyref constraint referring to a nested table contains the **msdata:IsNested="true"** annotation, the **DataSet** will create a single nested relationship that is based on the keyref constraint and the related unique/key constraint.

See Also

Deriving DataSet Relational Structure from XML Schema (XSD)
ADO.NET Managed Providers and DataSet Developer Center

Inferring DataSet Relational Structure from XML

8/31/2018 • 2 minutes to read • Edit Online

The relational structure, or schema, of a DataSet is made up of tables, columns, constraints, and relations. When loading a DataSet from XML, the schema can be predefined, or it can be created, either explicitly or through inference, from the XML being loaded. For more information about loading the schema and contents of a DataSet from XML, see Loading a DataSet from XML and Loading DataSet Schema Information from XML.

If the schema of a DataSet is being created from XML, the preferred method is to explicitly specify the schema using either the XML Schema definition language (XSD) (as described in Deriving DataSet Relational Structure from XML Schema (XSD)) or the XML-Data Reduced (XDR). If no XML Schema or XDR schema is available in the XML, the schema of the DataSet can be inferred from the structure of the XML elements and attributes.

This section describes the rules for DataSet schema inference by showing XML elements and attributes and their structure, and the resulting inferred DataSet schema.

Not all attributes present in an XML document should be included in the inference process. Namespace-qualified attributes can include metadata that is important for the XML document but not for the DataSet schema. Using InferXmlSchema, you can specify namespaces to be ignored during the inference process. For more information, see Loading DataSet Schema Information from XML.

In This Section

Summary of the DataSet Schema Inference Process
Provides a high-level summary of the rules for inferring the schema of a DataSet from XML.

Inferring Tables
Describes the XML elements that are inferred as tables in a DataSet.

Inferring Columns
Describes the XML elements and attributes that are inferred as table columns.

Inferring Relationships
Describes the DataRelation and ForeignKeyConstraint objects created for nested, inferred tables.

Inferring Element Text
Describes the columns that are created for text in XML elements, and explains when text in XML elements is ignored.

Inference Limitations
Discusses the limitations of schema inference.

Related Sections

Using XML in a DataSet
Describes how the DataSet object interacts with XML data.

Deriving DataSet Relational Structure from XML Schema (XSD)
Describes the relational structure, or schema, of a DataSet that is created from XML Schema definition language (XSD) schema.

ADO.NET Overview
Describes the ADO.NET architecture and components and how to use them to access existing data sources and

manage application data.

See Also

ADO.NET Managed Providers and DataSet Developer Center

Summary of the DataSet Schema Inference Process

8/31/2018 • 2 minutes to read • Edit Online

The inference process first determines, from the XML document, which elements will be inferred as tables. From the remaining XML, the inference process determines the columns for those tables. For nested tables, the inference process generates nested DataRelation and ForeignKeyConstraint objects.

Following is a brief summary of inference rules:

- Elements that have attributes are inferred as tables.
- Elements that have child elements are inferred as tables.
- Elements that repeat are inferred as a single table.
- If the document, or root, element has no attributes, and no child elements that would be inferred as columns, it is inferred as a DataSet. Otherwise, the document element is inferred as a table.
- Attributes are inferred as columns.
- Elements that have no attributes or child elements, and that do not repeat, are inferred as columns.
- For elements that are inferred as nested tables within other elements that are also inferred as tables, a nested **DataRelation** is created between the two tables. A new, primary key column named **TableName_Id** is added to both tables and used by the **DataRelation**. A **ForeignKeyConstraint** is created between the two tables using the **TableName_Id** column.
- For elements that are inferred as tables and that contain text but have no child elements, a new column named **TableName_Text** is created for the text of each of the elements. If an element is inferred as a table and has text, but also has child elements, the text is ignored.

See Also

Inferring DataSet Relational Structure from XML
Loading a DataSet from XML
Loading DataSet Schema Information from XML
Using XML in a DataSet
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Inferring Tables

8/31/2018 • 2 minutes to read • Edit Online

When inferring a schema for a DataSet from an XML document, ADO.NET first determines which XML elements represent tables. The following XML structures result in a table for the **DataSet** schema:

- Elements with attributes
- Elements with child elements
- Repeating elements

Elements with Attributes

Elements that have attributes specified in them result in inferred tables. For example, consider the following XML:

```
<DocumentElement>
  <Element1 attr1="value1"/>
  <Element1 attr1="value2">Text1</Element1>
</DocumentElement>
```

The inference process produces a table named "Element1."

DataSet: DocumentElement

Table: Element1

ATTR1	ELEMENT1_TEXT
value1	
value2	Text1

Elements with Child Elements

Elements that have child elements result in inferred tables. For example, consider the following XML:

```
<DocumentElement>
  <Element1>
    <ChildElement1>Text1</ChildElement1>
  </Element1>
</DocumentElement>
```

The inference process produces a table named "Element1."

DataSet: DocumentElement

Table: Element1

CHILDELEMENT1
Text1

The document, or root, element result in an inferred table if it has attributes or child elements that are inferred as columns. If the document element has no attributes and no child elements that would be inferred as columns, the element is inferred as a **DataSet**. For example, consider the following XML:

```
<DocumentElement>
  <Element1>Text1</Element1>
  <Element2>Text2</Element2>
</DocumentElement>
```

The inference process produces a table named "DocumentElement."

DataSet: NewDataSet

Table: DocumentElement

ELEMENT1	ELEMENT2
Text1	Text2

Alternatively, consider the following XML:

```
<DocumentElement>
  <Element1 attr1="value1" attr2="value2"/>
</DocumentElement>
```

The inference process produces a **DataSet** named "DocumentElement" that contains a table named "Element1."

DataSet: DocumentElement

Table: Element1

ATTR1	ATTR2
value1	value2

Repeating Elements

Elements that repeat result in a single inferred table. For example, consider the following XML:

```
<DocumentElement>
  <Element1>Text1</Element1>
  <Element1>Text2</Element1>
</DocumentElement>
```

The inference process produces a table named "Element1."

DataSet: DocumentElement

Table: Element1

ELEMENT1_TEXT
Text1
Text2

See Also

Inferring DataSet Relational Structure from XML
Loading a DataSet from XML
Loading DataSet Schema Information from XML
Using XML in a DataSet
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Inferring Columns

8/31/2018 • 2 minutes to read • Edit Online

After ADO.NET has determined from an XML document which elements to infer as tables for a DataSet, it then infers the columns for those tables. ADO.NET 2.0 introduced a new schema inference engine that infers a strongly typed data type for each **simpleType** element. In previous versions, the data type of an inferred **simpleType** element was always **xsd:string**.

Migration and Backward Compatibility

The **ReadXml** method takes an argument of type **InferSchema**. This argument allows you to specify inference behavior compatible with previous versions. The available values for the **InferSchema** enumeration are shown in the following table.

InferSchema
Provides backward compatibility by always inferring a simple type as String.

InferTypedSchema
Infers a strongly typed data type. Throws an exception if used with a DataTable.

IgnoreSchema
Ignores any inline schema and reads data into the existing DataSet schema.

Attributes

As defined in Inferring Tables, an element with attributes will be inferred as a table. The attributes of that element will then be inferred as columns for the table. The **ColumnMapping** property of the columns will be set to **MappingType.Attribute**, to ensure that the column names will be written as attributes if the schema is written back to XML. The values of the attributes are stored in a row in the table. For example, consider the following XML:

```
<DocumentElement>
  <Element1 attr1="value1" attr2="value2"/>
</DocumentElement>
```

The inference process will produce a table named **Element1** with two columns, **attr1** and **attr2**. The **ColumnMapping** property of both columns will be set to **MappingType.Attribute**.

DataSet: DocumentElement

Table: Element1

ATTR1	ATTR2
value1	value2

Elements Without Attributes or Child Elements

If an element has no child elements or attributes, it will be inferred as a column. The **ColumnMapping** property of the column will be set to **MappingType.Element**. The text for child elements is stored in a row in the table. For example, consider the following XML:

```
<DocumentElement>
  <Element1>
    <ChildElement1>Text1</ChildElement1>
    <ChildElement2>Text2</ChildElement2>
  </Element1>
</DocumentElement>
```

The inference process will produce a table named **Element1** with two columns, **ChildElement1** and **ChildElement2**. The **ColumnMapping** property of both columns will be set to **MappingType.Element**.

DataSet: DocumentElement

Table: Element1

CHILDELEMENT1	CHILDELEMENT2
Text1	Text2

See Also

Inferring DataSet Relational Structure from XML
Loading a DataSet from XML
Loading DataSet Schema Information from XML
Using XML in a DataSet
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Inferring Relationships

8/31/2018 • 2 minutes to read • Edit Online

If an element that is inferred as a table has a child element that is also inferred as a table, a DataRelation will be created between the two tables. A new column with a name of **ParentTableName_Id** will be added to both the table created for the parent element, and the table created for the child element. The **ColumnMapping** property of this identity column will be set to **MappingType.Hidden**. The column will be an auto-incrementing primary key for the parent table, and will be used for the **DataRelation** between the two tables. The data type of the added identity column will be **System.Int32**, unlike the data type of all other inferred columns, which is **System.String**. A ForeignKeyConstraint with **DeleteRule** = **Cascade** will also be created using the new column in both the parent and child tables.

For example, consider the following XML:

```
<DocumentElement>
  <Element1>
    <ChildElement1 attr1="value1" attr2="value2"/>
    <ChildElement2>Text2</ChildElement2>
  </Element1>
</DocumentElement>
```

The inference process will produce two tables: **Element1** and **ChildElement1**.

The **Element1** table will have two columns: **Element1_Id** and **ChildElement2**. The **ColumnMapping** property of the **Element1_Id** column will be set to **MappingType.Hidden**. The **ColumnMapping** property of the **ChildElement2** column will be set to **MappingType.Element**. The **Element1_Id** column will be set as the primary key of the **Element1** table.

The **ChildElement1** table will have three columns: **attr1**, **attr2** and **Element1_Id**. The **ColumnMapping** property for the **attr1** and **attr2** columns will be set to **MappingType.Attribute**. The **ColumnMapping** property of the **Element1_Id** column will be set to **MappingType.Hidden**.

A **DataRelation** and **ForeignKeyConstraint** will be created using the **Element1_Id** columns from both tables.

DataSet: DocumentElement

Table: Element1

ELEMENT1_ID	CHILDELEMENT2
0	Text2

Table: ChildElement1

ATTR1	ATTR2	ELEMENT1_ID
value1	value2	0

DataRelation: Element1_ChildElement1

ParentTable: Element1

ParentColumn: Element1_Id

ChildTable: ChildElement1

ChildColumn: Element1_Id

Nested: True

ForeignKeyConstraint: Element1_ChildElement1

Column: Element1_Id

ParentTable: Element1

ChildTable: ChildElement1

DeleteRule: Cascade

AcceptRejectRule: None

See Also

Inferring DataSet Relational Structure from XML
Loading a DataSet from XML
Loading DataSet Schema Information from XML
Nesting DataRelations
Using XML in a DataSet
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Inferring Element Text

8/31/2018 • 2 minutes to read • Edit Online

If an element contains text and has no child elements to be inferred as tables (such as elements with attributes or repeated elements), a new column with the name **TableName_Text** will be added to the table that is inferred for the element. The text contained in the element will be added to a row in the table and stored in the new column. The **ColumnMapping** property of the new column will be set to **MappingType.SimpleContent**.

For example, consider the following XML.

```
<DocumentElement>
  <Element1 attr1="value1">Text1</Element1>
</DocumentElement>
```

The inference process will produce a table named **Element1** with two columns: **attr1** and **Element1_Text**. The **ColumnMapping** property of the **attr1** column will be set to **MappingType.Attribute**. The **ColumnMapping** property of the **Element1_Text** column will be set to **MappingType.SimpleContent**.

DataSet: DocumentElement

Table: Element1

ATTR1	ELEMENT1_TEXT
value1	Text1

If an element contains text, but also has child elements that contain text, a column will not be added to the table to store the text contained in the element. The text contained in the element will be ignored, while the text in the child elements is included in a row in the table. For example, consider the following XML.

```
<Element1>
  Text1
  <ChildElement1>Text2</ChildElement1>
  Text3
</Element1>
```

The inference process will produce a table named **Element1** with one column named **ChildElement1**. The text for the **ChildElement1** element will be included in a row in the table. The other text will be ignored. The **ColumnMapping** property of the **ChildElement1** column will be set to **MappingType.Element**.

DataSet: DocumentElement

Table: Element1

CHILDELEMENT1
Text2

See Also

Inferring DataSet Relational Structure from XML

Loading a DataSet from XML
Loading DataSet Schema Information from XML
Using XML in a DataSet
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Inference Limitations

8/31/2018 • 2 minutes to read • Edit Online

The process of inferring a DataSet schema from XML can result in different schemas depending on the XML elements in each document. For example, consider the following XML documents.

Document1:

```
<DocumentElement>
  <Element1>Text1</Element1>
  <Element1>Text2</Element1>
</DocumentElement>
```

Document2:

```
<DocumentElement>
  <Element1>Text1</Element1>
</DocumentElement>
```

For "Document1," the inference process produces a **DataSet** named "DocumentElement" and a table named "Element1," because "Element1" is a repeating element.

DataSet: DocumentElement

Table: Element1

ELEMENT1_TEXT
Text1
Text2

However, for "Document2," the inference process produces a **DataSet** named "NewDataSet" and a table named "DocumentElement." "Element1" is inferred as a column because it has no attributes and no child elements.

DataSet: NewDataSet

Table: DocumentElement

ELEMENT1
Text1

These two XML documents may have been intended to produce the same schema, but the inference process produces very different results based on the elements contained in each document.

To avoid the discrepancies that can occur when generating schema from an XML document, we recommend that you explicitly specify a schema using XML Schema definition language (XSD) or XML-Data Reduced (XDR) when loading a **DataSet** from XML. For more information about explicitly specifying a **DataSet** schema with XML Schema, see Deriving DataSet Relational Structure from XML Schema (XSD).

See Also

Inferring DataSet Relational Structure from XML
Loading a DataSet from XML
Loading DataSet Schema Information from XML
Using XML in a DataSet
DataSets, DataTables, and DataViews
ADO.NET Managed Providers and DataSet Developer Center

Consuming a DataSet from an XML Web Service

8/31/2018 • 6 minutes to read • Edit Online

The DataSet was architected with a disconnected design, in part to facilitate the convenient transport of data over the Internet. The **DataSet** is "serializable" in that it can be specified as an input to or output from XML Web services without any additional coding required to stream the contents of the **DataSet** from an XML Web service to a client and back. The **DataSet** is implicitly converted to an XML stream using the DiffGram format, sent over the network, and then reconstructed from the XML stream as a **DataSet** on the receiving end. This gives you a very simple and flexible method for transmitting and returning relational data using XML Web services. For more information about the DiffGram format, see DiffGrams.

The following example shows how to create an XML Web service and client that use the **DataSet** to transport relational data (including modified data) and resolve any updates back to the original data source.

> **NOTE**
>
> We recommend that you always consider security implications when creating an XML Web service. For information on securing an XML Web service, see Securing XML Web Services Created Using ASP.NET.

To create an XML Web service that returns and consumes a DataSet

1. Create the XML Web service.

   In the example, an XML Web service is created that returns data, in this case a list of customers from the **Northwind** database, and receives a **DataSet** with updates to the data, which the XML Web service resolves back to the original data source.

   The XML Web service exposes two methods: **GetCustomers**, to return the list of customers, and **UpdateCustomers**, to resolve updates back to the data source. The XML Web service is stored in a file on the Web server called DataSetSample.asmx. The following code outlines the contents of DataSetSample.asmx.

```
<% @ WebService Language = "vb" Class = "Sample" %>
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Web.Services

<WebService(Namespace:="http://microsoft.com/webservices/")> _
Public Class Sample

Public connection As SqlConnection = New SqlConnection("Data Source=(local);Integrated
Security=SSPI;Initial Catalog=Northwind")

  <WebMethod( Description := "Returns Northwind Customers", EnableSession := False )> _
  Public Function GetCustomers() As DataSet
    Dim adapter As SqlDataAdapter = New SqlDataAdapter( _
      "SELECT CustomerID, CompanyName FROM Customers", connection)

    Dim custDS As DataSet = New DataSet()
    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey
    adapter.Fill(custDS, "Customers")

    Return custDS
  End Function

  <WebMethod( Description := "Updates Northwind Customers", EnableSession := False )> _
  Public Function UpdateCustomers(custDS As DataSet) As DataSet
    Dim adapter As SqlDataAdapter = New SqlDataAdapter()

    adapter.InsertCommand = New SqlCommand( _
      "INSERT INTO Customers (CustomerID, CompanyName) " & _
      "Values(@CustomerID, @CompanyName)", connection)
    adapter.InsertCommand.Parameters.Add( _
      "@CustomerID", SqlDbType.NChar, 5, "CustomerID")
    adapter.InsertCommand.Parameters.Add( _
      "@CompanyName", SqlDbType.NChar, 15, "CompanyName")

    adapter.UpdateCommand = New SqlCommand( _
      "UPDATE Customers Set CustomerID = @CustomerID, " & _
      "CompanyName = @CompanyName WHERE CustomerID = " & _
      @OldCustomerID", connection)
    adapter.UpdateCommand.Parameters.Add( _
      "@CustomerID", SqlDbType.NChar, 5, "CustomerID")
    adapter.UpdateCommand.Parameters.Add( _
      "@CompanyName", SqlDbType.NChar, 15, "CompanyName")

    Dim parameter As SqlParameter = _
      adapter.UpdateCommand.Parameters.Add( _
      "@OldCustomerID", SqlDbType.NChar, 5, "CustomerID")
    parameter.SourceVersion = DataRowVersion.Original

    adapter.DeleteCommand = New SqlCommand( _
      "DELETE FROM Customers WHERE CustomerID = @CustomerID", _
      connection)
    parameter = adapter.DeleteCommand.Parameters.Add( _
      "@CustomerID", SqlDbType.NChar, 5, "CustomerID")
    parameter.SourceVersion = DataRowVersion.Original

    adapter.Update(custDS, "Customers")

    Return custDS
  End Function
End Class
```

```
<% @ WebService Language = "C#" Class = "Sample" %>
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web.Services;

[WebService(Namespace="http://microsoft.com/webservices/")]
public class Sample
{
  public SqlConnection connection = new SqlConnection("Data Source=(local);Integrated
Security=SSPI;Initial Catalog=Northwind");

  [WebMethod( Description = "Returns Northwind Customers", EnableSession = false )]
  public DataSet GetCustomers()
  {
    SqlDataAdapter adapter = new SqlDataAdapter(
      "SELECT CustomerID, CompanyName FROM Customers", connection);

    DataSet custDS = new DataSet();
    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
    adapter.Fill(custDS, "Customers");

    return custDS;
  }

  [WebMethod( Description = "Updates Northwind Customers",
    EnableSession = false )]
  public DataSet UpdateCustomers(DataSet custDS)
  {
    SqlDataAdapter adapter = new SqlDataAdapter();

    adapter.InsertCommand = new SqlCommand(
      "INSERT INTO Customers (CustomerID, CompanyName) " +
      "Values(@CustomerID, @CompanyName)", connection);
    adapter.InsertCommand.Parameters.Add(
      "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    adapter.InsertCommand.Parameters.Add(
      "@CompanyName", SqlDbType.NChar, 15, "CompanyName");

    adapter.UpdateCommand = new SqlCommand(
      "UPDATE Customers Set CustomerID = @CustomerID, " +
      "CompanyName = @CompanyName WHERE CustomerID = " +
      "@OldCustomerID", connection);
    adapter.UpdateCommand.Parameters.Add(
      "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    adapter.UpdateCommand.Parameters.Add(
      "@CompanyName", SqlDbType.NChar, 15, "CompanyName");
    SqlParameter parameter = adapter.UpdateCommand.Parameters.Add(
      "@OldCustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand = new SqlCommand(
    "DELETE FROM Customers WHERE CustomerID = @CustomerID",
     connection);
    parameter = adapter.DeleteCommand.Parameters.Add(
      "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.Update(custDS, "Customers");

    return custDS;
  }
}
```

In a typical scenario, the **UpdateCustomers** method would be written to catch optimistic concurrency violations. For simplicity, the example does not include this. For more information about optimistic

concurrency, see Optimistic Concurrency.

2. Create an XML Web service proxy.

   Clients of the XML Web service require a SOAP proxy in order to consume the exposed methods. You can have Visual Studio generate this proxy for you. By setting a Web reference to an existing Web service from within Visual Studio, all the behavior described in this step occurs transparently. If you want to create the proxy class yourself, continue with this discussion. In most circumstances, however, using Visual Studio to create the proxy class for the client application is sufficient.

   A proxy can be created using the Web Services Description Language Tool. For example, if the XML Web service is exposed at the URL http://myserver/data/DataSetSample.asmx, issue a command such as the following to create a Visual Basic .NET proxy with a namespace of **WebData.DSSample** and store it in the file sample.vb.

   ```
   wsdl /l:VB -out:sample.vb http://myserver/data/DataSetSample.asmx /n:WebData.DSSample
   ```

   To create a C# proxy in the file sample.cs, issue the following command.

   ```
   wsdl -l:CS -out:sample.cs http://myserver/data/DataSetSample.asmx -n:WebData.DSSample
   ```

   The proxy can then be compiled as a library and imported into the XML Web service client. To compile the Visual Basic .NET proxy code stored in sample.vb as sample.dll, issue the following command.

   ```
   vbc -t:library -out:sample.dll sample.vb -r:System.dll -r:System.Web.Services.dll -r:System.Data.dll -
   r:System.Xml.dll
   ```

   To compile the C# proxy code stored in sample.cs as sample.dll, issue the following command.

   ```
   csc -t:library -out:sample.dll sample.cs -r:System.dll -r:System.Web.Services.dll -r:System.Data.dll -
   r:System.Xml.dll
   ```

3. Create an XML Web service client.

   If you want to have Visual Studio generate the Web service proxy class for you, simply create the client project, and, in the Solution Explorer window, right-click the project, click **Add Web Reference**, and select the Web service from the list of available Web services (this may require supplying the address of the Web service endpoint, if the Web service isn't available within the current solution, or on the current computer.) If you create the XML Web service proxy yourself (as described in the previous step), you can import it into your client code and consume the XML Web service methods. The following sample code imports the proxy library, calls **GetCustomers** to get a list of customers, adds a new customer, and then returns a **DataSet** with the updates to **UpdateCustomers**.

   Notice that the example passes the **DataSet** returned by **DataSet.GetChanges** to **UpdateCustomers** because only modified rows need to be passed to **UpdateCustomers**. **UpdateCustomers** returns the resolved **DataSet**, which you can then **Merge** into the existing **DataSet** to incorporate the resolved changes and any row error information from the update. The following code assumes that you have used Visual Studio to create the Web reference, and that you have renamed the Web reference to DsSample in the **Add Web Reference** dialog box.

```
Imports System
Imports System.Data

Public Class Client

  Public Shared Sub Main()
    Dim proxySample As New DsSample.Sample ()  ' Proxy object.
    Dim customersDataSet As DataSet = proxySample.GetCustomers()
    Dim customersTable As DataTable = _
      customersDataSet.Tables("Customers")

    Dim rowAs DataRow = customersTable.NewRow()
    row("CustomerID") = "ABCDE"
    row("CompanyName") = "New Company Name"
    customersTable.Rows.Add(row)

    Dim updateDataSet As DataSet = _
      proxySample.UpdateCustomers(customersDataSet.GetChanges())

    customersDataSet.Merge(updateDataSet)
    customersDataSet.AcceptChanges()
  End Sub
End Class
```

```
using System;
using System.Data;

public class Client
{
  public static void Main()
  {
    Sample proxySample = new DsSample.Sample();  // Proxy object.
    DataSet customersDataSet = proxySample.GetCustomers();
    DataTable customersTable = customersDataSet.Tables["Customers"];

    DataRow row = customersTable.NewRow();
    row["CustomerID"] = "ABCDE";
    row["CompanyName"] = "New Company Name";
    customersTable.Rows.Add(row);

    DataSet updateDataSet = new DataSet();

    updateDataSet =
      proxySample.UpdateCustomers(customersDataSet.GetChanges());

    customersDataSet.Merge(updateDataSet);
    customersDataSet.AcceptChanges();
  }
}
```

If you decide to create the proxy class yourself, you must take the following extra steps. To compile the sample, supply the proxy library that was created (sample.dll) and the related .NET libraries. To compile the Visual Basic .NET version of the sample, stored in the file client.vb, issue the following command.

```
vbc client.vb -r:sample.dll -r:System.dll -r:System.Data.dll -r:System.Xml.dll -
r:System.Web.Services.dll
```

To compile the C# version of the sample, stored in the file client.cs, issue the following command.

```
csc client.cs -r:sample.dll -r:System.dll -r:System.Data.dll -r:System.Xml.dll -
r:System.Web.Services.dll
```

See Also

ADO.NET
DataSets, DataTables, and DataViews
DataTables
Populating a DataSet from a DataAdapter
Updating Data Sources with DataAdapters
DataAdapter Parameters
Web Services Description Language Tool (Wsdl.exe)
ADO.NET Managed Providers and DataSet Developer Center