**Purpose:**

* The goal of this DAL is to provide strongly typed database access. You can fall back on preparing your own parameters and manually reading the resulting rows, but the intent of this project is to allow you to pass a class into a SQL query and get classes or scalar types back out in a single call. Examples of using this class can be found at the end of the document. This DAL provides four providers: SQL, ODBC, read only CSV, and an in memory objects provider. Unless otherwise specified, this document is specific to the SQL and ODBC providers.

**Properties:**

* SqlConnection Connection
  + The connection object to use if the ConnectionString property is not set. If not null, this takes precedence over the ConnectionString property.
  + Defaults to null.
* String ConnectionString
  + Will be used if the SqlConnection is null for database queries. Creates and disposes of a new connection for each query.
  + Defaults to null.
* Boolean PrefixDirection
  + Default value for the prefix direction setting used by queries. An @ is always prefixed, even if set to false.
  + Defaults to true.
* String InputParameterPrefix
  + The string to be prefixed to input parameter names.
  + Defaults to “@i”.
* String OutputParameterPrefix
  + The string to be prefixed to output parameter names.
  + Defaults to “@o”.
* int CommandTimeout
  + The default timeout value in seconds for SQL commands.
  + Defaults to 30.
* Boolean IsStoredProcedure
  + Default value for the stored procedure setting used by queries.
  + Defaults to true.
* Boolean PopulateDefaultValues
  + Sets whether or not model properties not returned by a query should be defaulted to the value defined in the DALDefaultValue attribute.
  + Defaults to false.
* Object Input
  + The object to be used as input to the query. This can be any class, an anonymous type, or an enumeration of preprpeared SqlParameter objects.
  + Defaults to null (no parameters).
* Dictionary<String, Object> OutputValues
  + Any output parameters that result from a query will be populated in this dictionary. This is read only.
* int? ExpectedAffectedRows
  + Any NonQuery action will raise an event if the number of affected rows does not match this property.
  + Defaults to null (event is disabled).
* List<String> CustomSqlErrors
  + Messages returned by the RAISERROR method on the SQL server from the last query. This is read only.
* Boolean HasCustomError
  + Returns true if RAISERROR messages were thrown during the last call. This is read only.
* TraceLevel TraceOutputLevel
  + The level at which to output trace event messages.
  + Defaults to NONE.

**Query Methods:**

There are eight types of query methods:

* ExecuteRead
  + Executes a read query against the SQL server.
* ExecuteReadSingle
  + Returns the first row of an ExecuteRead. This does not set rowcount to 1, or otherwise ensure that only one row will be returned from the database. The entire result set will come back to the c#. This is merely a convenience function.
* ExecuteSetRead
  + Performs an ExecuteRead against the SQL server that returns multiple tables.
* ExecuteRelatedSetRead
  + Executes a set read and relates the tables together using the provided relationship objects.
* ExecuteCrossProviderSetRead
  + Executes a set read using multiple data providers.
* ExecuteScalar
  + Executes a scalar query against the SQL server.
* ExecuteScalarEnumeration (generic only)
  + Executes a read query against the SQL server and returns the value of the first column of each row cast to the type T.
* ExecuteNonQuery
  + Executes a non query statement against the SQL server.
* Each of the read query methods has a generic method overload which converts the returned data row(s) into the type passed.
* The ExecuteSetRead, ExecuteRelatedSetRead, and ExecuteCrossProviderSetRead methods offer up to 16 overloads for generic parameters and should be passed in the order that they are returned from the SQL statement.
  + These methods return a DALTuple type which contains table properties which are of the type List<T> corresponding to each of the type parameters passed to the ExecuteSetRead call.
  + All Execute(Related/CrossProvider)SetRead operations have an overload which accepts an array of Types as a parameter. This allows you to get more than 16 tables back by passing in a type array of types in the order they will be returned. This method can only return a list of object lists, so each object list must have .OfType<T>().ToList() called on it with the type you expect each object to be in order to get strongly typed models back out of it. If using this overload with a related set read with known compile time types, the lists to be populated into must be object lists.
* The five types of Read methods (ExecuteRead, ExecuteSetRead, ExecuteReadSingle, ExecuteRelatedSetRead, ExecuteCrossProviderSetRead) can take in the type Object as a generic parameter. This will cause the DAL to construct a class a runtime which represents the return set. This can be used for easy serialization without having to write classes at compile time. The ExecuteSetRead method which accepts an array of types can also take advantage of this by putting Object in the array where necessary. See the examples below for how to preform predetermined in memory joins on result sets of this nature.
* If the SQL statement does not require any input or output parameters, simply do not set the Input property before executing the query.

**Events:**

Events can be subscribed to on each instance of a SqlDBAccess or ODBCAccess class or on the DALEvents static class. The static class’ events will be thrown for all SqlDBAccess and ODBCAccess instances.

**Available Events:**

* OnException
  + Fired whenever any exception occurs during a query operation.
  + Delegate: SqlDBAccess or ODBCAccess sender, DALExceptionEventArgs e
* OnSqlException
  + Fired whenever a SqlException occurs during a query operation.
  + Delegate: SqlDBAccess or ODBCAccess sender, DALSqlExceptionEventArgs e
* OnAffectedRowsMismatch
  + Fired whenever the ExpectedAffectedRows property is set and the number of affected rows from a NonQuery operation does not match the value of the property.
  + Delegate SqlDBAccess or ODBCAccess sender, DALAffectedRowsMismatchEventArgs e
* OnDALTrace (static only)
  + Fired whenever a trace event occurs at a level being listened to.
  + Delegate: IDBAccess sender, String message
* OnQueryComplete
  + Fired whenever a query operation finishes. This event provides the time it took for the query to complete.
  + Delegate: SqlDBAccess or ODBCAccess sender, DALQueryCompletedEventArgs e

Each event arg object derives from the DALEventArgs type which has the following properties:

* String QueryString
  + The statement that was executing when the event was thrown.
* IDBConnection Connection
  + The connection object being used when the event was thrown.
* IDBTransaction Transaction
  + If the execution was happening in a transaction, this will be set to the transaction object.
* int Timeout
  + The command timeout.
* List<IDataParameter> InputParameters
  + The SQL parameters that were used for the query.
* String QueryMethod
  + The type of query run. ExecuteRead, ExecuteReadSingle, ExecuteSetRead, ExecuteRelatedSetRead, ExecuteCrossProviderSetRead, ExecuteScalar, ExecuteScalarEnumeration or ExecuteNonQuery.
* String InputModelType
  + If an input class was passed, the type of class that was used to generate the SQL parameters.
* Object Input
  + If an input class was passed, this is the input object.

The DALExceptionEventArgs contains an additional property which is the exception that was thrown.

The DALQueryCompletedEventArgs contains on additional TimeSpan property which is the time the query took to complete.

The DALSqlExceptionEventArgs contains three additional properties:

* SqlException Exception
  + The SqlException that was thrown.
* Boolean HasCustomErrors
  + Boolean value indicating whether or not custom error messages were returned the by SQL server.
* List<String> CustomErrors
  + The list of custom errors returned by the call.

The DALAffectedRowsMismatchEventArgs contains two additional properties:

* int  AffectedRows
  + The actual number of affected rows.
* int  ExpectedAffectedRows - The expected number of affected rows.

**Modifier Attributes:**

* DALParameterDirection
  + Properties:
    - Direction – Specifies the direction of the SQL parameter. Defaults to input.
* DALWriteStringFormat
  + String.Format to be applied when writing a value into this property. This attribute may only be used on Strings.
  + Properties:
    - Format – Defaults to null.
* DALReadStringFormat
  + String.Format to be applied when reading this property into a SQL Parameter.
  + Properties:
    - Format – Defaults to null.
* DALIgnore
  + Instructs the DAL to ignore this property when reading from and writing to model properties.
* DALSQLParameterNameAttribute
  + Properties:
    - Name – Defaults to the property name. Instructs the DAL to create a different SQL parameter name for this property. When populating the model after a DB call, the DAL will use the column matching the DALSqlParameterName if provided.
    - This parameter name must be unique across the model, including the names of other properties which do not have the DALIgnore attribute.
* DALDefaultValue
  + Properties:
    - Value – The default value to apply to this property. Default values will only be applied if explicitly told to do so if set on the class.
    - This object passed in for the Value must be valid for the property type and cannot be null.

A model class can also contain other class instances as a parameter as long as the parameter is not the same type as the parent class type, does not derive from it, and does not contain any properties that are the same type as the outer model or which derive from it. All properties from all model bases will be flattened into one parameter list. Nested ModelBase properties must therefore all be unique. (IQuickPopulate models are exempt from these rules since the Populate routine is custom). The default Populate method will also populate into nested ModelBase properties.

The DALIgnore attribute can also be achieved by having only a getter or setter on a property. Whenever the DAL reads or writes, if the property does not have the appropriate accessor, it will be ignored.

**IQuickPopulate:**

**Purpose:**

* When one of your models which derives IQuickPopulate is used as a generic parameter to a query method, the DAL will use the DALPopulate method on your model instead of the normal reflection methods. You must write the DALPopulate routine in each model which inherits from this. Here is a basic example of a quick populate class:

    public class OutputModelQuickPopulate : IQuickPopulate

    {

        public int CustomerId { get; set; }

        public String FirstName { get; set; }

        public String LastName { get; set; }

        public int NumberOfOrders { get; set; }

        public Decimal AmountSpent { get; set; }

        public String CustomerSince { get; set; }

        public String LastSeen { get; set; }

        public List<T> DALPopulate<T>(List<Object[]> dataRows, Dictionary<String, int> indexes) where T : class, new()

        {

var ret = new List<OutputModelQuickPopulate>(dataRows.Count);  
 for (int i = 0; i < dataRows.Count; i++)  
 {  
 var dr = dataRows[i];

             ret.Add(new OutputModelQuickPopulate  
 {  
 CustomerId     = (int)dr[0],

             FirstName      = "Customer First Name",

             LastName       = "Customer Last Name",

              NumberOfOrders = (int)dr[1],

              AmountSpent    = (Decimal)dr[2],

             CustomerSince  = String.Format("{0:MM/dd/yyyy}", dr[3]),

              LastSeen       = String.Format("{0:MM/dd/yyyy}", dr[4])  
 });  
 }  
  
 return ret as List<T>;

        }

    }

* This is quicker than the normal reflection routines and also allows you to customize the way models are populated.
* Take special note of the last line where the return happens. You must use the “as” keyword to return the list back to the DAL.
* For clarity in the example I did not properly check for null in each column. Since this Populate routine is no different than regular DB access any normal checks you do there should be incorporated here. There is an extension method CastToT<>() that will do the appropriate checks (though this will be slower than doing the check manually on each line).
* I statically set first name and last name and did a String.Format on the date columns to highlight that any property attributes you wish to apply should be handled by the QuickPopulate function; you do not get this functionality in a QuickPopulate model.
* The indexes parameter can be used to lookup the index of a column name in the dr parameter if you do not know the column index.

**Table parameters:**An enumeration of a custom class as a property of an input class will be turned into a DataTable. The example below is for an IQuickRead table. Normally a table parameter can be prepared based off any class without any special work from the programmer.

**IQuickRead:**

**Purpose:**

This is similar in purpose to the IQuickPopulate interface. Using an IQuickRead object allows you to write a custom function to return an Object array from your table type. Like the IQuickPopulate, no property attributes are available, and the only function called to read from the table is the one provided. You must define the ToObjectArray method which returns the values to be put into the DataTable parameter as well as the GetColumnNamesTypes method. This method should return a dictionary of names and types which represent the columns of the objects returned in the ToObjectArray method.

**Validations:**

* The length of the objects returned by ToObjectArray() and GetColumnNamesTypes() must be the same.
* Types returned by GetColumnNamesAndTypes() should not be nullable value types even if the type in the class is nullable. The DataTable object cannot have nullable value type columns. Use the normal value type instead as shown below in the example.
* The DAL will create your quick read type to validate it. Be sure any nullable properties in the ToObjectArray function are properly checked for nulls if any functions are being called on them in the ToObjectArray method.

**Example:**

    public class UDTableQuickReadExample : UDTableQuickRead

    {

        public int? Int1 { get; set; }

        public int? Int2 { get; set; }

        public int? Int3 { get; set; }

        public int? Int4 { get; set; }

        public int? Int5 { get; set; }

        public Object[] ToObjectArray()

        {

            return new Object[] { Int1, Int2, Int3, Int4, Int5 };

        }  
  
 public Dictionary<String, Type> GetColumnNamesTypes()

{

return new Dictionary<String, Type> { {"Int1", typeof(int)}, {"Int2", typeof(int)}, {"Int3", typeof(int)}, {"Int4", typeof(int)},   
 {"Int5", typeof(int)} };

}

    }

**Exceptions:**

There are several exceptions which can be generated by the DAL class. These cannot be ignored, as they will prevent the class from being used at all. One of the exceptions below will be thrown depending on the problem.

Model exceptions:

* ModelPropertyInvalidException
  + Thrown if the property type is an enumeration of something other than a class or a Byte array.
  + Thrown if there are two properties in the model whose names are the same when ignoring case.
* ModelPropertyMisconfiguredException
  + Thrown if the model contains properties whose attributes are not configured properly.
* ModelPropertyColumnMismatchException
  + Thrown if a value from the SQL query cannot be assigned to the model property type.
* ModelPropertyNotNullableException
  + Thrown if there is an attempt to assign a null value into a model property that is not nullable.
* TableQuickReadMisconfiguredException
  + Thrown when the contract functions of an IQuickRead table class are not returning as expected.
* ArgumentException
  + Thrown if a null SqlConnection is passed without having first set the ConnectionString parameter.

Transaction exceptions:

* TransactionInProgressException
  + Thrown when a transaction is started with one already in progress.
* TransactionNotInProgressException
  + Thrown when an action is attempted against a transaction when none is in progress.
* TransactionSavePointNotFoundException
  + Thrown when an action is attempted on a save point which does not exist.
* TransactionSavePointAlreadyExistsException
  + Thrown when a save is attempted with a save point name that already exists.
* TransactionNoSavePointsFoundException
  + Thrown when RollbackToLastSavePoint is called and there are no save points.

ExecuteRelatedSetRead exceptions:

* DALRelationshipMisconfiguredException
  + Thrown when a custom relationship is passed with the same parent and child table index.
* DALRelationshipParentPropertyMissingException
  + Thrown when the parent table does not have the specified property for child objects.
* DALRelationshipParentPropertyNotAListException
  + Thrown when the parent table's children property is not a list.
* DALRelationshipParentPropertyListIncorrectTypeException
  + Thrown when the parent table's children property list is of the incorrect type.

**Examples:**

In order to be used by the DAL a class must either be public or internal, and any properties that should be used must also be public or internal. If internal properties or classes are to be used, the AssemblyInfo.cs file for the project must also contain the line. Note that anonymous types are internal.

[assembly: InternalsVisibleTo("FastDynamicAccessAccessors")]

**ExecuteRead**

The following is a simple ExecuteRead example showing some orders. First the return class:

    public class OrderDetail

    {

        public int OrderID { get; set; }

        public int ProductID { get; set; }

        public Decimal UnitPrice { get; set; }

        public short Quantity { get; set; }

        public float Discount { get; set; }

    }

Then the c# code to execute:

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT TOP 10 \* FROM [Order Details] WITH (NOLOCK)";

    List<OrderDetail> orders = db.ExecuteRead<OrderDetail>();

And finally the resulting return set of OrderDetail objects:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OrderID | ProductID | UnitPrice | Quantity | Discount |
| 10248 | 11 | 14.0000 | 12 | 0 |
| 10248 | 42 | 9.8000 | 10 | 0 |
| 10248 | 72 | 34.8000 | 5 | 0 |
| 10249 | 14 | 18.6000 | 9 | 0 |
| 10249 | 51 | 42.4000 | 40 | 0 |
| 10250 | 41 | 7.7000 | 10 | 0 |
| 10250 | 51 | 42.4000 | 35 | 0.15 |
| 10250 | 65 | 16.8000 | 15 | 0.15 |
| 10251 | 22 | 16.8000 | 6 | 0.05 |
| 10251 | 57 | 15.6000 | 15 | 0.05 |

Now let’s modify the query to filter on a discount amount. There are three ways to pass in the parameters for each of the query types:

* SQL parameters

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT TOP 10 \* FROM [Order Details] WITH (NOLOCK) WHERE Discount = @iDiscount";  
 **db.Input = new List<SqlParameter> { new SqlParameter("@iDiscount", .25f) };**

    List<OrderDetail> orders = db.ExecuteRead<OrderDetail>();

* Input class

    public class DiscountFilter

    {

        public float Discount { get; set; }

    }

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT TOP 10 \* FROM [Order Details] WITH (NOLOCK) WHERE Discount = @iDiscount";  
 **db.Input = new DiscountFilter { Discount = .25f };**

    List<OrderDetail> orders = db.ExecuteRead<OrderDetail>();

* Anonymous type

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT TOP 10 \* FROM [Order Details] WITH (NOLOCK) WHERE Discount = @iDiscount";  
 **db.Input = new { Discount = .25f };**

    List<OrderDetail> orders = db.ExecuteRead<OrderDetail>();

All three of the above methods will produce the following output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OrderID | ProductID | UnitPrice | Quantity | Discount |
| 10260 | 41 | 7.7000 | 16 | 0.25 |
| 10260 | 62 | 39.4000 | 15 | 0.25 |
| 10260 | 70 | 12.0000 | 21 | 0.25 |
| 10263 | 16 | 13.9000 | 60 | 0.25 |
| 10263 | 30 | 20.7000 | 60 | 0.25 |
| 10263 | 74 | 8.0000 | 36 | 0.25 |
| 10279 | 17 | 31.2000 | 15 | 0.25 |
| 10284 | 27 | 35.1000 | 15 | 0.25 |
| 10284 | 60 | 27.2000 | 20 | 0.25 |
| 10284 | 67 | 11.2000 | 5 | 0.25 |

**ExecuteReadSingle**

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT \* FROM [Order Details] WITH (NOLOCK) WHERE OrderID = @iOrderID AND ProductID = @iProductID";  
 db.Input = new { OrderID = 10260, ProductID = 41 };

    OrderDetail order = db.ExecuteReadSingle<OrderDetail>();

The above call will produce a single class instance (the first row) regardless of how many rows actually came back.

|  |  |
| --- | --- |
| OrderDetail | |
| OrderID | 10260 |
| ProductID | 41 |
| UnitPrice | 7.7000 |
| Quantity | 16 |
| Discount | 0.25 |

**ExecuteSetRead**

ExecuteSetRead works just like ExecuteRead except that it takes multiple generic parameters corresponding to the types the call will return. Each Table property in the resulting DALTuple object is a list. In this example we will also take advantage of a table parameter and also notice that we are calling a stored procedure, so we should not set the IsStoredProcedure property to false and leave it set to its default of true.

    var db = new SqlDBAccess(connString);   
 db.Input = new

{

CustomerIDs = new List<CustomerFilter>  
 {  
 new CustomerFilter { CustomerID = "LAZYK" },  
 new CustomerFilter { CustomerID = "LAUGB" }  
 }

};  
 db.QueryString = "stp\_GetCustomerInfoAndOrders";

    DALTuple<Customer, Order> custOrders = db.ExecuteSetRead<Customer, Order>();

The above query results in the following output:

|  |  |
| --- | --- |
| DALTuple<Customer,Order> | |
| **Table1** | |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | CustomerID | CompanyName | ContactName | ContactTitle | Address | City | Region | PostalCode | Country | Phone | Fax | | LAUGB | Laughing Bacchus Wine Cellars | Yoshi Tannamuri | Marketing Assistant | 1900 Oak St. | Vancouver | BC | V3F 2K1 | Canada | (604) 555-3392 | (604) 555-7293 | | LAZYK | Lazy K Kountry Store | John Steel | Marketing Manager | 12 Orchestra Terrace | Walla Walla | WA | 99362 | USA | (509) 555-7969 | (509) 555-6221 | |
| **Table2** | |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | OrderID | CustomerID | EmployeeID | OrderDate | RequiredDate | ShippedDate | ShipVia | Freigh | ShipName | ShipAddress | ShipCity | ShipRegion | ShipPostalCode | ShipCountry | | 10495 | LAUGB | 3 | 4/3/1997 12:00:00 AM | 5/1/1997 12:00:00 AM | 4/11/1997 12:00:00 AM | 3 | 4.6500 | Laughing Bacchus Wine Cellars | 2319 Elm St. | Vancouver | BC | V3F 2K1 | Canada | | 10620 | LAUGB | 2 | 8/5/1997 12:00:00 AM | 9/2/1997 12:00:00 AM | 8/14/1997 12:00:00 AM | 3 | 0.9400 | Laughing Bacchus Wine Cellars | 2319 Elm St. | Vancouver | BC | V3F 2K1 | Canada | | 10810 | LAUGB | 2 | 1/1/1998 12:00:00 AM | 1/29/1998 12:00:00 AM | 1/7/1998 12:00:00 AM | 3 | 4.3300 | Laughing Bacchus Wine Cellars | 2319 Elm St. | Vancouver | BC | V3F 2K1 | Canada | | 10482 | LAZYK | 1 | 3/21/1997 12:00:00 AM | 4/18/1997 12:00:00 AM | 4/10/1997 12:00:00 AM | 3 | 7.4800 | Lazy K Kountry Store | 12 Orchestra Terrace | Walla Walla | WA | 99362 | USA | | 10545 | LAZYK | 8 | 5/22/1997 12:00:00 AM | 6/19/1997 12:00:00 AM | 6/26/1997 12:00:00 AM | 2 | 11.9200 | Lazy K Kountry Store | 12 Orchestra Terrace | Walla Walla | WA | 99362 | USA | |

**ExecuteSetRead with more than 16 return sets**

The generic ExecuteSetRead overloads accept up to 16 generic parameters. If you have more than 16 return sets, there is an additional overload which accepts an array of Type objects. The return will be of type List<List<Object>>, and you must cast each inner list to the type you are expecting from it if you need to use them. See the example below:

    var types = new Type[]

    {

        typeof(ReadType1Return),

        typeof(ReadType2Return),

        typeof(ReadType3Return),

        typeof(ReadType4Return),

        typeof(ReadType5Return),

        typeof(ReadType6Return),

        typeof(ReadType7Return),

        typeof(ReadType8Return),

        typeof(ReadType9Return),

        typeof(ReadType10Return),

        typeof(ReadType11Return),

        typeof(ReadType12Return),

        typeof(ReadType13Return),

        typeof(ReadType14Return),

        typeof(ReadType15Return),

        typeof(ReadType16Return),

        typeof(ReadType17Return)

    };

   var db = new SqlDBAccess(connString);  
 db.QueryString = "stp\_SomeProcWith17ReturnSets";

    var tables = db.ExecuteSetRead(types);  
  
 List<ReadType1Return> table1 = tables[0].OfType<ReadType1Return>().ToList();  
 List<ReadType2Return> table2 = tables[1].OfType<ReadType2Return>().ToList();  
 List<ReadType3Return> table3 = tables[2].OfType<ReadType3Return>().ToList();  
 ...  
 List<ReadType17Return> table17 = tables[16].OfType<ReadType17Return>().ToList();

**ExecuteRelatedSetRead**

This version of ExecuteSetRead relates the data tables to each other after the call returns. It requires a list of relationships to be passed in. The relationships are of type DALRelationship:

* String ParentColumn
  + The column of the parent table to be used in the join.
  + In the case of joining to a property in a nested class, use a period to define a path to the property. ex if the CustomerID was located in a property called CustomerInfo, then the parent column in the relationship would become “CustomerInfo.CustomerID”. Child columns can also be nested in the same way.
* String ChildColumn
  + The column of the child table to be used in the join.
* String ParentProperty
  + The property on the parent class in which to populate the child elements. This must be a List of type matching the child elements. If you are letting the DAL create classes for you, it will create a List<Object> to use.
* int ParentTableIndex
  + This can be used to define a custom relationship so that multiple child tables join to one parent table. If you are setting this then the ChildTableIndex must also be set.
* int ChildTableIndex
  + This must be used in tandem with ParentTableIndex. This defines which child table maps to the parent table.

The example shows a result set of one customer, his orders, and each order’s details. First the classes:

   public class Customer

   {

        public String CustomerID { get; set; }  
        public String CompanyName { get; set; }  
        public String ContactName { get; set; }  
        public String ContactTitle { get; set; }  
        public String Address { get; set; }  
        public String City { get; set; }  
        public String Region { get; set; }  
        public String PostalCode { get; set; }  
        public String Country { get; set; }  
        public String Phone { get; set; }  
        public String Fax { get; set; }  
        public List<Order> Orders { get; set; }

}  
  
   public class Order

   {

        public int OrderID { get; set; }  
        public String CustomerID { get; set; }  
        public int? EmployeeID { get; set; }  
        public DateTime? OrderDate { get; set; }  
        public DateTime? RequiredDate { get; set; }  
        public DateTime? ShippedDate { get; set; }  
        public int ShipVia { get; set; }  
        public Decimal Freight { get; set; }  
        public String ShipName { get; set; }  
        public String ShipAddress { get; set; }  
        public String ShipCity { get; set; }  
        public String ShipRegion { get; set; }  
        public String ShipPostalCode { get; set; }  
        public String ShipCountry { get; set; }  
        public List<OrderDetail> OrderDetails { get; set; }

}

   public class OrderDetail

   {

        public int OrderID { get; set; }  
        public int ProductID { get; set; }  
        public Decimal UnitPrice { get; set; }  
        public short Quantity { get; set; }  
        public float Discount { get; set; }  
 }

And the code to populate them:

    var relationships = new List<DALRelationship>

    {

        new DALRelationship  
 {

         ChildColumn = "CustomerID",

         ParentColumn = "CustomerID",

         ParentProperty = "Orders",

},  
 new DALRelationship  
 {

         ChildColumn = "OrderID",

         ParentColumn = "OrderID",

         ParentProperty = "OrderDetails",

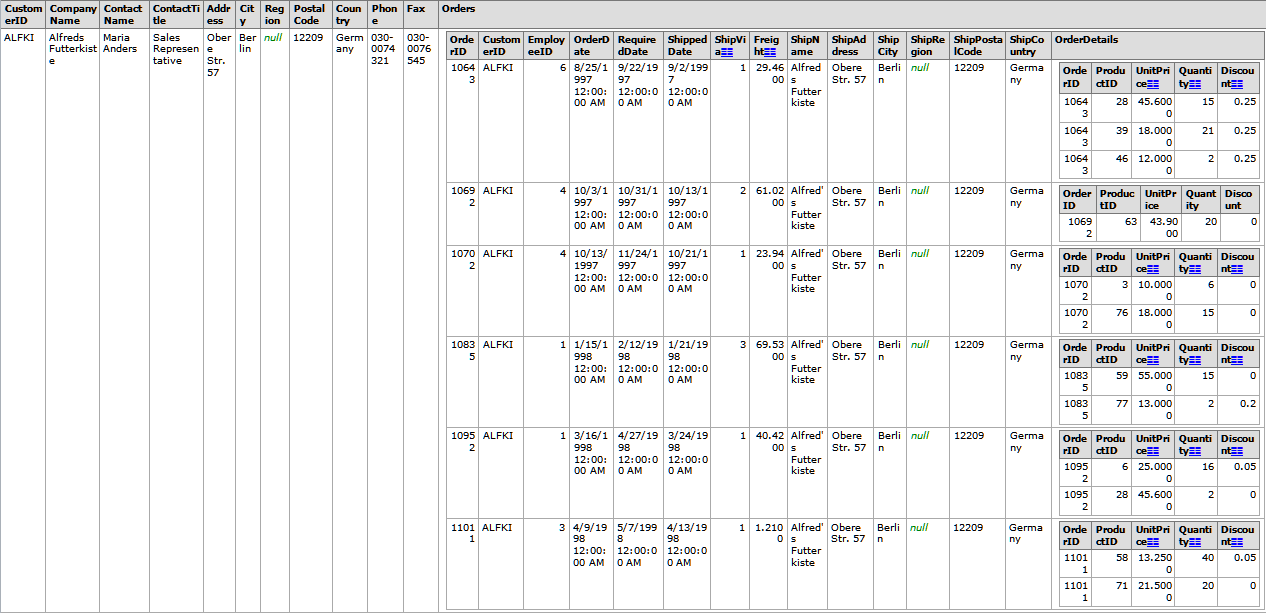
}

};

   var db = new SqlDBAccess(connString);  
 db.IsStoredProcedure = false;  
 db.QueryString = @"SELECT \* FROM Customers  
 SELECT \* FROM Orders  
 SELECT \* FROM [Order Details]";

    var tables = db.ExecuteRelatedSetRead<Customer, Order, OrderDetail>(relationships);

The contents of the tables variable (for the first customer):



**ExecuteScalar**

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT COUNT(\*) FROM Orders WITH (NOLOCK)";

    int numOrders = db.ExecuteScalar<int>();

The above ExecuteScalar query returns the result of the select statement as an integer.

**ExecuteScalarEnumeration**

ExecuteScalarEnumeration will return a list of scalar values populated with the first column of each row of a result set.

    var db = new SqlDBAccess(connString);  
    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT TOP 10 ProductName,UnitPrice FROM Products ORDER BY UnitPrice DESC";

    List<String> mostExpensiveProductNames = db.ExecuteScalarEnumeration<String>();

The above query returns the following list of strings:

|  |
| --- |
| List<String> |
| Côte de Blaye |
| Thüringer Rostbratwurst |
| Mishi Kobe Niku |
| Sir Rodney's Marmalade |
| Carnarvon Tigers |
| Raclette Courdavault |
| Manjimup Dried Apples |
| Tarte au sucre |
| Ipoh Coffee |
| Rössle Sauerkraut |

**ExecuteNonQuery**

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.ExpectedAffectedRows = 1;  
 db.Input = new

{  
 CustomerID = "LAZYK",  
 CompanyName = "UPDATE"  
 };  
 db.QueryString = "UPDATE Customers SET CompanyName = @iCompanyName WHERE CustomerID = @iCustomerID";

    int rowsAffected = db.ExecuteNonQuery();

There is an additional feature of the ExecuteNonQuery: the OnAffectedRowsMismatch event. In the above example, if for some reason there were two customers matching the @iCustomerID parameter, the OnAffectedRowsMismatch event would be thrown, since we were only expecting one row to be updated. Like all other events, this event can be handled for a specific instance of a SqlDBAccess or ODBCAccess class, or globally in the DALEvents static class. Setting the ExpectedRows property is not required for a non query call. If this is left as null, the check will not be made and the event will not fire.

**Cross Provider Set Read**

This DAL comes with four providers by default with the ability to add your own (covered later in this document). The four default providers are SQL, ODBC, CSV and an in memory objects provider. The CSV and in memory objects providers allow read only access. Some quick examples are provided later in this document.

You can join multiple queries from multiple providers into a single data set and even apply relationships to them. The resulting data set can be a maximum of 16 strongly typed tables, or an unlimited number of weakly typed tables.

Example:

var tuple = db1.BeginCrossProviderRead<Object>()

.JoinWith(db2, new CPSegment<Object, Object>())

.JoinWith(db3, new CPSegment<Object>())

.AsDataSet();

db1, db2, and db3 can be any IDBAccess providers. In this instance the first provider returns one table, db2 returns two tables, and db3 returns one table. We are creating runtime types for all of these tables. We could have instead passed in classes if we had them available. Finally, we want to get everything as a dataset. The resulting tuple will have a Table1, Table2, Table3, and Table4. We could have also built a set of relationships and instead of calling AsDataSet we could have called AsRelatedDataSet passing in the relationships to get a related data set back just like the normal ExecuteRelatedSetRead.

**Other functionality**

**Preparing your own parameters**

As previously shown, a SqlParameter enumeration can be used as input to a database call. If you want to manipulate the parameters yourself, the method which converts an object into parameters is exposed. Simply call the PrepareParameters method off of an IDBAccess object, passing your input object and optionally any prefix settings.

* IEnumerable<SqlParameter> PrepareParameters(Object input, Boolean? prefixDirection = null, String inputPrefix = null, String outputPrefix = null)
  + Prepares a collection of SqlParameter objects.
  + Input
    - The object to be prepared. This can be any class, anonymous type, or an enumeration of SqlParameter object.
  + InputPrefix
    - The string to prefix to the input parameter names. Defaults to the value set on the class.
  + OutputPrefix
    - The string to prefix to the output parameter names. Defaults to the value set on the class.

**Serialization**

There are two extension methods that can be used on any object, SerializeToJSON and SerializeToXML.

Use the XMLTableNames property of the SqlDBAccess or ODBCAccess class when serializing runtime types to XML if you want to control what class names are output as or JSON if you are using the include TableName setting. When using the SerializeToJSON method, you can pass whether or not to include the TableName property in the JSON (this setting is ignored for XML). This property will be present on all types created at runtime, and will be equal to the names set on the XMLTableNames property. The SerializeToXML method optionally allows you to pass the name of the root element. These functions cache the serialized string so you can quickly retrieve it more than one time from the same DALTuple class.

This DAL also supports serialization of types that .NET does not. Note, these extra serializable types may not usable over web services and may not be deserializable, but can be written out to a file. Extra supported types are LINQ IGrouping objects.

**Serialization Examples**:

The following code sets up an execute related set read with some table names that we are specifying:

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;

    var rels = new List<DALRelationship>

{

new DALRelationship

{

ChildColumn = "CustomerID",

ParentColumn = "CustomerID",

ParentProperty = "Orders"

},

new DALRelationship

{

ChildColumn = "OrderID",

ParentColumn = "OrderID",

ParentProperty = "OrderDetails"

}

};

    db.XMLTableNames = new List<String> { "Customer", "Order", "OrderDetail" };

    db.QueryString = @"SELECT TOP 2 \* FROM Customers ORDER BY CustomerID

              SELECT \* FROM Orders ORDER BY CustomerID

               SELECT \* FROM [Order Details]";

    var objs = db.ExecuteRelatedSetRead<Object, Object, Object>(rels);

The next two statements will demonstrate the difference between providing the name of the root element and not providing it.

    String xml = objs.SerializeToXML();

    String xmlWithCustomers = objs.SerializeToXML("Customers");

The xml variable will be serialized as such. Notice that lines in red correspond to table names we set in the XMLTableNames property, and lines in blue correspond to the ParentProperty name we set in the relationships.



Notice that in the above document the entire xml document is wrapped in <DALRuntimeTypeBaseList>. In the following document which is the content of the xmlWithCustomers variable, the document is wrapped in a <Customers> tag.



The following two examples highlight the difference between including the table names and not including the table names in a JSON serialization.

    String json = objs.SerializeToJSON();

    String jsonWithTableNames = objs.SerializeToJSON(true);

****

If serializing a related set read and are not using a generic version, then when serializing the resulting dataset you should be sure to only serialize the first list in the resulting set.

var objs = db.ExecuteRelatedSetRead(rels);  
 String json = objs[0].SerializeToJSON();

**Transactions**

Transactions work the same as the regular SqlTransaction object does. You can either pass your own transaction object into a SqlDBAccess or ODBCAccess call, or you can use the transaction methods exposed on the SqlDBAccess or ODBCAccess class to have the DAL manage the transaction for you. There are a few additional features available when using the DAL transaction methods.

**Auto Rollback:**

A transaction can be set to automatically rollback when an exception occurs during a query. A transaction can be rolled back completely, or just to the last save point (SqlDBAccess only). These settings can be passed in during the BeginTransaction call, or set on the SqlDBAccess or ODBCAccess object.

Ex. 1. Complete auto rollback.

    var db = new SqlDBAccess(connString);  
 db.AutoRollbackCompletely = true;

    db.IsStoredProcedure = false;  
 db.BeginTransaction();  
 db.QueryString = "SELECT CategoryID,CategoryName,Description FROM Categories WITH (NOLOCK)";

    db.ExecuteRead<Category>();

The above code produces the following output:

|  |  |  |
| --- | --- | --- |
| CategoryID | CategoryName | Description |
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses |
| 5 | Grains Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |

Now we will execute an update statement and an insert statement:

db.QueryString = "UPDATE Categories SET Description = 'Cheeses and eggs' WHERE CategoryID = 4";  
    db.ExecuteNonQuery();  
 db.SaveTransaction("DairyUpdate");

db.QueryString = "INSERT INTO Categories (CategoryName,Description) VALUES ('Books','Specialty recipe books')";  
    db.ExecuteNonQuery();

After the update and the insert statement the data set now looks like this to the transaction:

|  |  |  |
| --- | --- | --- |
| CategoryID | CategoryName | Description |
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses and eggs |
| 5 | Grains Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |
| 9 | Books | Specialty recipe books |

Now we will execute a delete statement which causes an error, causing the transaction to be rolled back.

try   
 {     
 db.QueryString = "DELETE Categories WHERE CategoryID = 1";  
 db.ExecuteNonQuery();  
 }  
 catch  { }

After the above statement is attempted, the dataset now looks like it did at the beginning:

|  |  |  |
| --- | --- | --- |
| CategoryID | CategoryName | Description |
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses |
| 5 | Grains Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |

If we had not set the rollback completely option, it would have rolled back to the last save point when CategoryID 4 was updated, and the final data set would have instead looked like this:

|  |  |  |
| --- | --- | --- |
| CategoryID | CategoryName | Description |
| 1 | Beverages | Soft drinks, coffees, teas, beers, and ales |
| 2 | Condiments | Sweet and savory sauces, relishes, spreads, and seasonings |
| 3 | Confections | Desserts, candies, and sweet breads |
| 4 | Dairy Products | Cheeses and eggs |
| 5 | Grains Cereals | Breads, crackers, pasta, and cereal |
| 6 | Meat Poultry | Prepared meats |
| 7 | Produce | Dried fruit and bean curd |
| 8 | Seafood | Seaweed and fish |

**Table parameters as anonymous types**

Table parameters can also be constructed out of anonymous types, but only if the entire input property is an anonymous type (the reason being that if you had an input model, you could not declare one of the properties to be an anonymous type or an enumeration of an anonymous type). Here is an example of using the return value of a LINQ select:

    var custs = GetActiveCustomersByYear(DateTime.Now.Year - 1);  
 var mvpCusts = custs  
 .Where(c => c.AmountSpentLastYear >= 5000)  
 .Select(c => new  
 {  
 CustomerID = c.CustomerID,  
 MVPDiscount = GetMVPDiscountByCustomerID(c.CustomerID)  
 });  
  
 var db = new SqlDBAccess(connString);  
 db.Input = new  
 {  
 Year = DateTime.Now.Year,  
 MVPCustomers = mvpCusts  
 };  
 db.QueryString = "stp\_insertNewMVPCustomers";

    db.ExecuteNonQuery();

**Using runtime types via the FastDynamicAccess class:**

The FastDynamicAccess class can be used in your code to get properties of any class, including classes generated at runtime. In the following example we use the FastDynamicAccess class and LINQ to group the orders returned to us in a list of a type generated at runtime and count how many in each group do not have a shipping region.

    var db = new SqlDBAccess(connString);

    db.IsStoredProcedure = false;  
 db.QueryString = "SELECT \* FROM Orders WITH (NOLOCK)";

    var orders = db.ExecuteRead<Object>();  
  
 //Get a FastDynamicAccess object for an order object.

FastDynamicAccess ordersFDA = FastDynamicAccess.Get(orders.First());  
 var groups = orders.GroupBy(o => ordersFDA.Get<String>(o, "ShipCountry"));  
 foreach (var g in groups)  
 {

g.Key; // the group key. In this case the ShipCountry.  
 g.Count(groupOrders => ordersFDA.Get<String>(groupOrders, "ShipRegion") == null); // the count of null ShipRegion in this group.  
 }

Alternatively, you can access properties of objects using the FastDynamicAccess class by the index the fda object has assigned it. The indexes can be looked up in the PropertyToArrayIndex property on the fda object. Using the index of your property instead of the property name is significantly faster.

Another alternative is to use the GetValue, GetValue<T>, or SetValue extension methods available on any object. These then call an FDA object. Since the FDA object must be looked up each time this method is slower but it is more convenient than using an FDA object directly.

**Attribute Examples:**

There will generally be two different models for each SQL operation if you are returning a data set. There will be one input class and one output class. In the below example we are searching for all customers who have spent more than X amount with an optional date range filter. Note that optional parameters to SQL queries or parameters which represent nullable database columns should be passed as a nullable value type if it is not already a reference type.

For these examples, changes to the model since the last step will be bolded.

    public class InputModel

    {

public String SearchType { get; set; }

        public Decimal AmountSpent { get; set; }

        public DateTime? BeginDate { get; set; }

        public DateTime? EndDate { get; set; }

public int TotalCustomers { get; set; }

    }

    public class OutputModel

    {

        public int CustomerId { get; set; }

        public String FirstName { get; set; }

        public String LastName { get; set; }

        public int NumberOfOrders { get; set; }

        public Decimal AmountSpent { get; set; }

        public String CustomerSince { get; set; }

        public String LastSeen { get; set; }

    }

In the following example of our InputModel, the begin date and end dates are called something differently from the view than what they are called in the SQL statement or procedure. The view calls them BeginDate and EndDate, while the SQL expects to get StartRangeFilter and EndRangeFilter.

    public class InputModel

    {

public String SearchType { get; set; }

        public Decimal AmountSpent { get; set; }

**[DALSQLParameterName(Name = "StartRangeFilter")]**

**public DateTime? BeginDate { get; set; }**

**[DALSQLParameterName (Name = "EndRangeFilter")]**

**public DateTime? EndDate { get; set; }**

public int TotalCustomers { get; set; }

    }

In the following example we add the DALIgnore attribute on the SearchType. The search type property may for example determine exactly what SQL to execute, but it is not itself an input to the SQL.

    public class InputModel

    {

**[DALIgnore]**

**public String SearchType { get; set; }**

        public Decimal AmountSpent { get; set; }

        [DALSQLParameterName(Name = "StartRangeFilter")]

        public DateTime? BeginDate { get; set; }

        [DALSQLParameterName(Name = "EndRangeFilter")]

        public DateTime? EndDate { get; set; }

public int TotalCustomers { get; set; }

    }

The next example shows how we can define the direction of a sql parameter in our model. In our case, we have a TotalCustomers int in the input model. Setting this to be an output parameter will allow us to retrieve the value of this after the SQL call.

    public class InputModel

    {

        [DALIgnore]

        public String SearchType { get; set; }

        public Decimal AmountSpent { get; set; }

        [DALSQLParameterName(Name = "StartRangeFilter")]

        public DateTime? BeginDate { get; set; }

        [DALSQLParameterName(Name = "EndRangeFilter")]

        public DateTime? EndDate { get; set; }

**[DALParameterDirection(Direction = ParameterDirection.Output)]**

**public int TotalCustomers { get; set; }**

    }

The next example shows the DALWriteStringFormat attribute. In this example, the output from the SQL returns two dates, but we want to apply a String.Format to them. Simply add the DALWriteStringFormat attribute to the property, and the returned value will have that string format applied before being set in the model. In this case we want to apply a custom format to the returned DateTime values. Note, this attribute can only be used with String properties.

    public class OutputModel

    {

        public int CustomerId { get; set; }

        public String FirstName { get; set; }

        public String LastName { get; set; }

        public int NumberOfOrders { get; set; }

        public Decimal AmountSpent { get; set; }

**[DALWriteStringFormat(Format = "{0:MM/dd/yyyy}")]**

**public String CustomerSince { get; set; }**

**[DALWriteStringFormat(Format = "{0:MM/dd/yyyy}")]**

**public String LastSeen { get; set; }**

    }

The final example shows the DALDefaultValue attribute. The default value attribute allows you to assign a default value to a property in the case where the SQL statement does not return that property. The default value type must match the property type and cannot be null, since any nullable or reference type will automatically default to null. In this example, the SQL does not actually return the names of the customers, but we must send something not null back to our view.

    public class OutputModel

    {

        public int CustomerId { get; set; }

**[DALDefaultValue(Value = "Customer First Name")]**

**public String FirstName { get; set; }**

**[DALDefaultValue(Value = "Customer Last Name")]**

**public String LastName { get; set; }**

        public int NumberOfOrders { get; set; }

        public Decimal AmountSpent { get; set; }

        [DALWriteStringFormat(Format = "{0:MM/dd/yyyy}")]

        public String CustomerSince { get; set; }

        [DALWriteStringFormat(Format = "{0:MM/dd/yyyy}")]

        public String LastSeen { get; set; }

    }

After all of our changes the two Models now look like this:

    public class InputModel

    {

        [DALIgnore]

        public String SearchType { get; set; }

        public Decimal AmountSpent { get; set; }

        [DALSQLParameterName(Name = "StartRangeFilter")]

        public DateTime? BeginDate { get; set; }

        [DALSQLParameterName(Name = "EndRangeFilter")]

        public DateTime? EndDate { get; set; }

        [DALParameterDirection(Direction = ParameterDirection.Output)]

        public int TotalCustomers { get; set; }

    }

    public class OutputModel

    {

        public int CustomerId { get; set; }

        [DALDefaultValue(Value = "Customer First Name")]

        public String FirstName { get; set; }

        [DALDefaultValue(Value = "Customer Last Name")]

        public String LastName { get; set; }

        public int NumberOfOrders { get; set; }

        public Decimal AmountSpent { get; set; }

        [DALWriteStringFormat(Format = "{0:MM/dd/yyyy}")]

        public String CustomerSince { get; set; }

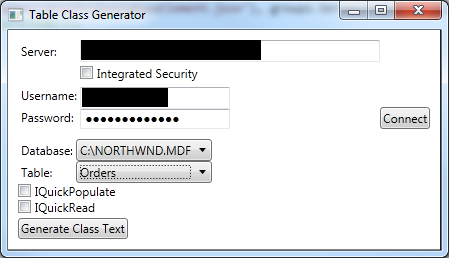
        [DALWriteStringFormat(Format = "{0:MM/dd/yyyy}")]

        public String LastSeen { get; set; }

    }

**TableClassGenerator:**

There is a small and simple utility which will generate classes for you based on an existing database table.

****

Enter the server name and a username and password (or select Integrated Security) and press connect. The databases will be populated. Select a database, and then select a table to create a class for. Press Generate Class Text and the class code will be copied to the clipboard as text:

public class Orders

{

public int OrderID { get; set; }

public String CustomerID { get; set; }

public int? EmployeeID { get; set; }

public DateTime? OrderDate { get; set; }

public DateTime? RequiredDate { get; set; }

public DateTime? ShippedDate { get; set; }

public int? ShipVia { get; set; }

public decimal? Freight { get; set; }

public String ShipName { get; set; }

public String ShipAddress { get; set; }

public String ShipCity { get; set; }

public String ShipRegion { get; set; }

public String ShipPostalCode { get; set; }

public String ShipCountry { get; set; }

}

**CSV Provider**

The CSV provider can be used to read in a CSV file into the DAL to be joined with other data sources or to be mapped to classes. Only basic reading functionality is provided. An ExecuteSetRead is possible when providing multiple CSV files. There is no support for updates or querying the CSV files.

**In Memory Provider**

The in memory provider allows you to take a collection of objects from any source and given to the DAL to be joined to other data sources. You must pass the type of each object in the collection to the class at instantiation as well as the collection. Optionally you can provide a table name for this collection if serialization will be done.

var db = new InMemoryAccess<MyClass>(myList);

Once an InMemoryAccess is instantiated it can be used in a cross provider set read as previously shown.

**Extensibility**

The DAL can be extended to work with other DB providers by implementing the IDBAccess interface. The IDBAccess interface provides provider specific functionality (reading and updating of the data source). The rest of the functionality of the DAL is implemented via extension methods, so the mapping/populating into classes/runtime types comes with the implementation of the IDBAccess interface. Any provider specific functionality (for example, an execute read which returns just a DataTable) must be written by your provider, as the interface only handles the functionality of mapping returned data sets/scalars into object types.

There are three properties which must be implemented:

* Boolean PopulateDefaultValues
  + Sets whether or not model properties not returned by a query should be defaulted to the value defined in the DALDefaultValue attribute.
* TraceLevel TraceOutputLevel
  + The level at which to output trace event messages.
* Dictionary<Type, ModelData> ModelsData\*
  + Used internally by the DAL to cache information about model types for population/parameter preparation.
  + The constructor of your IDBAccess object must set this to be a new Dictionary<Type, ModelData>()

Along with four methods:

* ExecuteReadQuickTuple ExecuteReadQuick()\*
  + Function which performs an execute read against a data source using the properties set on the IDBAccess class.
* int ExecuteNonQuery()
  + Function which performs an execute non query against a data source using the properties set on the IDBAccess class.
* Object ExecuteScalar()
  + Function which performs an execute scalar against a data source using the properties set on the IDBAccess class.
* List<ExecuteReadQuickTuple> ExecuteSetReadQuick()\*
  + Function which performs an execute set read against a data source using the properties set on the IDBAccess class.

\* Denotes that this part of the interface does not need to be publically accessible (and should not be) and should probably be implemented explicitly in order to hide it from intellisense for an object implementing the interface.

ExecuteReadQuickTuple

The ExecuteReadQuickTuple class contains a representation of the returned data set. It has the following properties:

* List<Object[]> DataRows
  + A list of object arrays representing each row and its columns from the returned data table.
* List<String> ColumnNames
  + A list of the column names in the same order as their values appear in each object array in the DataRows list.
* List<Type> ColumnTypes
  + A list of the column types in the same order as their values appear in each object array in the DataRows list. If any values in the DataRows list are null, the type must be nullable in some way: either a reference type or a nullable value type.
* String TableName
  + A name given to the table which populated this tuple object. This is used for proper XML and JSON serialization, and can be left null if that functionality is not needed.

**ICustomPopulate**

The ICustomPopulate interface can be used to define a custom population routine (mapping of return data set to objects). It has the following methods:

* List<T> PopulateModelBaseEnumeration<T>(ExecuteReadQuickTuple tuple, List<String> parentChildPropertyName = null) where T : class, new()
  + This method populates a list of type T based on the given tuple.
  + parentChildPropertyName is only used if you are implementing runtime types.
* List<Object> PopulateModelBaseEnumeration(ExecuteReadQuickTuple tuple, Type modelType, List<String> parentChildPropertyName = null)
  + This method populates a list of objects based on the given tuple. This is used if the caller does not want a strongly typed list. For instance, during an ExecuteSetRead of size > 16.
  + modelType is the type of object to create for each row in the tuple.
  + parentChildPropertyName is only used if you are implementing runtime types.

In both of these methods a call to ValidateForDAL may be required for your custom provider object if the flow through the DAL will hit code that uses this dictionary. If you are using the default PrepareParameters routine you will need this, otherwise you will most likely not need to check for this in an ICustomPopulate situation.

if (!db.ModelsData.ContainsKey(modelType))

{

db.ValidateForDAL(new T());

}

**Runtime types with ICustomPopulate**

To implement runtime types in your provider, you can simply return a call to PopulateModelBaseEnumeration passing it the tuple and a further call to tuple.DataRows.GetRuntimeType() as the type to populate. Your population code must be generic enough to handle populating into the runtime type that is generated by the call to GetRuntimeType.

**DotNETCompatibleProvider**

The DotNETCompatibleProvider is a base class from which one can derive a custom provider. Providers which derive this class must use all .NET database interface types (IDbConnection, IDbCommand, IDbDataParameter, IDbTransaction, IDbDataAdapter).

The most basic derivation of this class is to simply derive from it with the appropriate generic arguments. The following example will allow MySQL access with the full functionality that the DAL provides:

public class MySQLAccess : DotNETCompatibleProvider<MySqlConnection, MySqlCommand, MySqlParameter, MySqlTransaction, MySqlDataAdapter, MySqlException>

{

}

Here are four recommended constructors to include with your provider. Notice all of the constructors call a base constructor. These are the four constructors on the DotNETCompatibleProvider class that are available to child classes.

public class MySQLAccess : DotNETCompatibleProvider<MySqlConnection, MySqlCommand, MySqlParameter, MySqlTransaction, MySqlDataAdapter, MySqlException>

{  
 public MySQLAccess()  
 : base(null)  
 { }  
  
 public MySQLAccess(String connString)  
 : base(connString, null)  
 { }  
  
 public MySQLAccess(String connString, String sqlStatement)  
 : base(connString, sqlStatement, null)  
 { }  
  
 public MySQLAccess(String connString, String sqlStatement, Object input)  
 : base(connString, sqlStatement, input)  
 { }

}

The behavior of the DotNETCompatibleProvider can be customized by overriding the base methods. The following methods can be overridden.

**Helper Functions**:

* public virtual void BeginTransaction(Boolean? autoRollback = null)
  + In the base class this method will flag that the next query operation should start a transaction. If autoRollback isn’t supplied it will default to the value set on the class at the time.
* public virtual void BeginTransaction(IsolationLevel iso, Boolean? autoRollback = null)
  + In the base class this method will flag that the next query operation should start a transaction. If autoRollback isn’t supplied it will default to the value set on the class at the time.
  + This overload should also set the isolation level on the class to the isolation level that is passed in.
* public virtual void CommitTransaction()
  + In the base class this commits a transaction and disposes it.
* public virtual void RollbackTransaction()
  + In the base class this rolls a transaction back and disposes it.
* public virtual void PerformAutoRollback()
  + In the base class this is called whenever an exception occurs during a query.
* public virtual IEnumerable<TParameter> PrepareParameters(Object input, Boolean? prefixDirection = null, String inputPrefix= null, String outputPrefix= null)
  + Return type example: IEnumerable<MySqlParameter>
  + In the base class this enumerates over the input properties of the input object and calls the PrepareSingleParameter method for each one.
  + If the input object is already of type IEnumerable<TParameter> then this method should simply return the input.
  + If the type of the input does not exist in the ModelsData property of the DBAccess class, you should call this.ValidateForDAL on the input object before the calls to PrepareSingleParameter are made. Do NOT override or implement your own version of ValidateForDAL.
* protected virtual IEnumerable<TParameter> PrepareSingleParameter(ModelData data, Object input, Boolean? prefixDirection, String propertyName, Object value, String inputPrefix= null, String outputPrefix= null)
  + In the base class this creates an object of the type TParameter based on the property name and object value.
  + The parameter data should be the value from the ModelsData dictionary of the DBAccess class.
  + This should return an enumeration in the case that the value parameter is actually a nested model. In this case the object should be passed to the PrepareParameters function, and the result of that returned from this function.
* protected virtual T ExecuteSql<T>(Func<TCommand, T> sqlAction)
  + In the base class this is a wrapper for all query operations.
  + The sqlAction parameter is a function which takes in a parameter of TCommand (ex: MySqlCommand object).
  + The return value T is what the query function will return. Example: DataTable or DataSet.
  + This method should take care of preparing the TCommand object, timing the query (if that is necessary), catching any exceptions which occur, raising any appropriate events, setting the OutputValues dictionary if there are any, and calling the RaiseOnQueryComplete method.
  + The command object should be properly disposed of in this method.
* protected virtual TCommand PrepareCommand()
  + In the base class this method prepares a command object of type TCommand (ex: MySqlCommand).
  + This method should take care of setting the connection on the object (creating and opening a new one if necessary), starting a new transaction if necessary, assigning the query text, query type, and any parameters to the command object.

**Query Functions**:

* public virtual int ExecuteNonQuery()
  + In the base class this method calls ExecuteSql passing it a function which calls ExecuteNonQuery on the command object.
  + This method should also call RaiseOnAffectedRowsMismatch when necessary.
* public virtual Object ExecuteScalar()
  + In the base class this method calls ExecuteSql passing it a function which calls ExecuteScalar on the command object.
* ExecuteReadQuickTuple IDBAccess.ExecuteReadQuick()
  + In the base class this method calls ExecuteSql passing it a function which calls ExecuteRead on the command object.
  + This method must populate an ExecuteReadQuickTuple object. This is the method which will be called by the ExecuteRead functions which map to objects. You can manually iterate through the DataReader object to quickly get the results of the query.
  + This method is also used by the ExecuteReadSingle method which maps to an object. If using a DataReader object and you know if you have come from this method, you can short circuit the reading of the entire dataset since only one row will be returned.
  + Because this method is an explicitly implemented interface method it cannot be overridden. If this should be overridden, simply declare this method in the child class and mark it with the new keyword.
* public virtual DataTable ExecuteRead()
  + In the base class this method calls ExecuteSql passing it a function which calls ExecuteRead on the command object and fills a DataTable.
* public virtual DataRow ExecuteReadSingle()
  + In the base class this method calls ExecuteRead and returns the first DataRow or null if none exist.
* List<ExecuteReadQuickTuple> IDBAccess.ExecuteSetReadQuick()
  + In the base class this method calls ExecuteSql passing it a function which calls ExecuteSetRead on the command object.
  + This method must populate a list of ExecuteReadQuickTuple objects. This is the method which will be called by the ExecuteSetRead functions which map to objects. You can manually iterate through the DataReader object to quickly get the results of the query.
  + This method is also used by the ExecuteRelatedSetRead methods which map to objects.
  + Because this method is an explicitly implemented interface method it cannot be overridden. If this should be overridden, simply declare this method in the child class and mark it with the new keyword.
* public virtual DataSet ExecuteSetRead()
  + In the base class this method calls ExecuteSql passing it a function which calls ExecuteSetRead on the command object and fills a DataSet.

**Events**:

* protected virtual void HandleOnQueryException(TException ex, TConnection conn, IEnumerable<TParameter> parameters, int? timeout, String queryString)
  + In the base class this calls RaiseOnQueryException.
* protected virtual void HandleOnException(Exception ex, TConnection conn, IEnumerable<TParameter> parameters, int? timeout, String queryString)
  + In the base class this calls RaiseOnException.
* protected virtual void HandleOnException(int affectedRows, TConnection conn, IEnumerable<TParameter> parameters, int? timeout, String queryString)
  + In the base class this calls RaiseOnAffectedRowsMismatch.
* protected virtual void RaiseOnQueryException(DALQueryExceptionEventArgs e)
  + In the base class this calls the static version of RaiseOnQueryException to ensure static listeners are raised and raises the OnQueryException event if there is a subscriber.
  + It should also call RaiseOnException.
* protected virtual void RaiseOnException(DALExceptionEventArgs e)
  + In the base class this calls the static version of RaiseOnException and raises the OnException event if there is a subscriber.
* protected virtual void RaiseOnAffectedRowsMismatch(DALAffectedRowsMismatchEventArgs e)
  + In the base class this calls the static version of RaiseOnAffectedRowsMismatch and raises the OnAffectedRowsMismatch event if there is a subscriber and only if the ExpectedAffectedRows property is not null.
* protected virtual void RaiseOnQueryComplete(DALQueryCompleteEventArgs e)
  + In the base class this calls the static version of RaiseOnQueryComplete and raises the OnQueryComplete event if there is a subscriber.

Static methods cannot be overridden. If the static event methods should be overridden, simply declare them in the child class and mark them with the new keyword.

**Performance**

Numbers are in thousands of properties per second. For comparison, reading from a DataTable object by column name and column index is 16.4 and 38.5 million integer properties per second respectively.

**ExecuteRead**:

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **5 Properties** | **20 Properties** | **100 Properties** |
| ExecuteRead Integers Test | 33,026 | 35,681 | 30,431 |
| ExecuteRead Nullable Integers Test | 13,926 | 14,824 | 12,231 |
| ExecuteRead Nullable Integers All Null Test | 18,075 | 19,421 | 16,429 |
| ExecuteRead WriteStringFormat Test | 3,982 |  |  |
| ExecuteRead Integers Default Value Test | 94,492 | 151,992 | 228,683 |
| ExecuteRead Strings Test | 27,690 | 31,500 | 23,231 |
| ExecuteRead Nested Integers Test | 10,423 |  |  |
| ExecuteRead Integers Test IQuickPopulate | 82,774 | 101,714 | 111,071 |
| ExecuteRead Integers Test IQuickPopulate Using Dictionary | 17,289 | 16,785 | 17,155 |
| ExecuteRead Nullable Integers Test IQuickPopulate | 20,972 | 24,543 | 22,364 |
| ExecuteRead Strings Test IQuickPopulate | 61,425 | 81,483 | 65,989 |
| ExecuteRead Integers Runtime Type Test | 30,188 | 53,790 | 74,969 |
| ExecuteRead Nullable Integers All Null Runtime Type Test | 21,072 | 26,894 | 33,197 |

**ExecuteScalarEnumeration**:

|  |  |
| --- | --- |
| **Name** |  |
| ExecuteScalarEnumeration Integers Test | 106,533 |
| ExecuteScalarEnumeration Floats Test | 108,739 |
| ExecuteScalarEnumeration DateTimes Test | 99,339 |
| ExecuteScalarEnumeration Strings Test | 55,717 |
| ExecuteScalarEnumeration Nullable Int Test | 22,995 |
| ExecuteScalarEnumeration DBNull To String Test | 85,379 |
| ExecuteScalarEnumeration DBNull To Nullable Int Test | 19,140 |

**UDTable**:

|  |  |
| --- | --- |
| **Name** | **5 Properties** |
| UDTable Integers Test | 3,030 |
| UDTable Nullable Integers Test | 2,130 |
| UDTable ReadStringFormat Test | 1,370 |
| UDTable Strings Test | 3,125 |
| UDTable Integers Test IQuickRead | 3,703 |
| UDTable Strings Test IQuickRead | 4,166 |

Because a DataTable object is very slow to insert into, the throughput of a UDTable parameter increases greatly as the size of each row increases.

**ExecuteSetRead**:

There are no numbers provided for ExecuteSetRead and ExecuteRelatedSet read as these tests take much longer to get good numbers like the numbers above. There is no significant performance hit for both ExecuteSetRead and ExecuteRelatedSetRead.