mv.NET Adapter Objects

Developer's Introductory Guide

A product from BlueFinity

Copyright Notices

Contacting Us

We are always very happy to be able to discuss all aspects of our products with our customers – prospective and current alike. You can contact us via the following means:

Website: www.bluefinity.com
Email: support@bluefinity.com
Address: 10260 SW Greenburg Road, Suite 400, Portland, OR 97223, USA
Address: 575-599 Maxted Road, Hemel Hempstead, Herts, HP2 7DX, UK

Trademark Acknowledgements

The mv.NET product and logo are trademarks of BlueFinity International Limited.

All other trademarks and trade names are the property of their respective owners and are used in this documentation for identification purposes only

Contents

Welcome to mv.NET

Firstly, thank you for either purchasing one or more of the mv.NET products, or for taking the time to explore the great functionality that they can provide to you and your fellow developers.

This chapter outlines the members of the mv.NET family of products and also summarizes the contents of this guide.

The mv.NET Family of Products

Adapter Objects is one of the members of the mv.NET family of products authored by BlueFinity. mv.NET is *the* essential tool for any multivalued database developer wishing to create .NET based application interfaces to their current or new multivalue database file system.

The design goal of mv.NET is to enable the multivalued developer to combine the power and flexibility of proven multivalued technology with the state-of-the art, feature rich .NET environment. Its design also enables and encourages the developer to leverage, wherever possible, previously acquired multivalue skills.

BlueFinity's team of software engineers has huge knowledge and experience of using both multivalue systems and the .NET environment. We proudly regard ourselves as being one of the foremost companies in providing this technology bridge and look forward to working with you to enable you to meet your software development goals.

Feature Overview

The Adapter Objects product provides a sophisticated implementation of the ADO.NET managed data provider model, along with Visual Studio integration components to assist the developer in using the data provider within the VS.NET IDE.

The Adapter Objects architecture has been designed with both performance and flexibility in mind. This, combined with an implementation that provides seamless integration with the .NET environment, provides a powerful tool for enabling developers to harness the full power of both their MultiValued system and the .NET platform.

The product's key features are as follows:

- 100% implementation of the ADO.NET managed data provider model.

- Visual Studio integration components, such as creation wizards and XML schema generation.

- Support for optimistic locking and transaction boundaries.

The mv.NET Suite

Adapter Objects is one of three products within the mv.NET suite; the suite as a whole comprising of:

- **Core Objects** – object oriented native .NET access to MultiValued databases.

- **Binding Objects** – high performance databinding technology that enables standard .NET controls to become fully MultiValued-aware. Binding Objects links with Core Objects to provide its functionality.

- **Adapter Objects** – complete implementation of an ADO.NET managed data provider for multivalue databases, offering a standardized interface to database access.

Getting Started Guide Contents

The contents of this guide are designed to provide a basis for learning about the Adapter Objects module. Further help is provided within the Visual Studio environment using the product's dynamic and intellisense help systems. The chapters of this guide are as follows:

ADO.NET Basics

This chapter explores the basics of the ADO.NET data access architecture. It also contains links to other general information sources.

Adapter Objects Overview

This chapter provides an overview of the components that comprise the Adapter Objects package.

The mvConnection Class

The mvConnection class is responsible for establishing and holding a connection to a multivalued database. This chapter covers the Adapter Objects specific implementation of this class.

The mvCommand Class

The mvCommand class is responsible for holding the definition of a range of possible database manipulation commands. This chapter details the aspects of the class that are peculiar to the Adapter Objects implementation of the Command class.

Dynamic Normalization

This chapter outlines the Dynamic Normalization technology incorporated within Adapter Objects which addresses the problem of how to transform multi and subvalued data into ADO.NET data structures.

Multiple Commands

The mvCommand object allows you to define multiple commands to be performed in unison. This chapter explains this process and explores the implications of doing so.

The Data Adapter Definition Wizard

This chapter describes the wizard which is invoked when you drag and drop an mvDataAdapter control onto the surface of a form within Visual Studio.

ADO.NET Basics

ADO.NET is the standard method by which .NET developers are able to interact with databases. It comprises a large suite of class definitions which, collectively, provide a rich environment for database access and manipulation. This chapter explores some of the key aspects of the ADO.NET model and also provides links to sources of further information.

Installation

Adapter Objects is installed as part of mv.NET's Client Interface Developer setup routine. Please refer to the Getting Started and Core Objects guides for further information on this topic.

ADO.NET Architectural Summary

ADO.NET has a relatively complex architecture and it is beyond the scope of this manual to document all aspects of this technology. However, below is a diagram which summarizes the way in which ADO.NET's architecture is structured.
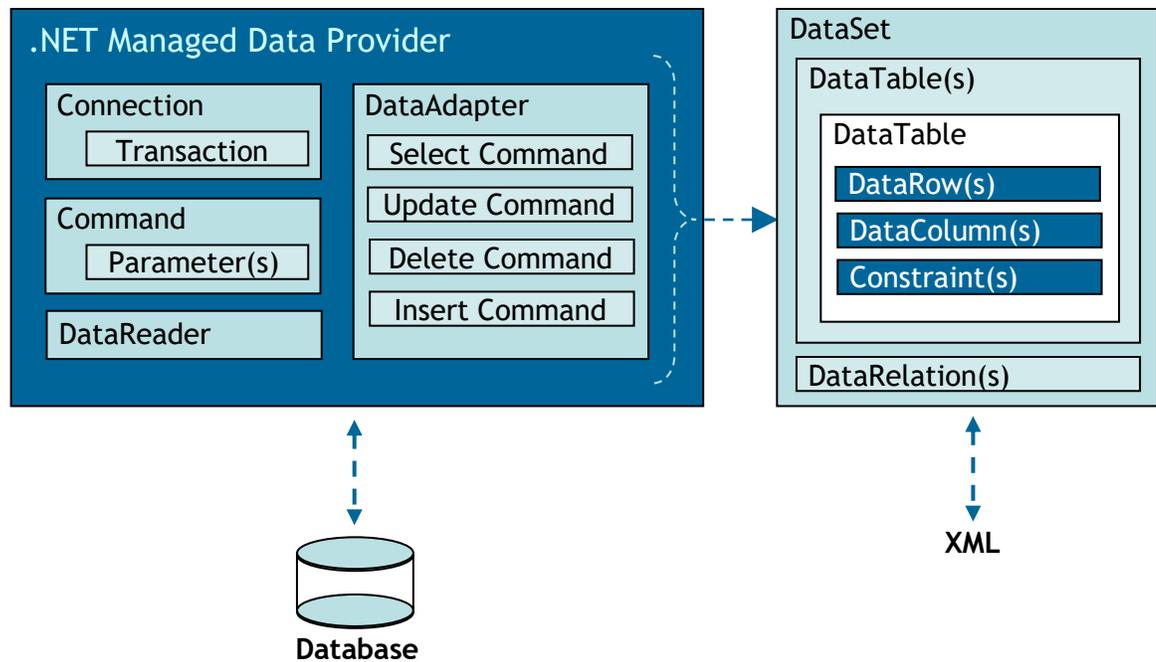
Diagram 1 : The ADO.NET Architecture

ADO.NET is an evolution of Microsoft's previous ADO data access model. ADO.NET uses some previous ADO class names, such as the Connection and Command, but also introduces many new classes, the key ones being the DataSet, DataReader, and DataAdapter.

The key difference between ADO.NET and previous Microsoft data architectures is the existence of the 'DataSet' class. This class introduces the concept of a separate and distinct level of data repository from the source data store (database). Because of this, the DataSet functions as a standalone entity and may thus be regarded as an always disconnected recordset with no knowledge of the source or eventual destination of the data it contains

The DataSet is comprised of entities which mimic the traditional database paradigm, containing such things as tables, columns, relationships, constraints and views.

A key concept within ADO.NET is that of the 'DataAdapter' class connecting to the database in order to fill the DataSet with data. Upon data update, it can then connect back to the database in order to persist the updates.

Historically, data maintenance has been primarily connection-based. However, in an attempt to make multi-tiered apps more efficient, data processing is favoring a

message-based approach revolving around the exchange of chunks of information.

The DataAdapter lies at the heart of this approach, providing a bridge to retrieve and save data between a DataSet and its source data store. It accomplishes this by the use of various 'Command' objects, each of which being configured by the developer to contain the requisite database manipulation commands in order to interact with the data store in the desired manner.

The DataSet is engineered heavily around the storage of data in XML format, providing a consistent programming model able to work with broad range of data storage products: flat, relational, and hierarchical. It does this by not recording any information relating to the source of its data, and by representing the data that it holds as collections and data types. Irrespective of the actual source of the data within the DataSet, its contents are manipulated through the same set of standard APIs exposed through the DataSet and its subordinate objects.

While the DataSet has no knowledge of the source of its data, ADO.NET revolves around the concept of a 'managed data provider', which, conversely, has very detailed and specific information relating to the data source. The role of the managed data provider is to connect, fill, and persist the DataSet content to and from data stores. The concept of a managed data provider manifests itself as a series of interfaces; these interfaces need to be implemented by a developer in order to provide the database specific logic which ultimately allows the database neutral functionality of the DataSet to be connected to a specific data store in order to provide data persistence.

Thus, in summary, ADO.NET consists of the following conceptual objects, the implementation of which is provided partly generically by the .NET framework and partly by the database vendor/integrator.

Connections

Connections are used to 'talk to' databases, and are represented by provider-specific classes such as mvConnection in the case of mv.NET. Connections can be opened explicitly by calling the Open method of the connection, or will be opened implicitly when using a DataAdapter

Commands

Commands contain the information that is submitted to a database, and are represented by provider-specific classes such as mvCommand. A command can be a stored procedure call, a database DML statement, or a statement that returns

results. You can also use input and output parameters, and return values as part of your command syntax. Commands travel over connections and resultsets are returned in the form of streams which can be read by a DataReader object, or pushed into a DataSet object.

DataReaders

The DataReader object is somewhat synonymous with a traditional read-only/forward-only cursor over data. The DataReader API supports flat as well as hierarchical data. A DataReader object is returned after executing a command against a database.

DataSets

The DataSet object represents a cache of data, with database-like structures such as tables, columns, relationships, and constraints. However, though a DataSet can and does behave much like a database, it is important to remember that DataSet objects do not interact directly with databases, or other source data. This allows the developer to work with a programming model that is always consistent, regardless of where the source data resides.

DataAdapters

The DataAdapter object works as a bridge between the DataSet and the source of data, pulling data into the DataSet, and reconciling (pushing) data back into the database.

The DataAdapter object uses commands to update the data source after changes have been made to the DataSet. Using the 'Fill' method of the DataAdapter calls the SELECT command; using the 'Update' method calls the INSERT, UPDATE or DELETE command for each changed row. You can explicitly set these commands in order to control the statements used at runtime to resolve changes, including the use of stored procedures.

The following URL (at the time of writing this document) contains further information on the ADO.NET architecture:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconoverviewofadonet.asp

Adapter Objects Overview

mv.NET's Adapter Objects product provides the developer with a range of components designed to allow efficient ADO.NET based access to multivalued databases. This chapter outlines the major aspects of Adapter Objects.

Component Overview

In order to provide a comprehensive ADO.NET solution, Adapter Objects provides the following 2 groups of components:

- Multivalued database specific Implementations of the ADO.NET classes/interfaces

- Visual Studio.NET addin components to aid developer productivity in the use of Adapter Objects

ADO.NET Implementation

In order to present a multivalued oriented ADO.NET managed data provider, Adapter Objects provides the developer with the following mv.NET specific classes, most of which inherit from the corresponding .NET framework IDbxxx interface:

mvCommand
mvConnection
mvConnectionString
mvDataAdapter
mvDataReader
mvParameter
mvParameterCollection
mvTransaction

Visual Studio Integration

In order to ease the use of Adapter Objects within the Visual Studio IDE, Adapter Objects provides a range of VS.NET extensions which are used in various places within the IDE:

- mvDataAdapter creation wizard – invoked when an mvDataAdapter is dropped onto a form or when an existing mvDataAdapter is reconfigured.

- DataSet schema generation – invoked when the Generate Typed DataSet option within the Properties window is clicked.

- Customer property designers for a range of class properties.

Important Prerequisites

In order to use Adapter Objects the developer MUST be familiar with the concepts intrinsic with the ADO.NET model. It is, therefore, strongly recommended that developers new to ADO.NET do some background reading on the subject matter prior to their use of Adapter Objects. MSDN is a good starting point for this research.

Also, in order for Adapter Objects to correctly interact with multivalued data, extended dictionary information for each field being accessed via Adapter Objects

must be created along with the creation of general file properties. Please refer to the Dynamic Normalization chapter for more details on this topic.

Data Typing

ADO.NET, and more specifically, DataRow values are strongly typed data containers. This is in sharp contrast to the somewhat relaxed attitude that MultiValued databases take to data storage.

The consequence of this difference in approach is that you need to make sure that the data which you are requesting to be held within ADO.NET conforms to the data type specification within the extended dictionary definition.

Where a field holds a blank (empty string) value within the MultiValued database, this will be represented by a <null> value within the DataRow, unless the field is defined as Alphanumeric, in which case a blank string value will be used.

Adapter Objects does incorporate code which attempts to coerce numeric data into strict numeric representation, but this will result in invalid fields being represented by a value of zero.

The mvConnection Class

The mvConnection class is responsible for establishing and holding a connection to a multivalued database. This chapter covers the Adapter Objects specific implementation of this class.

Class Overview

ADO.NET works primarily on the principle of only acquiring physical connections to the database when data transfer is actually required. Therefore, the primary purpose of the mvConnection class is to make connections to multivalued database available to ADO.NET at the time when it requires them.

The mvConnection class utilizes the functionality of mv.NET's Core Objects package to acquire database connections thereby allowing it to leverage Core Objects' implicit connection pooling capabilities.

The mvConnection class implements the IDbConnection interface. Please refer to the MSDN documentation for further information on the architecture of this interface.

mvConnection Members

The table below lists the members of the mvConnection interface that are peculiar to Adapter Objects. More detailed documentation can be found in the on-line help integrated within Visual Studio.

Name	Description
mvConnection	Constructor : The constructor of the mvConnection class allows a connection string to be passed into the object. Please see ConnectionString section below for more details.
Account	Property : Returns a reference to the CoreObjects.mvAccount instance used internally by this class. This reference will only be available after the class' Open method has been invoked.
Close	Method : Forces the Logout method of the internal mvAccount instance to be invoked and sets the ConnectionState property of the object to Closed.
ConnectionString	Property : This property allows you to specify the database into which to object is to connect. Please refer to the following section for details on the required format of this property.
Open	Method : Allows a connection to the specified database to be established. Internally, the mvCommand object establishes a CoreObjects.mvAccount instance. An exception will be raised if the ConnectionString property contains an invalid format or if the Open request fails. Upon successful open the ConnectionState property of the mvConnection object is set to Open.
Transaction	Property : Allows an mvTransaction object to be assigned to this connection.

ConnectionString Property

The ConnectionString property of the mvConnection class allows you to specify the database into which the object is to establish a connection. It should be of one of the following 2 formats:

```
Login=lpn;user=un;password=pw
```

Or

```
Server=spn;Account=apn;user=un;password=pw
```

The first format (which is the recommended format) allows you to specify the name of a login profile within the mv.NET's configuration database in order to indicate which database is to be connected into. Please refer to the Core Objects guide for further details on the topic of manipulating the configuration database. The user and password settings are optional and only need supplying if the server profile referenced by the specified login profile requires a user name or password to be supplied which is not supplied by the associated account profile.

The second format allows you to specify the server and account profile directly. Again, as with the first format, the user and password settings are optional.

There are 2 additional elements that may be included within the connection string:

`ConfigDBLocation=`***`path`***

This allows the location of the configuration database to be specified. The value of ***`path`*** should be the fully qualified path (URN) of the folder which contains the CONFIGURATION folder

`Gateway=`***`url`***

This allows the location of the required gateway service to be specified. Please refer to the Gateways chapter of the Core Objects guide for further details on Gateways.

The mvCommand Class

The mvCommand class is responsible for holding the definition of a range of possible database manipulation commands. This chapter details the aspects of the class that are peculiar to the Adapter Objects implementation of the Command class.

Class Overview

In order to provide a database neutral data access paradigm, the ADO.NET architecture abstracts all database platform specific details (in terms of data retrieval and manipulation) into the Command class. In Adapter Objects, this is represented by the mvCommand class.

The mvCommand class implements the IDbCommand interface. Please refer to the MSDN documentation for further information on the architecture of this interface.

mvCommand Members

The table below lists the members of the mvCommand interface that are peculiar to Adapter Objects. More detailed documentation can be found in the on-line help integrated within Visual Studio.

Name	Description
mvConnection	Constructor : The constructor of the mvCommand class allows an mvConnection instance to be passed into the object.

CommandText	Property : Holds the syntax of the database command associated with the object. See [following section](#) for more details on the supported syntax for this property.
CommandType	Property: Indicates the general type of the command.

CommandText Property

The contents of the CommandText property will vary according to the setting of the CommandType property and also the context in which it is being used.

CommandText for 'Text' Types

For a CommandType property setting of Text, the CommandText property needs to hold the syntax of the command to be run against the associated database. In this situation, the usage context will be one of:

Select command
Update command
Insert command
Delete command

Adapter Objects supports 2 different styles of syntax for CommandText commands: a proprietary mv.NET syntax (termed 'native' syntax) and standard ANSI SQL syntax (termed 'SQL' syntax).

The native syntax is described first followed by the SQL syntax equivalent.

Native CommandText Syntax

For each of the above commands, the CommandText property needs to be set to a semicolon separated list of 'command segments'.

Select Command

For Select commands, the CommandText native syntax is as follows:

```
Select;File=fn;Criteria=sel;Sort=srt;Fields=fld
```

or

```
Select;File=fn;ID=id;Fields=fld
```

For example:

```
Select;File=SALESORDER;Criteria=CUSTOMER < "10";Fields=NUMBER
CUSTOMER CUSTOMERNAME PRODUCT DELIVERYQTY DELIVERYDATE;Sort=BY
NUMBER
```

Where:

fn represents the name of the file from which to select items

sel represents a multivalue format selection clause, e.g. `CUSTOMER = "850"`

srt represents a multivalue format sort clause, e.g. `BY CUSTOMERNAME`

id represents an attribute mark, char(254), delimited list of item IDs

fld represents a space separated list of the required field (dictionary) names

The select command may also have the segments `;Normalized` or `;MVFilter` appended to it in order to control dynamic data normalization and multivalued data filtering (respectively). Please refer to the [Dynamic Normalization](#) chapter for further details on these topics.

For TableDirect command types **only**, you may control how multivalue and subvalue marks are handled when data is extracted by the use of 2 extra segments in the select command:

```
;ReplaceVM=vmrepl;ReplaceSVM=svmrepl
```

Where *vmrepl* represents the character string to replace multivalue marks and *svmrepl* represents the character string to replace subvalue marks. If you wish to use control characters in the replacement string, you should use the following format:

```
~c1~c2...
```

Where `c1` is the ASCII value of the first required character, `c2` is the second, etc. You may concatenate as many ~n characters as required. For example:

```
ReplaceVM=~13~10
```

indicates that each multivalue mark is to be replaced by a carriage return line feed character pairing.

If you wish to use a semicolon within the replacement string, you should use `/;` to indicate this, for example:

```
ReplaceSVM=/;
```

indicates that each subvalue mark is to be replaced by a semicolon character.

Within the Criteria and Sort segments of the select command you may specify run-time parameters as follows:

```
Criteria=CUSTOMER = "{CUSTOMER}"
```

In the above example, `{CUSTOMER}` denotes a parameter called 'CUSTOMER' that will require an mvParameter instance creating within the command object's parameter collection. Note, if you use the Data Adapter creation wizard this will be done automatically for you. See the [Data Adapter Wizard chapter](#) for more details.

The final segment which may be included within the Select command is the `;AutoLink` segment. If present, this indicates that for select commands which, in fact, contain multiple selection commands, if any of the files referenced in the set of selection commands contain foreign key links to one another, relationship information will be automatically created within a host DataSet. See the [Multiple Commands](#) chapter for more details.

Update Command

For Update commands, the CommandText native syntax is as follows:

```
Update;File=fn;ID=id;Set=fld To val;UpdateControl=uc
```

Where:

fn represents the name of the file which is to be updated
id represents the ID of the item which is to be updated
fld and *val* represent a field name and value pairing indicating how a specific field is to be updated – see below for more details.
uc represents a space separated list of field names that control how optimistic lock control is to be managed – see below for more details.

The ID segment will typically reference a run-time parameter which is mapped to the column holding the item ID. For example:

```
ID={Product ID}
```

The Set segment(s) of the update command allows you to specify how one or more fields are to be updated. Multiple Set segments can be included as necessary and, typically, the value portion will be a run-time parameter. For example:

```
Set=TYPE To C;Set NAME To {NAME}
```

The above set segments indicate that the TYPE field is to be set to a value of 'C' and that the value of the NAME field is to be set to the value of parameter 'NAME'. If you need to include a semicolon within the value portion you should use the string "/;" to do so.

The update control segment of the update command takes the form of a space separated list of field names. The presence of a field name in this list indicates that the value of this field will participate in the optimistic lock checking process which is automatically performed by Adapter Objects in order to coordinate multi-user access to the database. There are 2 things to note with update control segments:

1. If the command object is associated with a DataAdapter, each field within the update control segment must also have been included in the Fields segment of the select command which initially populated the DataTable. This is because the update control process needs access to the original values of the fields it is checking for optimistic lock control – these original values will only available if the field has been retrieved as part of the Fields segment.

2. There must be an mvParameter object (within the command) for each field within the update control segment. Each of these parameter objects must have a name of *field*.Orig, where **field** is the name of the update control segment field name. It is the value of these parameters which are used to supply the original values of the fields specified within the update control segment. These original values are used in the optimistic lock checking process – see below. If the command object has been created using the Data Adapter wizard or the mvDataAdapter.GenerateCommnds method, these parameter objects will have been created automatically.

Optimistic locking works using the following mechanism: at the point of update, the original value of an amended update control field is passed to the database server along with the desired new value. If the original value is the same as the current database value, the update is allowed to continue, otherwise it is blocked.

The update control segment thus allows you to restrict the optimistic lock checking to only the relevant fields within the update set.

The update command may also contain a Normalized segment and multivalued/subvalue replacement segments in order to control the handling of multivalued and subvalued data. See Select Command above for details on these 2 command segments.

Please refer to chapter 'Updating Data' for detailed notes on using the UpdateCommand object.

Insert Command

For Insert commands, the CommandText native syntax is as follows:

```
Insert;File=fn;ID=id;Set=fld To val
```

Please refer the Update Command above for details on the insert command segments. As per the Update command, the Normalized and multivalued/subvalue replacement segments can also be included with the Insert command text.

Delete Command

For Delete commands, the native syntax is as follows:

```
Delete;File=fn;ID=id;UpdateControl=uc
```

Please refer the Update Command above for details on the delete command segments. The Delete command does not use Normalized and multivalued/subvalue replacement segments.

SQL CommandText Syntax

Adapter Objects supports a subset and extended form of the ANSI SQL syntax definition. The syntax supported for each of the 4 command types are detailed below.

Select Command

For Select commands, the CommandText SQL syntax is as follows:

```
SELECT fld FROM fn [WHERE sel] [ORDER BY srt [DESC]] [NORMALIZED]
[REPLACEVM 'rvm'] [REPLACESVM 'rsvm'] [MVFILTER]
```

(clauses contained within square brackets are optional)

Please refer to the Dynamic Normalization chapter for further details on the NORMALIZED and MVFILTER keywords.

For example:

```
SELECT NUMBER, CUSTOMER, CUSTOMERNAME, PRODUCT, DELIVERYQTY,
DELIVERYDATE FROM SALESORDER WHERE CUSTOMER < 10 ORDER BY NUMBER
```

Where:

fld represents a comma separated list of the required field (dictionary) names
fn represents the name of the file from which to select items
sel represents a selection clause, e.g. `CUSTOMER = "850"`
srt represents a sort clause, e.g. `ORDER BY CUSTOMERNAME`.
rvm represents the character string to replace multivalue marks when a command type of TableDirect is used.
rsvm represents the character string to replace subvalue marks when a command type of TableDirect is used.

For *rvm* and *rsvm* if you wish to use control characters in the replacement string, you should use the following format:

```
~c1~c2...
```

Where `c1` is the ASCII value of the first required character, `c2` is the second, etc. You may concatenate as many ~n characters as required. For example:

```
ReplaceVM '~13~10'
```

indicates that each multivalue mark is to be replaced by a carriage return line feed character pairing.

Within the *sel* (Criteria) and *srt* (Sort) segments of the select command you may specify run-time parameters as follows:

```
WHERE CUSTOMER = "{CUSTOMER}"
```

In the above example, `{CUSTOMER}` denotes a parameter called 'CUSTOMER' that will require an mvParameter instance creating within the command object's parameter collection. Note, if you use the Data Adapter creation wizard this will be done automatically for you. See the [Data Adapter Wizard chapter](#) for more details.

Select Command Syntax Restrictions

The SQL syntax support is a subset of the full ANSI standard. The following notes detail the supported constructs.

1. Multiple tables

Adapter Objects only supports direct retrieval from a single table (file). It is assumed that any data joining will be performed via dictionary definitions within the specified file.

2. Selection criteria

The following operators are supported:

```
NOT   AND   OR   =   >   >=   <=   LIKE   IN
```

AND and OR operators may not be both present within in the same select command, nor may IN and OR operators. Only one IN clause may be present.

Update Command

For Update commands, the CommandText SQL syntax is as follows:

```
UPDATE fn SET fldval WHERE upd [NORMALIZED] [REPLACEVM 'rvm']
[REPLACESVM 'rsvm']
```

(clauses contained within square brackets are optional)

For example:

```
UPDATE SALESORDER SET DATEPLACED = {DATEPLACED} WHERE NUMBER =
{NUMBER} AND DATEPLACED = {DATEPLACED.Orig}
```

Where:

fn represents the name of the file from which to select items

fldval represents a series of set clauses

upd represents a series of optimistic lock control conditions

rvm represents the character string to be replaced by a multivalue mark when a command type of TableDirect is used.

rsvm represents the character string to be replaced by a subvalue mark when a command type of TableDirect is used.

For *rvm* and *rsvm* if you wish to use control characters in the replacement string, you should use the following format:

```
~c1~c2...
```

Where `c1` is the ASCII value of the first required character, `c2` is the second, etc. You may concatenate as many ~n characters as required. For example:

```
REPLACEVM '~13~10'
```

indicates that each occurrence of a carriage return line feed character pairing is to be replaced by a single multivalue mark.

Within the *upd* segment of the update command you may specify run-time parameters as follows:

```
NAME = "{NAME.Orig}"
```

In the above example, `{NAME}` denotes a parameter called 'NAME' that will require an mvParameter instance creating within the command object's parameter collection. Note, if you use the Data Adapter creation wizard this will be done automatically for you. See the Data Adapter Wizard chapter for more details.

Update Command Syntax Restrictions

The SQL syntax support is a subset of the full ANSI standard. The following notes detail the supported constructs.

1. WHERE clause structure

The WHERE clause within an update command must adhere to the following rules:

a) The first condition must reference a dictionary name representing the item ID field. It is the first condition within the WHERE clause that identifies the item to be updated.

b) Second and subsequent fields allow optimistic locking criteria to be specified.

c) Second and subsequent conditions must reference non-item ID field names.

d) Second and subsequent conditions must only use the "=" operator.

e) The value part of the second and subsequent conditions must reference either a literal value or a parameter name. If a parameter name is referenced it must be of the format {*name*.Orig} where *name* is the field name, e.g:

```
NAME = '{NAME.Orig}'
```

Please refer to chapter 'Updating Data' for detailed notes on using the UpdateCommand object.

CommandText for 'StoredProcedure' Types

For a CommandType property setting of StoredProcedure, the CommandText needs to hold the name of the DataBASIC subroutine to be called on the server.

If you wish to pass/receive data to/from the subroutine via arguments, you need to add the relevant number of mvParameter objects to the mvCommand's Parameters collection – one mvParameter for each argument that the subroutine will expect to be passed in. The SourceColumn of each mvParameter should be set to the logical position (as specified in the subroutine's first line of code) of the argument it represents. The name of the mvParameter is not significant but we recommend that you set it to the same name as the variable defined in the subroutine's declaration statement in order to aid clarity.

For example, the VB code below calls a cataloged subroutine called END.OF.MONTH, passing in 3 arguments and retrieving information returned by the subroutine in the 3rd argument.

```
Dim myCommand As New mvCommand("END.OF.MONTH", myConnection)

myCommand.Parameters.Add(New mvParameter("YEAR", DbType.String, "1"))
CType(myCommand.Parameters("YEAR"), mvParameter).Value = txtArg1.Text

myCommand.Parameters.Add(New mvParameter("MONTH", DbType.String, "2"))
CType(myCommand.Parameters("MONTH"), mvParameter).Value = txtArg2.Text

myCommand.Parameters.Add(New mvParameter("STATUS", DbType.String, "3"))

myCommand.CommandTimeout = 300 ' so we can debug
myCommand.ExecuteNonQuery()

txtArg3.Text = CType(myCommand.Parameters("STATUS"), mvParameter).Value
```

 Note, AdapterObjects automatically treats parameters used in this way as InputOutput parameter types.

Also note that if the subroutine can potentially take a significant amount of time to execute, you will need to set the CommandTimeout property of the mvCommand object to the maximum period of time (seconds) that the subroutine may take to complete. If you wish to debug the subroutine via the Connection Window, you will also need to extend the CommandTimeout property to allow your debugging work to be completed.

Filling DataSets using Stored Procedures

For situations where a stored procedure (DataBASIC subroutine) is required to generate (or fill) a DataSet, the following programming pattern should be used.

If you use a StoredProcedure command type object as the SelectCommand object of an mvDataAdapter, the Fill method will invoke the specified databasic subroutine and expect the results to be returned in a set argument number in a set format.

Specifically, the CommandText property of such a command object must be of the format:

subname;File=*file name*;Fields=*field list*;Criteria=*criteria*;Sort=*sort*;
Normalized

Where:

subname is the name of the subroutine to be called
file name is the name of the file whose dictionary holds dictionary items as specified by the Fields clause
field list is the (space delimited) list of dictionary item names defining the order of data organisation (and data typing) within the returned data
criteria is the (optional) selection criteria (this is free-text and may, actually, be set to anything useful to the called subroutine)
sort is the (optional) sort definition (this is free text as above)
Normalized (if present) indicates that multi/subvalued data is to split into separate DataTables

Note, the file and field list segments do not define where the data is to be drawn from, that is for the subroutine to decide. They are only used, in this context, to define where dictionary items are to be found. These dictionary items are used to control the creation of the resulting DataTable object(s).

The subroutine specified within the CommandText must have 6 arguments defined in its calling signature:

arg#1 – (input) the name of the file; i.e. *file name* in the CommandText
arg#2 – (input) the required fields; i.e. *field list* in the CommandText (but please note that this is a VM delimited list)
arg#3 – (input) the required select criteria; i.e. *criteria* in the CommandText
arg#4 – (input) the required sort criteria; i.e. *sort* in the CommandText
arg#5 – (output) the returned data

arg#6 – (output) (optional) error message text – non-blank indicates an error

The structure of the returned arg#5 should be as follows:

item *sep* item *sep* ...

That is, a series of item strings concatenated together with a char(30) separating each one. The content of each item string must be the value of each field specified with the CommandText's *field list* segment with each value occupying one attribute position. The entire item string should also have the relevant item ID (or unique string) concatenated at the front as the first attribute irrespective of whether the item ID is specified as one of the required fields or not. For example:

```
"1":CHAR(254):"Test 1":CHAR(30):"2":CHAR(254):"Test 2"
```

The above return value returns one field value for item IDs "1" and "2".

Filling DataSets using Stored Procedures which Supply Schema Data

For situations where the developer needs to be shielded from all aspects of data retrieval (including the names of files and fields), there is an alternative CommandText syntax to the above stored procedure calling mechanism:

subname;Context=some user data;Normalized

As with the previously discussed syntax for a stored procedure call, the 'Normalized' keyword is optional and indicates (if present) that multi/subvalued data is to split into separate DataTables

The "Context" keyword indicates that the subroutine will be called in 2 different modes; a schema retrieval mode and a data retrieval mode. The string supplied with the Context keyword can be anything which allows the subroutine to uniquely identify the particular context in which it being called, for example a record ID or functional area. mv.NET will decide when and in which mode the subroutine is called. The mode can be detected by the contents of the subroutine's first argument which will be set to either "Get schema" or "Get data". The signature of the subroutine should still support 6 arguments as above. When the subroutine is called in a data retrieval mode, the use of these 6 arguments is exactly the same as above, but when called in a schema retrieval mode, the use of these 6 arguments is as follows:

arg#1 – (input) contains the string "Get schema"

arg#2 – (input) the context as supplied in the CommandText property

arg#3 – (input) empty string (not used)

arg#4 – (output) the name of the file being used (the dictionary of which holds the schema)

arg#5 – (output) the space separated list of required field (column) names

arg#6 – (output) (optional) error message text – non-blank indicates an error

NOTE: when using this stored procedure calling syntax, an mvConnection instance must be passed into the mvCommand constructor. If an mvConnection is not available at the time of mvCommand construction, an extra segment needs inserting at the **beginning** of the CommandText string:

```
Login={login name}
```

e.g.

subname`;Login=SOP;Context=some user data;Normalize`

This extra segment indicates which login profile is to be used for the mvConnection that will be created internally on a temporary basis by the mvCommand construction process in order to allow the retrieval of schema information via the specified subroutine.

Below is an example subroutine which illustrates this principle. Note, the data is hard-coded here, whereas in reality it would be typically selected dynamically from a file.

```
SUBROUTINE ADAPTERFILL (ARG1, ARG2, ARG3, ARG4, RETURNDATA, ERRMSG)
*
    IF ARG1 = 'Get schema' THEN
        ARG4 = 'SALESORDER'
        RETURNDATA = 'NUMBER CUSTOMERNAME PRODUCT'
    END ELSE
        AM = CHAR(254)
        SEP = CHAR(30)
        RETURNDATA = '3':AM:'3':AM:'Leather Logistics Associates
         Corp.':AM:'YSS61²WTM43²BSS69²MAP63':SEP:'4':AM:'4':AM:'Carpets
         Technology Brokers Corp.':AM:'G4M11²BFS86²RRG1²BAS73²PSS67'
    END
*
    RETURN
```

CommandText for 'Table Direct' Types

For a CommandType property setting of TableDirect, the CommandText property needs to hold the syntax of the command to be run against the associated database. This is exactly the same syntax as per a CommandType setting of 'Text'.

The difference between a TableDirect and a Text CommandType is that the TableDirect does not split (normalize) multivalues or subvalued data into separate rows/tables. VM and SVM marks are left within the data, although you do have the ability to control how these characters are transformed for display purposes if required – please refer to the Select Command section for further details on this capability.

Accessing Schema Information

The GetFullSchema function of the mvCommand class returns a DataSet describing the characteristics of each column within each DataTable returned by a select command.

Each DataTable contains the following column names:

```
ColumnName
ColumnOrdinal
ColumnSize
NumericPrecision
NumericScale
DataType
ProviderType
IsLong
AllowDBNull
IsReadOnly
IsRowVersion
IsUnique
IsKeyColumn
IsAutoIncrement
BaseSchemaName
BaseCatalogName
BaseTableName
BaseColumnName
```

Each row within each DataTable represents a column within the resultant DataTable and describes the characteristics of the associated column.

Dynamic Normalization

This chapter outlines the Dynamic Normalization technology incorporated within Adapter Objects. This addresses the problem of how to transform multi and subvalued data into ADO.NET data structures.

The Need for Normalization

Because multivalued databases typically contain data structures incorporating nested (multivalued) data, an important requirement for an ADO.NET managed data provider is to support a mechanism whereby this nested data is 'flattened' to fit the 2-dimensional table structures of ADO.NET.

There are 2 basic approaches to producing flattened data; either transform all data on a regular/scheduled basis so that there is a permanently flattened version of the data available to satisfy selection requirements; or, alternatively, perform this flattening process dynamically in real-time.

Adapter Objects adopts the latter of these 2 approaches in order to provide a real-time data feed to applications and also to prevent the creation of duplicate data repositories. The term given to this process within Adapter Objects is 'Dynamic Normalization'

Dynamic Normalization Overview

There are several challenges to be met when moving data (bidirectionally) from a multidimensional data source to a 2-dimensional repository:

a) Multivalued fields have to be split into multiple rows or multiple related files.

b) Multivalue associations have to be represented/mirrored in the 2-dimensional repository.

c) Physical ordering of multivalued data has to be preserved.

d) What ever approach is taken to the above 3 issues, it must be possible to reconstitute a correct multivalued representation of the data in order to allow updates of the original multivalued data source to be performed.

The task of representing multivalued associations within the ADO.NET environment is eased somewhat by the fact that the DataSet class supports the concept of relationships; i.e. it not only allows data to be held, but also the definition of the relationships between the data. Thus, the end result of the dynamic normalization process is typically a dataset with a number of tables related together in a manner which reflects the nature of the multivalued data with the source file.

The Use of Extended Dictionary Data

The dynamic normalization process within Adapter Objects draws upon a number of extended dictionary definition areas which may be created and maintained by the Data Manager utility. These areas are as follows:

File Properties

In the Data Manager's treeview list of files within an account, you may right-click a file name and select the *Properties* option to view a range of general details pertaining to the file. At the bottom of Properties window is an area titled 'Adapter Objects Name Mappings'. Within this area you may specify the DataTable name by which the file will be known within the ADO.NET environment and also the names of the normalized tables that will be produced as a result of the dynamic normalization process.

Each multivalued and subvalue group defined within the mv.NET schema for the file is represented by a row within the Group Data Tables Names grid. You are able to define the Table name of the ADO.NET DataTable that will be created to hold each multivalued/subvalued group data. You are also able to define the name of the column which is created within the DataTable to hold the ordinal multivalued/subvalue position of each individual nested data element.

Dictionary Schema

Within the Extended tab of the Data Manager's Schema Maintenance window for a file, you are able to define the Adapter Column name for each field. This name is used as the column name for the field whenever it is represented within a DataTable within the ADO.NET environment.

In addition to defining a column name it is important to make sure that all multivalued/subvalued fields are both flagged as being multi/subvalued (via the MV Type field) and are also assigned an MVGroup/SVGroup name within the extended dictionary definition.

It is the presence of an MVGroup/SVGroup name in conjunction with an MV Type setting of Multivalued or Subvalued which indicates to Adapter Objects that dynamic normalization is relevant for a particular field.

It is also important to make sure that the Data Type setting within the extended definition is correct for each data field.

Using Dynamic Normalization

In order to trigger dynamic normalization within Adapter Objects, you will need to add the `;Normalized` segment to all of your commands. See Text Command Types section in the mvCommand Class chapter for more details on this. Thus, when the command is invoked (using either the Fill or Update method of the mvDataAdapter class), dynamic normalization will be used automatically. If you are using SQL syntax, you need to add the `NORMALIZED` keyword to the end of your SQL command.

When invoked, dynamic normalization will (where relevant) produce multiple related DataTables within the host DataSet. You may then use the standard ADO.NET methods of navigating parent/child rows within DataSet as necessary.

If you add or delete rows within a dynamically normalized (child) table, Adapter Objects will automatically maintain the ordinal multivalue/subvalue position columns ready for when the data is de-normalized and written back to the multivalue database.

If the Normalized segment/keyword is omitted from the command text, multivalues and subvalues will be 'exploded' into multiple rows. i.e. a 1st normal form simulation.

Exploding and Filtering Multivalued Data

If the `;Normalized` segment is omitted from the CommandText property and the CommandType is set to Text, data will be returned in 'exploded' 1st normal form. In such a scenario, the `;MVFilter` segment may be appended to the CommandText property to indicate that selection criteria should be applied to the exploded data in order to filter out those multivalues/subvalues that do not pass the selection criteria. If you are using SQL syntax, the MVFILTER keyword should be used.

Note, the Normalized and MVFilter functionality are mutually exclusive. Also note that MVFiltered data may only be used for read-only purposes.

Updating Data

The topic of updating database information warrants special treatment within this guide. This chapter explains the different ways in which you may update MultiValued data from within the ADO.NET environment.

The Update Options Available

There are 2 basic ways of updating data via Adapter Objects:

1. Using the mvDataAdapter.Update method on a DataSet which has been previous populated using the mvDataAdapter.Fill method. The data adapter's selection command must include the 'Normalized' keyword. Note, the data adapter must have an UpdateCommand configured for this approach to work.

2. Creating a standalone UpdateCommand and using its ExecuteNoQuery method.

Updating via the mvDataAdapter.Update Method

The mvDataAdapter's Update method requires that its UpdateCommand property references a correctly configured mvCommand object. It is this command object which is used to perform the update processing.

Adapter Objects' UpdateCommand object performs its updating using a series of principles and assumptions. Understanding these will allow you to understand how you may use it to update your MultiValued data. These principles/assumptions are as follows:

1. The CommandText contains one or more 'Set' segments indicating which fields are to be updated.

2. The CommandText (optionally) contains an 'UpdateControl' segment indicating which fields are to be used to control optimistic locking.

3. If the UpdateControl segment is used, the UpdateCommand must contain a parameter within its Parameters collection for each field specified within the UpdateControl segment. The name of each parameter should be *field*.Orig where *field* is the name of the field as specified within the UpdateControl segment. Its value needs to be the original value of the field as retrieved initially from the database.

If you wish to update individual multivalue and subvalue data elements, the SelectCommand of the data adapter (which was used to originally populate the DataSet) must have a CommandText which includes the 'Normalized' keyword. The inclusion of this keyword forces Adapter Objects to split the retrieved data into several related hierarchical DataTables.

The DataTables of the DataSet created by this process include special columns that allow Adapter Objects to keep track of which multivalue and subvalue position each piece of non-singular data relates to. If you insert or delete rows of data within a DataTable which represents multivalued or subvalued data, AdapterObjects will automatically maintain the relevant multivalue/subvalue position column data.

Thus, when the Update of the data adapter is used, Adapter Objects is able to recompose the item data ensuring that the correct physical ordering of multivalued and subvalued data is honored.

If you are only updating singular data fields, the Normalized keyword is optional.

Creating an Update Command Object

If you have configured your mvDataAdapter using either the Data Adapter Wizard within the Visual Studio design environment or using the mvDataAdapter.GenerateCommands method (see following section), an UpdateCommand object will be automatically created within your data adapter.

The syntax of this generated update command will assume that all retrieved fields will need updating with current values and that all fields are to be checked as part of the optimistic lock control process.

If you have manually instantiated and configured the SelectCommand of your data adapter, then you will need to manually create the UpdateCommand object and configure it using the guidelines listed in the previous section. The following section discusses this topic in detail.

The mvDataAdapter.GenerateCommands Method

The mvDataAdapter class has a method called 'GenerateCommands'. This method can be used to programatically generate default Select, Update, Insert and Delete command objects within a data adapter based on the contents of a supplied select statement.

Using a Manually Created Update Command

If you wish to perform an update using an mvCommand object which you have instantiated and configured programmatically, there are 4 possible valid scenarios:

1. Using the mvDataAdapter.Update method, you are updating fields (some of which contain multivalued or subvalued data) within a DataSet which has been previously populated using a select command which includes the 'Normalized' keyword.

2. Using the mvDataAdapter.Update method, you are updating only singular field information within a DataSet which has been previously populated using a select command which may or may not have included the 'Normalized' keyword.

3. Using the mvCommand.ExecuteNonQuery method, you are updating fields, some of which contain multivalued or subvalued data. The mvCommand object has a CommandType set to TableDirect

4. Using the mvCommand.ExecuteNonQuery method, you are updating singular data fields only.

All of the above valid scenarios may be augmented with the use of optimistic locking via an UpdateControl segment.

Each of these 4 scenarios is discussed below:

Scenario 1: Multivalued data update within a normalized DataSet

In this scenario, the update command object must use a CommandText property which contains the appropriate Set segments and the Normalized keyword. If the UpdateControl segment is used, the command object must also contain the relevant parameter objects as described in the previous section.

Scenario 2: Singular data update within a DataSet

In this scenario, the update command object must use a CommandText property which contains the appropriate Set segments. If the UpdateControl segment is used, the command object must also contain the relevant parameter objects as described in the previous section.

Scenario 3: Multivalued data update via ExecuteNonQuery

In this scenario, the update command object must use a CommandText property which contains the appropriate Set segments and the ID segment to identify the relevant item ID to be updated. If the UpdateControl segment is used, the command object must also contain the relevant parameter objects as described in the previous section. The command object's CommandType should be set to TableDirect and the multivalue/subvalue marks should be included within the supplied data to the Set segments. The ReplaceVM and ReplaceSVM segments may be used to automatically convert printable characters in system delimiters.

Scenario 4: Singular data update via ExecuteNonQuery

As per scenario 2.

Multiple Commands

The mvCommand object allows you to define multiple commands to be performed in unison. This chapter explains this process and explores the implications of doing so.

Defining Multiple Commands

Within a single command object, you may set the CommandText property to multiple 'back-to-back' commands with each command being separated by the string '/ /'. The data adapter definition wizard allows multiple selection commands to be defined, with corresponding multiple update, insert and delete commands being generated automatically.

Select Command Execution

Each individual selection command is executed in isolation but after all commands have been performed, the resultant DataTables are related based on foreign key dependencies defined within the extended dictionary definitions.

Update Processing

The Update method of the mvDataAdapter class processes each DataTable within the DataSet, applying the correct update, insert and delete commands as necessary.

The Data Adapter Definition Wizard

This chapter describes the wizard which is invoked when you drag and drop an mvDataAdapter control onto the surface of a form within Visual Studio.

Invoking the Wizard

Within the Data tab of Visual Studios' Toolbox window are 3 Adapter Objects controls:

mvDataAdapter
mvCommand
mvConnection

If you drag and drop the mvDataAdapter control onto a form, the Adapter Objects' Data Adapter wizard will be invoked.

Once you have created an mvDataAdapter instance, you may re-invoke the wizard by clicking the Define Adapter link within the Properties window when the data adapter is selected.

Wizard Steps

In order to define your data adapter, the wizard guides you through a series of steps:

Step 1 : Define your data source

This step allows you to specify which database is to accessed by the data adapter. You can either select the name of a login profile which you have previously defined within the Data Manager or, if you already have one or more mvConnection instances defined within the form, select the name of an existing mvConnection instance. The Define new connection button shown in this step allows you to define a new login profile name, or allows you to invoke the Data Manager on order to define new server/account profile definitions.

Step 2 : Define your selection command(s)

This step allows you to specify which database is to be accessed by the data adapter. You may define multiple selection commands if required – see Multiple Commands chapter for more details on this topic. For each command, you may define:

- the source data file
- the selection criteria
- the sort criteria
- the list of fields that you wish to extract from the source data file
- whether you wish to invoke dynamic normalization
- multivalue/subvalue character translation (if dynamic normalization is not used)

Sample Application

The installation of the mv.NET Client Interface Developer product will install a sample application illustrating the use of many of the features described in this manual. This project may be found in the following location:

`C:\Documents and Settings\All Users\Application Data\BlueFinity\mv.NET\Version4.0\Examples`

Or, for Vista/Win7/Server2008 systems:

`C:\ProgramData\BlueFinity\mv.NET\Version4.0\Examples`

VB.NET and C# sample code is provided.

The following sections of this chapter cover each WinForm component within the application.

frmWizard (not in the VS2010 sample)

This form contains example use of the following features:

1. The drag and drop of an mvDataAdapter control onto a form with the resulting invocation of the data adapter wizard.

2. The generation of typed datasets.

3. The passing of datasets into the DataGrid control.

4. The passing of datasets into Crystal Reports.

5. The manual construction of mvDataAdapter and mvCommand objects.

frmStandalone/frmMain

This form contains example use of the following features:

1. The use of the mvDataReader class.

2. The use of the TableDirect command type.

3. The use of the StoredProcedure command type.

4. The generation of XML data from datasets.

5. The manual creation and use of Update, Insert and Delete commands.