

mv.NET Adapter Objects



Developer's Introductory Guide

A product from BlueFinity



Copyright Notices

Copyright BlueFinity 2004
Document ref: mvNet_BO_01
Revision 1.0
All rights reserved BlueFinity 2004

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
Telephone: +44 (0) 1442 450 435
Address: Hamilton House
111 Marlowes
Hemel Hempstead
Hertfordshire HP1 1BB
United Kingdom

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

mv.NET Adapter Objects	1
Copyright Notices.....	2
Contacting Us.....	2
Trademark Acknowledgements.....	2
Welcome to mv.NET	1
The mv.NET Family of Products.....	1
Feature Overview.....	2
The mv.NET Suite	2
Getting Started Guide Contents	3
ADO.NET Basics	4
Installation	4
ADO.NET Architectural Summary	4
Connections.....	6
Commands	6
DataReaders	7
DataSets	7
DataAdapters.....	7
Adapter Objects Overview	8
Component Overview	8
ADO.NET Implementation	9
Visual Studio Integration.....	9
Important Prerequisite.....	9
The mvConnection Class	10
Class Overview	10
mvConnection Members	10
ConnectionString Property.....	11

The mvCommand Class	13
Class Overview	13
mvCommand Members	13
CommandText Property	14
CommandText for 'Text' Types	14
Select Command	14
Update Command	16
Insert Command	17
Delete Command	17
CommandText for 'StoredProcedure' Types	17
CommandText for 'Table Direct' Types	17
Dynamic Normalization	18
The Need for Normalization	18
Dynamic Normalization Overview	18
The Use of Extended Dictionary Data	19
File Properties	19
Dictionary Schema	20
Using Dynamic Normalization	20
Multiple Commands	21
Defining Multiple Commands	21
Select Command Execution	21
Update Processing	21
The Data Adapter Definition Wizard	23
Invoking the Wizard	23
Wizard Steps	23
Step 1 : Define your data source	24
Step 2 : Define your selection command(s)	24

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 multivalued 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 start-of-the-art, feature rich .NET environment. Its design also enables and encourages the developer to leverage, wherever possible, previously acquired multivalued skills.

BlueFinity's team of software engineers has huge knowledge and experience of using both multivalued 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 mv developers to harness the full power of both their mv 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 mv databases.
- **Binding Objects** – high performance databinding technology that enables standard .NET controls to become fully mv-aware. Binding Objects links with Core Objects to provide its functionality.
- **Adapter Objects** – complete implementation of an ADO.NET managed data provider for multivalued 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 mvCommand Class](#)

This chapter describes mv.NET's databinding concepts, as well as providing an overview of the primary components that are supplied as part of the Binding Objects module.

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 constructed.

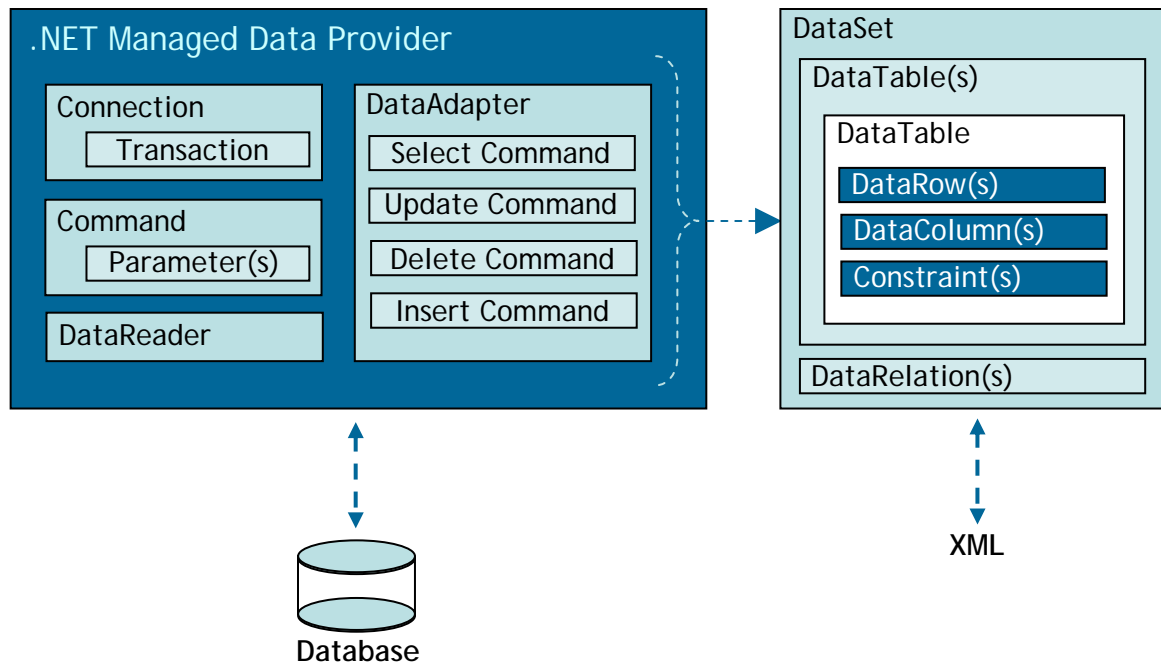


Diagram 1 : The ADO.NET Architecture

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, which introduces the concept of a separate and distinct level of data repository from any 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 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, an 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 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 Prerequisite

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.

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, which thus, allows it to leverage Core Object's 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.

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

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 property syntax is as follows:

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

Where:

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

sel represents a multivalued selection clause, e.g. CUSTOMER = "850"

srt represents a multivalued sort clause, e.g. BY CUSTOMERNAME

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

The select command can also have the segment `;Normalized` appended to it in order to indicate that dynamic data normalization is required. Please refer to the [Dynamic Normalization](#) chapter for further details on this topic.

If dynamic normalization is not specified, you may control how multivalued 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 multivalued 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 multivalued 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 runtime 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 property 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 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'.

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. Note, if a field is present within the update control segment of an update command, it must also have been included in the Fields segment of the select command which initially populated the DataTable.

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 or 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.

Insert Command

For Insert commands, the CommandText property 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 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.

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.

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.

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.

The Need for Normalization

Because multivalued databases typically contain data structures incorporating nested (multivalued) data, an important requirement for a 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, this flattening process can be performed 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 a duplicate data repository. 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 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.

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.

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.

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 be 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 in 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)