Contents

.NET Data Provider Guide

July 2005

Version 9.1

This guide gives an overview of the .NET Data Provider. It describes how to set up and use the .NET Data Provider to access a Dharma SDK database from .NET applications.

July 2005

Dharma Systems welcomes your comments on this document and the software it describes. Send comments to:

Documentation Comments

Dharma Systems, Inc.

Brookline Business Center.

#55, Route 13

Brookline, NH 03033

Phone: 603-732-4001

Fax: 603-732-4003

Electronic Mail: support@dharma.com

Web Page: http://www.dharma.com

Introduction

PURPOSE OF THIS GUIDE

This guide gives an overview of the Dharma SDK .NET Data Provider. It describes how to set up and use the Dharma SDK .NET Data Provider. The Dharma SDK .NET Data Provider provides access to Dharma SDK databases from .NET applications

AUDIENCE

This guide is directed towards .NET programmers writing database applications using the Dharma SDK. It assumes knowledge of the .NET Framework and .NET programming languages.

STRUCTURE

This guide contains the following chapters:

Chapter 1	Gives an overview of Microsoft .NET Technology and introduces the Dharma SDK .NET Data Provider.
Chapter 2	Describes installation and configuration requirements for the Dharma SDK .NET Data Provider.
Chapter 3	Describes the features of the Dharma SDK .NET Data Provider.
Chapter 4	Describes various steps involved in developing applications using the Dharma SDK .NET Data Provider.
Chapter 5	Contains reference on the Dharma SDK .NET Data Provider specific classes.
Appendix A	Lists the error messages in the Dharma SDK .NET Data Provider.
Appendix B	Contains sample programs using the Dharma SDK .NET Data Provider.
Appendix C	Contains a glossary of terms you should know.

SYNTAX DIAGRAM CONVENTIONS

UPPERCASE	Uppercase type denotes reserved words. You must include reserved words in statements, but they can be upper or lower case.
lowercase	Lowercase type denotes either user-supplied elements or names of other syntax diagrams. User-supplied elements include names of tables, host-language variables, expressions and literals. Syntax diagrams can refer to each other by name. If a diagram is named, the name appears in lowercase type above and to the left of the diagram, followed by a double-colon (for example, privilege ::). The name of that diagram appears in lowercase in diagrams that refer to it.
{ }	Braces denote a choice among mandatory elements. They enclose a set of options, separated by vertical bars (|). You must choose at least one of the options.
[ ]	Brackets denote an optional element or a choice among optional elements.
|	Vertical bars separate a set of options.
...	A horizontal ellipsis denotes that the preceding element can optionally be repeated any number of times.
( ) , ;	Parentheses and other punctuation marks are required elements. Enter them as shown in syntax diagrams.

RELATED DOCUMENTATION

Refer to the following guides for more information:

Dharma SDK SQL Reference Manual	This manual describes syntax and semantics of SQL language statements and elements for Dharma SDK .
Dharma SDK User Guide	This manual describes the Software Development Kit (SDK).It describes implementing JDBC, ODBC and .NET access to proprietary data and considerations for creating a release kit to distribute the completed implementation.
Dharma SDK ISQL Reference Manual	This manual provides reference material for the ISQL interactive tool provided in the Dharma SDK environment. It also includes a tutorial describing how to use the ISQL utility.
Dharma SDK ODBC Driver Guide	This manual describes Dharma SDK support for ODBC (Open Database Connectivity) interface and how to configure the Dharma SDK ODBC Driver.

Dharma SDK JDBC Driver Guide	Describes Dharma SDK support for the JDBC interface and how to configure the Dharma SDK JDBC Driver.
Dharma SDK .NET Data Provider Guide	This guide gives an overview of the .NET Data Provider. It describes how to set up and use the .NET Data Provider to access a Dharma SDK database from .NET applications.

Chapter 1

Introduction

This chapter introduces the Dharma SDK .NET Data Provider, an implementation of the Data Provider interfaces of Microsoft for accessing the Dharma SDK environment from .NET applications.

This chapter contains the following topics:

1. Overview of Microsoft's .NET Technology
2. Overview of .NET Data Providers
3. Overview of the Dharma .NET Data Provider

1.1 OVERVIEW OF MICROSOFT'S .NET TECHNOLOGY

The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet. ActiveX Data Objects for the .NET Framework (ADO.NET) is a set of classes that expose data access services to the .NET programmer. ADO.NET provides a rich set of components for creating distributed, disconnected and data-sharing applications. ADO.NET is an integral part of the .NET Framework, providing relational data access to systems such as Dharma SDK.

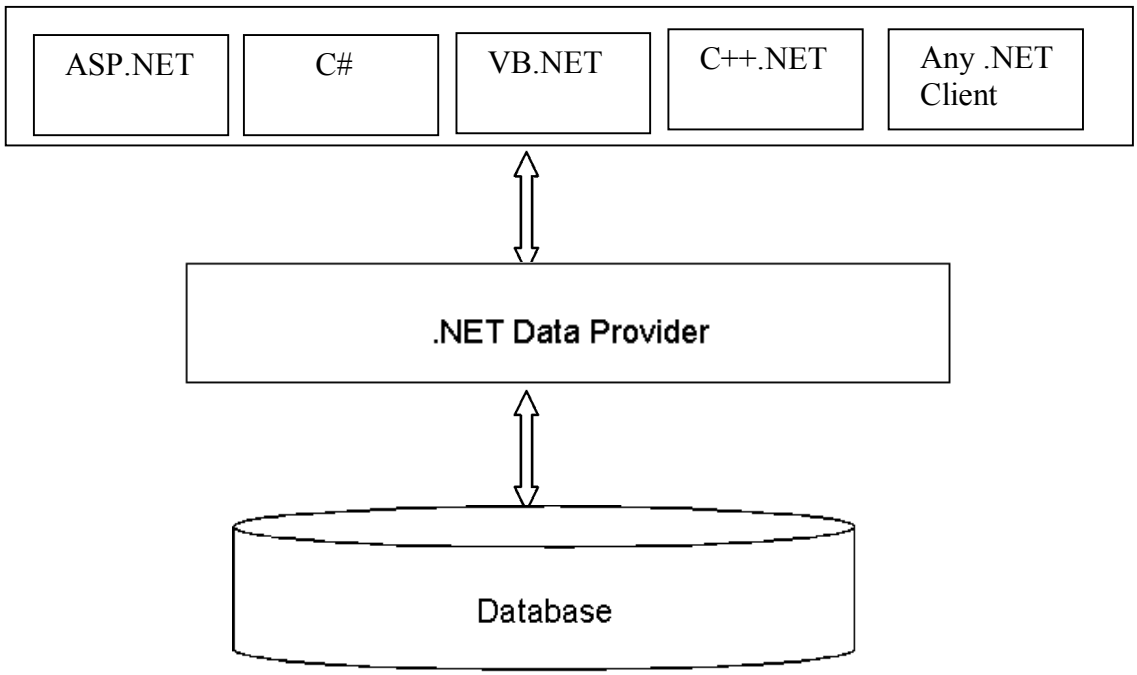
1.2 OVERVIEW OF .NET DATA PROVIDERS

A .NET Data Provider is a bridge used for connecting ADO.NET applications to a database, executing commands and retrieving results. The .NET Data Provider is designed to be lightweight, creating a minimal layer between the data source and your code, thus increasing performance without sacrificing functionality. A .NET Data Provider consists of a set of classes that implement interfaces specified in Microsoft's specification for .NET Data Providers. The Dharma SDK .NET Data Provider, which is introduced in the next section, is an implementation of a Data Provider for accessing the Dharma SDK environment from .NET applications.

1.3 OVERVIEW OF THE DHARMA SDK .NET DATA PROVIDER

The Dharma SDK .NET Data Provider is an implementation of the Data Provider interfaces of Microsoft for accessing the Dharma SDK environment from .NET applications. The Dharma SDK .NET Data Provider uses Dharma SDK native APIs to offer fast and reliable access to Dharma SDK data from any .NET applications. The Dharma SDK .NET Data Provider also uses and inherits classes and interfaces available in the Microsoft .NET Framework Class Library. The following figure illustrates the architecture of data access from a Dharma SDK Database using the Dharma SDK .NET Data Provider.

Figure 1-1: Dharma .NET Data Provider Architecture



Chapter 2

Installation and Configuration

This chapter describes the installation and configuration requirements for the Dharma SDK .NET Data Provider.

This chapter contains the following topics:

1. System Requirements
2. Assembly
3. Installing the Dharma SDK .NET Data Provider
4. File Location

2.1 SYSTEM REQUIREMENTS

The Dharma SDK .NET Data Provider requires the following:

- Windows 2000 or XP
- Access to Dharma SDK Server 9.0
- Microsoft .NET Framework version 1.1

2.2 ASSEMBLY

The name of the Dharma SDK .NET Data Provider assembly is Dharma.Data.Sql.dll. It provides the Dharma.Data.Sql namespace. The Dharma.Data.Sql namespace contains the Dharma SDK .NET Data Provider classes.

2.3 INSTALLING THE DHARMA SDK .NET DATA PROVIDER

To install the Dharma SDK .NET Data Provider, execute the SETUP.EXE program that exists in the root directory of the distribution kit of the Dharma SDK.NET Data Provider.

When you install the Dharma SDK .NET Data Provider, the installer program automatically installs and registers the provider with the Global Assembly Cache (GAC).

After successful installation you need to register the Dharma SDK .NET Data Provider components in the Toolbox of VisualStudio.NET by following these steps:

1. Open Visual Studio.NET and go to the Toolbox window.
2. Add a 'DharmaData' tab by right clicking on the General tab and selecting Add Tab
3. Right click on the 'DharmaData' tab. Select Customize Toolbox... from the displayed popup menu.

4. Go to the .NET Framework Components tab within the opened dialog box.
5. Click on the 'Browse' button on the .NET Framework Components tab and select the installed DLL (Dharma.Data.Sql.dll).
6. This will add DharmaCommand, DharmaConnection and DharmaDataAdapter in the .NET Framework Components tab and by default all of them will be selected.
7. Press the OK button.

Once these steps are completed, the components will be available from the toolbox.

See Also: The Dharma SDK User guide for installation instructions.

2.4 FILE LOCATION

The installer program installs Dharma.Data.Sql.dll in the <Dharma SDK Installation directory>\bin directory.

Chapter 3

Features

This chapter describes the features of the Dharma SDK .NET Data Provider.

The following topics are covered:

1. Data Provider classes
2. Connection String
3. Data type Mapping
4. Internationalization Support
5. Error Messages
6. Limitations

3.1 DATA PROVIDER CLASSES

The Dharma SDK .NET Data Provider classes in the Dharma.Data.Sql namespace are listed below.

Table 3-1: Dharma SDK .NET Data Provider Classes

Data Provider Class	Description
DharmaConnection	This class implements the IDbConnection interface. Represents an open connection to a data source.
DharmaCommand	This class implements the IDbCommand interface. Represents an SQL statement or stored procedure to execute against a data source.
DharmaDataReader	This class implements the IDataReader interface. Provides a way of reading a forward-only stream of data rows from a data source.
DharmaDataAdapter	This class implements the IDbDataAdapter interface. Represents a set of data commands and a connection to a data source that are used to fill the DataSet and update the data source.
DharmaParameter	This class implements the IDbDataParameter interface. Represents a parameter to a DharmaCommand.

Table 3-1: Dharma SDK .NET Data Provider Classes

DharmaParameterCollection	This class implements the IDataParameterCollection interface. Represents a collection of parameters relevant to DharmaCommand as well as their respective mappings to columns in a DataSet.
DharmaRowUpdatedEventArgs	Provides data for the RowUpdated event.
DharmaRowUpdatingEventArgs	Provides data for the RowUpdating event.
DharmaTransaction	This class implements the IDbTransaction interface. Represents an SQL transaction in the Dharma SDK Database.
DharmaError	Collects information relevant to a warning or error returned by the Data Provider.
DharmaErrorCollection	Collects all errors generated by the Dharma SDK .NET Data Provider
DharmaException	The exception that is generated when a warning or error is returned by the Dharma SDK .NET Data Provider.

3.2 CONNECTION STRING

This section describes the Name-Value pairs used in forming a connection string to connect to a Dharma SDK database using the Dharma SDK .NET Data Provider.

The connection string to be passed into the DharmaConnection Object is a set of semicolon separated Name-Value pairs.

The following table lists Name-Value pairs used in connection strings for the Dharma SDK .NET Data Provider. The mandatory values are UID, PWD and Database. If other values are not specified, default values will be assigned.

Table 3-2: Connection string

Name	Default Value	Description
UID	None	The user name for login
PWD	None	The password for the user.
Database	None	The name of the database.
Server	localhost	The name of machine on which the Dharma SDK Server resides.
Service	sqlnw	Indicates the port number used by the Dharma Server for listening.

3.3 DATA TYPE MAPPING

The following table lists the mapping between Dharma SDK data types and .NET Framework types. It also lists DbTypes and .NET Framework typed accessors.

Table 3-3: Mapping between Dharma SDK Data Types and .NET Framework Types

Dharma SDK Type	DbType	.NET Framework Type	.NET Framework Typed Accessor
Bigint	Int64	Int64	GetInt64()
Binary	Binary	Byte[ ]	GetBytes()
Bit	Boolean	Boolean	GetBoolean()
Character	AnsiStringFix-edLength	Byte/Byte[ ]	GetByte()/GetBytes()
Date	DateTime	DateTime	GetDateTime()
Double precision/ Float	Double	Double	GetDouble()
Integer	Int32	Int32	GetInt32()
Money	Decimal	Decimal	GetDecimal()
Nchar	StringFixedLength	String	GetString()
Nvarchar	String	String	GetString()
Numeric	Decimal	Decimal	GetDecimal()
Real	Single	Single	GetSingle()
Smallint	Int16	Int16	GetInt16()
Time	Time	String	GetString()
Timestamp	DateTime	DateTime	GetDateTime()
Tinyint	SByte	SByte	GetInt16()
Varchar	AnsiString	Byte/Byte[ ]	GetByte()/GetBytes()

Note:- NCHAR and NVARCHAR are supported only in the Dharma/SQL product suites. Information given in this document on Internationalization or Unicode support is not applicable for the SDK product. Please contact Dharma support for further details.

The following sections discuss Dbtype and .NET Framework data types and conversions supported by the Dharma SDK .NET Data Provider

3.3.1 Setting Values for Parameters

The following table gives the preferred .NET data type to be used for setting the parameter value for Unicode and ANSI character columns. Supported conversions are also listed.

Table 3-4: .Net data types to be used for setting Unicode and non-Unicode data

	.NET Data Types			
DbTypes		String	Char/Char[ ]	Byte/Byte[ ]
	AnsiString AnsiStringFixedLength	Supports conversion from Unicode String to ANSI characters. UTF-8 is the encoding scheme used	Supports conversion from Unicode Char/Char[ ] to ANSI characters. UTF-8 is the encoding used	Preferred data type to set non-Unicode data. No conversion is done.
	String StringFixedLength	Preferred data type to set Unicode data. No conversion is done.	Preferred data type to set Unicode data. No conversion is done.	Supports conversion from Byte/Byte[ ] to Unicode data. The Byte/Byte[ ] would be treated as UTF-8 encoded.

3.3.2 Getting Values from DataReader

The following table gives the preferred typed accessor methods to be used for getting data from DharmaDataReader object for Unicode and ANSI character columns. Supported conversions are also mentioned.

Table 3-5: .Net accessor methods to be used for getting Unicode and non-Unicode data

	.NET Typed Accessor Methods			
DbTypes		GetString()	GetChar() GetChars()	GetByte() GetBytes()
	AnsiString AnsiString FixedLength	Supports conversion from non-Unicode data to Unicode String. Byte/Byte[ ]. The column data would be treated as UTF-8 encoded.	Supports conversion from non-Unicode data to Unicode Char/Char[ ]. The column data would be treated as UTF-8 encoded.	Preferred method for getting non-Unicode data. No conversion is done.
	String String FixedLength	Preferred method for getting Unicode data. No conversion is done.	Preferred method for getting Unicode data. No conversion is done.	Supports conversion from Unicode data to Byte/Byte[ ]. UTF-8 is the encoding used.

3.4 ERROR MESSAGES

Appendix A lists the error messages returned by the Dharma SDK .NET Data Provider.

3.5 LIMITATIONS

The following table lists the limitations of the Dharma SDK .NET Data Provider

Table 3-6: Limitations of Dharma SDK .NET Data Provider

Feature not Supported	Class	Remarks
Long data type	Not Applicable.	Not supported in this release.
GetData()	DharmaDataReader	Not supported.
GetSchemaTable()	DharmaDataReader	This method will not give key information in this release.
ChangeDatabase	DharmaConnection DharmaCommand	Not supported.
CommandTimeout	DharmaConnection	Not supported.
GetGuid()	DharmaDataReader	Not supported.
Cancel()	DharmaCommand	Cancelling a command will not have any effect.
Parameter Names	DharmaParameter	Not supported.
Stored Procedure with return values	DhParamter	Not supported in this release.
Object , Empty, DBNull and UnIt64 .NET Framework data types	Not Applicable	Not supported in this release.
UInt64 and VarNumeric DbTypes	Not Applicable	Not supported in this release.

Chapter 4

Developing Applications

This chapter describes various steps involved in developing applications using the Dharma SDK .NET Data Provider.

This chapter contains the following topics:

1. Building .NET Applications
2. Connecting to a Dharma SDK Database
3. Executing an SQL Command
4. Retrieving Data
5. Using Stored Procedures
6. Performing Transactions
7. Populating a DataSet
8. Using Parameters
9. Using Stored Procedures with Parameters
10. Exception Handling

4.1 BUILDING .NET APPLICATIONS

The following subsections describe how to build a .NET application.

4.1.1 Importing the DLLs

Import the following DLLs to the .NET application.

1. System.Data.dll
2. Dharma.Data.Sql.dll

For C++ .NET applications, import mscorlib.dll also. The following C++ code example demonstrates how to import required DLLs into the application.

```
#using <mscorlib.dll>
#using <System.Data.dll>
#using <Dharma.Data.Sql.dll>
```

Refer to the .NET Framework SDK documentation of Microsoft to find out how to import the DLLs in other .NET languages.

4.1.2 Using the Namespaces

Include the following namespaces in the .NET application.

1. System
2. System.Data
3. Dharma.Data.Sql

The following C++ code example demonstrates how to include the required namespaces in the application.

```
using namespace System;
using namespace System::Data;
using namespace Dharma::Data::Sql;
```

Refer to the .NET Framework SDK documentation of Microsoft to find out how to include the namespaces in other .NET languages.

4.1.3 Compiling the Application

To compile a C++ application, say, sample.cxx, from the command line, use the following command. Ensure that Dharma.Data.Sql.dll is present in %libpath%.

```
cl /clr sample.cxx
```

The compiler generates an executable called sample.exe. Refer to the .NET Framework SDK documentation to find out how to compile the application in other .NET languages.

4.2 CONNECTING TO A DHARMA SDK DATABASE

The Dharma SDK .NET Data Provider provides connectivity to Dharma SDK Databases using the DharmaConnection object. The Dharma SDK .NET Data Provider supports a connection string format that is similar to an ODBC connection string format. For valid string format names and values, see Chapter 3, Section Connection String.

The following C++ code example demonstrates how to create and open a connection to a Dharma SDK Database.

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=dharma;pwd=asterix;database=sampledb");
// Open the connection.
dhCon->Open();
```

Closing the DharmaConnection

You must always close the DharmaConnection object when you are finished using it. This can be done using the Close() method of the DharmaConnection object.

4.3 EXECUTING AN SQL COMMAND

After establishing a connection to the data source, you can execute commands and return results from the data source using a DharmaCommand object. You can create a command using the DharmaCommand constructor, which takes optional arguments of an SQL statement to execute at the data source, a DharmaConnection object and a DharmaTransaction object. You can also create a command for a particular DharmaConnection using the CreateCommand() method of the DharmaConnection object. The SQL statement of the Command object can be queried and modified using the CommandText property.

The DharmaCommand object exposes several Execute methods you can use to perform the intended action. When returning results as a stream of data, use ExecuteReader() to return a DharmaDataReader object. Use ExecuteScalar() to return a singleton value. Use ExecuteNonQuery() to execute commands that do not return rows.

When using the Command object with a stored procedure, you may set the CommandType property of the Command object to StoredProcedure. With a CommandType of StoredProcedure, you may use the Parameters property of the Command to access input and output parameters and return values. The Parameters property can be accessed regardless of the Execute method called. However, when calling ExecuteReader(), return values and output parameters will not be accessible until the DataReader is closed.

The following C++ code example demonstrates how to format a DharmaCommand object to return a list of regions from the sampledb database.

```
// Create a new command object.
IDbCommand *dhCmd = new DharmaCommand
      ("SELECT * FROM region", dhCon);
```

4.4 RETRIEVING DATA

You can use the DharmaDataReader to retrieve a read-only, forward-only stream of data from the database.

After creating an instance of the Command object, you create a DharmaDataReader by calling DharmaCommand.ExecuteReader() to retrieve rows from the data source, as shown in the following example.

```
// Execute the command and create a DataReader object.
IDataReader *dhRdr = dhCmd->ExecuteReader();
```

You use the Read() method of the DharmaDataReader object to obtain a row from the results of the query. You can access each column of the returned row by passing the name or ordinal reference of the column to the DharmaDataReader. However, for best performance, the DharmaDataReader provides a series of methods that allow you to access column values in their native data types (GetDateTime, GetDouble, GetInt32 and so on). Using the typed accessor methods when the underlying data type is known will reduce the number of type conversions required when retrieving the column value.

The following C++ code example iterates through a DharmaDataReader object and returns the first two columns from each row.

```
// Read records and print on the console.
while ( dhRdr->Read())
{
         Console::WriteLine ("\t{0}\t{1}",
         (dhRdr->GetInt32(0)).ToString(),dhRdr->GetString (1));
}
```

Closing the DharmaDataReader

You should always call the Close() method when you have finished using the DharmaDataReader object.

If your DharmaCommand contains output parameters or return values, they will not be available until the DharmaDataReader is closed.

Note that while a DharmaDataReader is open, the DharmaConnection is in use exclusively by that DharmaDataReader. You will not be able to execute any commands for the DharmaConnection, including creating another DharmaDataReader, until the original DharmaDataReader is closed.

4.5 USING STORED PROCEDURES

Stored procedures offer many advantages in data-driven applications. Using stored procedures, database operations can be encapsulated in a single command improving the performance

To execute a stored procedure, set the CommandType of the DharmaCommand object to StoredProcedure, as in the following example.

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=dharma;pwd=asterix;database=sampledb");
// Create a new command object.
IDbCommand *dhCmd = new DharmaCommand ("get_regions()”,dhCon);
// Set the command type.
dhCmd->CommandType = CommandType::StoredProcedure;
// Open the connection.
dhCon->Open();
// Execute the command and create a DataReader object.
IDataReader *dhRdr = dhCmd->ExecuteReader();
// Read the resultset returned by the procedure execution
// and print on the console.
while ( dhRdr->Read())
{
      Console::WriteLine (dhRdr->GetString (0));
}
```

```
// Close the reader.
dhRdr->Close();
// Close the connection.
dhCon->Close();
```

4.6 PERFORMING TRANSACTIONS

Transactions are a group of database operations combined into a logical unit of work and are used to control and maintain the consistency and integrity of the database despite errors that might occur in the database system.

In ADO.NET, you control transactions using the DharmaConnection and DharmaTransaction objects.

To perform a transaction

1. Call the BeginTransaction method of the DharmaConnection object to mark the start of the transaction. BeginTransaction returns a reference to the DharmaTransaction. Retain this reference so that you can assign it to DharmaCommands that are used in the transaction.
2. Assign the DharmaTransaction object to the Transaction property of the DharmaCommands to be executed. If a DharmaCommand is executed on a DharmaConnection with an active Transaction and the DharmaTransaction object has not been assigned to the Transaction property of the DharmaCommand, an exception will be thrown.
3. Execute the required commands.
4. Call the Commit() method of the DharmaTransaction object to complete the transaction, or call the Rollback() method to cancel the transaction.

The following C++ code example demonstrates the usage of transactions.

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=dharma;pwd=asterix;database=sampledb");
// Open the connection.
dhCon->Open();
// Begin a Transaction.
IDbTransaction * dhTxn = dhCon->BeginTransaction();
// Create a command.
IDbCommand *dhCmd = dhCon->CreateCommand();
// Enlist the command in the current transaction.
dhCmd->Transaction = dhTxn;
// Set the command text and execute the command.
try
{
      dhCmd-> CommandText = "INSERT INTO Region (RegionID,
      RegionDescription) VALUES (100, 'Chicago')";
```

```
      dhCmd->ExecuteNonQuery();
      dhCmd-> CommandText = "INSERT INTO Region (RegionID,
      RegionDescription) VALUES (101, 'New york')";
      dhCmd->ExecuteNonQuery();


      // Commit the changes.
      dhTxn->Commit();
      Console::WriteLine
         ("Both records have been written to the database.");
}
catch(Exception * e)
{
      dhTxn->Rollback();
      Console::WriteLine(e->ToString());
      Console::WriteLine
         ("Neither record was written to the database.");
}
```

4.7 POPULATING A DATASET

The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model independent of the data source. Because the DataSet is independent of the data source, the DataSet can include data local to the application, as well as data from multiple data sources. Interaction with existing data sources is controlled through the DharmaDataAdapter.

The DharmaDataAdapter is used to retrieve data from a Dharma SDK Database and populate tables within a DataSet. The DharmaDataAdapter also resolves changes made to the DataSet back to the data source. The DharmaDataAdapter uses the DharmaConnection object to connect to a Dharma SDK Database and DharmaCommand objects to retrieve data from and resolve changes to the data source.

The following code example creates an instance of a DharmaDataAdapter that uses a DharmaConnection object to connect to the Dharma SDK sampledb database and populates a DataTable in a DataSet with the list of regions. The SQL statement and DharmaConnection arguments passed to the DataAdapter constructor are used to create the SelectCommand property of the DataAdapter.

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=dharma;pwd=asterix;database=sampledb");
// Open the connection.
dhCon->Open();
// Create a new command object.
IDbCommand *selCmd = new DharmaCommand
      (“SELECT * FROM region", dhCon);
// Create an Adapter and set the command.
```

```
IDbDataAdapter * dhDa = new DharmaDataAdapter();
dhDa->SelectCommand = selCmd;
// Create a data set for regions and fill it.
DataSet * regionDs = new DataSet();
dhDa->Fill(regionDs);
```

4.8 USING PARAMETERS

The DharmaParameter object is used to execute parameterized queries.

The following example illustrates the usage of parameters.

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=dharma;pwd=asterix;database=sampledb");
// Open the connection.
dhCon->Open();

// Create a new command object.
IDbCommand *dhCmd = new DharmaCommand
         ("SELECT * FROM region WHERE regionid = ?", dhCon);
// Create a parameter and add to the command.
IDbDataParameter * dhParam = new DharmaParameter();
dhParam->DbType = DbType::Int32;
dhCmd->Parameters->Add(dhParam);

// Prepare the statement.
dhCmd->Prepare();

// Set the value for region id.
dhParam->Value  = __box (10);

// Execute the command and create a DataReader object.
IDataReader *dhRdr = dhCmd->ExecuteReader();
// Read records and print on the console.
while (dhRdr->Read())
{
    Console::WriteLine (dhRdr->GetString (1));
}
```

4.9 USING STORED PROCEDURES WITH PARAMETERS

While a stored procedure can be called by simply passing the stored procedure name followed by parameter arguments as an SQL statement, using the DharmaParameter-

Collection in the DharmaCommand object enables you to more explicitly define stored procedure parameters as well as to access output parameters.

The following example illustrates the usage of a stored procedure with parameters.

```
// Create a new connection object.
IDbConnection * dhCon = new DharmaConnection
("uid=dharma;pwd=asterix;database=sampledb");
// Create a new command object.
IDbCommand *dhCmd = new DharmaCommand ("get_region(?)", dhCon);
// Set the command type.
dhCmd->CommandType = CommandType::StoredProcedure;
// Open the connection.
dhCon->Open();
// Construct a parameter and add to the command.
IDbDataParameter * dhParam = new DharmaParameter();
dhParam->DbType = DbType::Int32;
dhCmd->Parameters->Add(dhParam);

// Prepare the statement.
dhCmd->Prepare();

// Set the value for region id.
dhParam->Value  = __box (10);
// Execute the command and create a DataReader object.
IDataReader * dhRdr = dhCmd->ExecuteReader();

// Read records and print on the console.
while (dhRdr->Read())
{
    Console::WriteLine (dhRdr->GetString (0));
}
```

Note:- Stored Procedures are supported only in Dharma/SQL product suites. Please contact Dharma support for further details on this .

4.10 EXCEPTION HANDLING

The errors generated in the Data Provider are thrown as exceptions. Refer to Chapter 5 for more details. The client application will have to catch the exceptions and then process the exceptions.

The following example illustrates the usage of exceptions.

```
try
{
   ...
```

```
   // Do some operations.
   ...
}
catch(ArgumentNullException * ex)
{
   Console::WriteLine(ex->ToString());
}
catch(DharmaException * ex)
{
   for(int i=0;i < ex->Errors->Count;i++)
   {
         Console::WriteLine(ex->Errors->get_Item(i)->Number);
         Console::WriteLine(ex->Errors->get_Item(i)->State);
         Console::WriteLine(ex->Errors->get_Item(i)->Message);
   }

}
catch(Exception * ex)
{
    Console::WriteLine(ex->ToString());
}
```

Chapter 5

Data Provider Class Reference

This chapter provides reference material on the Dharma SDK .NET Data Provider classes to be used for exception and error handling. The Dharma SDK .NET Data Provider has three classes for exception and error handling - DharmaException, DharmaErrorCollection and DharmaError.

For a detailed description of methods in other classes, refer to Microsoft documentation on .NET Data Provider interfaces.

This chapter contains the following topics:

1. DharmaException
2. DharmaErrorCollection
3. DharmaError

5.1 DHARMAEXCEPTION

A DharmaException is thrown when the Dharma SDK Data Provider returns an error. The following table lists the methods and properties in this class.

Table 5-1: DharmaException members

Name	Method/Property	Description
DharmaErrorCollection * get_Errors()	Property	Gets a collection of type DharmaErrorCollection of one or more DharmaError objects that give detailed information about exceptions generated
String * get_State()	Property	Gets the SQLSTATE associated with the first error in the DharmaErrorCollection
int get_Number()	Property	Gets the error number associated with the first error in the DharmaErrorCollection
String * get_Message()	Property	Gets the error message associated with the first error in the DharmaErrorCollection
String * get_AllMessages()	Property	Gets the error messages of all the errors in the DharmaErrorCollection. In the current implementation, the DharmaErrorCollection object contains only one DharmaError object. Hence, this method returns only one error message.

5.2 DHARMAERRORCOLLECTION

DharmaErrorCollection is a collection of DharmaError.

The following table lists the methods and properties in this class.

Table 5-2: DharmaErrorCollection Members

Name	Method/Property	Description
int get_Count()	Property	Returns the number of DharmaErrors in the collection.
DharmaError* get_Item(int index)	Property	Returns the DharmaError at a specified index.
bool get_IsSynchronized()	Property	Gets a boolean value indicating whether access to the DharmaError Collection is synchronized or not. Returns false.
Object * get_SyncRoot()	Property	Gets the current object that can be used to synchronize access to the DharmaErrorCollection.

5.3 DHARMAERROR

DharmaError contains information relevant to an error generated by the provider. One or more DharmaError objects are managed by the DharmaErrorCollection class, which in turn is created by the DharmaException class.

The following table lists the methods and properties in this class.

Table 5-3: DharmaError Members

Name	Method/Property	Description
String * get_Message()	Property	Returns error message.
int get_Number()	Property	Returns error number.
String* get_State()	Property	Returns SQLSTATE.

Appendix A

Error Messages

This appendix lists the error messages generated by the Dharma SDK .NET Data Provider.

Table A-1: Error Codes and Messages

Error Code	SQLSTATE	Error Message
101	S1000	Invalid connection state
102	HYC00	Feature not implemented
103	25S04	Isolation level not supported
104	R9001	Prepared statement expects parameter value, which is not supplied
105	HY090	Invalid or null connection
106	25S05	Invalid or null transaction
107	42000	Syntax error
108	HY090	Invalid string or buffer length
109	R2001	Command does not have any associated transaction
110	R3001	Transaction does not belong to the set connection
111	08003	Connection not opened
112	R2001	Command properties missing or not initialized
113	R2001	Connection property not set
114	R2001	CommandText property not set or initialized
115	R1001	Server name exceeds 15 characters
116	R2002	Rolling back not allowed in autocommit on state
117	R5001	Invalid data type
118	R6001	Invalid size for a parameter
119	R2002	Resetting of connection not allowed after command is prepared
120	R2003	Prepared command text no longer valid

Table A-1: Error Codes and Messages

121	R9001	No memory for allocating SQLDA
122	R4006	GetDateTime() can not be used for fetching time fields; use GetString()
123	R4007	DataReader is not closed

Appendix B

Sample Programs

This appendix provides complete code listings of two sample C++ programs demonstrating the usage of the Dharma SDK .NET Data Provider.

B.1 RETRIEVING DATA

The following sample program executes a statement and fetches the records from the database.

```
/*
 *      Copyright (C) Dharma Systems Inc.        1988-2004.
 *      Copyright (C) Dharma Systems (P) Ltd.    1988-2004.
 *
 *      This Module contains Proprietary Information of
 *      Dharma Systems Inc. and Dharma Systems (P) Ltd.
 *      and should be treated as Confidential.
 */

/*
 * Purpose : To show execution of a query and fetching rows from its resultset.
 */

#using <mscorlib.dll>
#using <System.dll>
#using <System.Data.dll>
#using <System.Data.dll>
#using <Dharma.Data.Sql.dll>

using namespace System;
using namespace System::IO;
using namespace System::Data;
using namespace Dharma::Data::Sql;

int main()
{

    IDbConnection * dhCon = 0;
    IDbCommand * dhCmd = 0;
    IDataReader * dhRdr = 0;

    try
    {
       // Create a new connection object.
       dhCon = new DharmaConnection
          ("uid=dharma;pwd=asterix;database=sampledb");

       // Open the connection.
       dhCon->Open();
```

```
       // Create a new command object.
       dhCmd = new DharmaCommand ("SELECT * FROM region", dhCon);

       // Execute the command and create a DataReader object.
       dhRdr = dhCmd->ExecuteReader();

       // Read records and print on the console.
       while (dhRdr->Read())
       {
           Console::WriteLine ("\t{0} \t{1}",(dhRdr->GetInt32(0)).ToString(),
                               dhRdr->GetString(1));
       }
   }
   catch (DharmaException *e)
   {
       for(int i=0;i < e->Errors->Count;i++)
       {
           Console::WriteLine(e->Errors->Item[i]->Message);
       }
   }
   catch (Exception *e)
   {
       Console::WriteLine(e->ToString());
   }
    __finally
   {
       // Close the reader.
       if (dhRdr)
           dhRdr->Close();

       // Close the connection.
       if (dhCon)
           dhCon->Close();
   }

}
```

B.2 USING PARAMETERS

The following sample program demonstrates the usage of parameters.

```
/*
 *      Copyright (C) Dharma Systems Inc.        1988-2004.
 *      Copyright (C) Dharma Systems (P) Ltd.    1988-2004.
 *
 *      This Module contains Proprietary Information of
 *      Dharma Systems Inc. and Dharma Systems (P) Ltd.
 *      and should be treated as Confidential.
 */

/*
 *      Purpose : To demonstrate the usage of parameters.
 */

#using <mscorlib.dll>
#using <System.dll>
#using <System.Data.dll>
#using <System.Data.dll>
#using <Dharma.Data.Sql.dll>
```

```
using namespace System;
using namespace System::IO;
using namespace System::Data;
using namespace Dharma::Data::Sql;
int main()
{

    IDbConnection * dhCon = 0;
    IDbCommand * dhCmd = 0;
    IDataReader * dhRdr = 0;

    try
    {
       // Create a new connection object.
       dhCon = new DharmaConnection ("uid=dharma;pwd=asterix;database=sampledb");

       // Open the connection.
       dhCon->Open();

       // Create a new command object.
       dhCmd = new DharmaCommand ("get_region(?)", dhCon);
       dhCmd->CommandType = CommandType::StoredProcedure;

       // Construct a parameter and add to the command.
       IDbDataParameter * dhParam = new DharmaParameter();
       dhParam->DbType = DbType::Int32;
       dhCmd->Parameters->Add(dhParam);

       // Prepare the statement.
       dhCmd->Prepare();

       // Set the value for region id.
       dhParam->Value  = __box (10);

       // Execute the command and create a DataReader object.
       dhRdr = dhCmd->ExecuteReader();

       // Read records and print on the console.
       while (dhRdr->Read())
       {
           Console::WriteLine (dhRdr->GetString (0));
       }
   }
   catch (DharmaException *e)
   {
       for(int i=0;i < e->Errors->Count;i++)
       {
          Console::WriteLine(e->Errors->Item[i]->Message);
       }
   }
   catch (Exception *e)
   {
       Console::WriteLine(e->ToString());
   }
    __finally
   {
       // Close the reader.
       if (dhRdr)
```

```
            dhRdr->Close();

        // Close the connection.
        if (dhCon)
         dhCon->Close();
    }
}
```

Appendix C

Glossary

C.1 TERMS

.NET Data Provider
A .NET Data Provider is a bridge used for connecting ADO.NET applications to a database, executing commands and retrieving results.

.NET Framework
The .NET Framework is a new computing platform that simplifies application development in the highly distributed environment of the Internet.

ANSI Character Set
A character set containing 256 characters, numbered 0 to 255. Values 0 to 127 are the same as in the ASCII character set. Values 128 to 255 contain European characters and special characters.

ASCII Character Set
ASCII is the acronym for American Standard Code for Information Interchange, a 7-bit code that is the U.S. national variant of ISO/IEC 646. Formally, the U.S. standard ANSI X3.4. ASCII is a code for representing English characters as numbers, with each letter assigned a number from 0 to 127.

Assembly
A collection of functionality built, versioned, and deployed as a single implementation unit (one or multiple files). An assembly is the primary building block of a .NET Framework application. All managed types and resources are marked either as accessible only within their implementation unit or as exported for use by code outside that unit. In the common language runtime, the assembly establishes the name scope for resolving requests and the visibility boundaries are enforced. The common language runtime can determine and locate the assembly for any running object because every type is loaded in the context of an assembly.

Assembly Cache
A machine-wide code cache used for side-by-side storage of assemblies. There are two parts to the cache: the global assembly cache contains assemblies that are explicitly installed to be shared among many applications on the computer; the download cache stores code downloaded from Internet or intranet sites, isolated to the application that triggered the download so that code downloaded on behalf of one application or page does not impact other applications. See also: global assembly cache.

Dharma SDK .NET Data Provider
The Dharma SDK .NET Data Provider is an implementation of the Data Provider interfaces of Microsoft for accessing the Dharma SDK environment from .NET applications.

Global Assembly Cache (GAC)
A machine-wide code cache that stores assemblies specifically installed to be shared by many applications on the computer. Applications deployed in the global assembly cache must have a strong name.

JDBC Driver
Database-specific software that receives calls from the JDBC driver manager, translates them into a form that the database can process, and returns data to the application.

ODBC Driver
Vendor-supplied software that processes ODBC function calls for a specific data source. The driver connects to the data source, translates the standard SQL statements into syntax the data source can process and returns data to the application. Dharma SDK includes an ODBC driver for its internal database system. In addition, there are ODBC drivers for every major database system.

SQL Engine
The core component of the Dharma SDK database. The SQL engine receives requests from applications, processes them and returns results. The Dharma SQL engine calls the storage interfaces to convey requests to an underlying storage system.

SQLSTATE
A 5-character status parameter that indicates the condition status returned by the most recent SQL statement. SQLSTATE is specified by the SQL-92 standard as a replacement for the SQLCODE status parameter (which was part of SQL-89). SQLSTATE defines many more specific error conditions than SQLCODE, which allows applications to implement more portable error handling.

Stored Procedure
A snippet of Java source code embedded in an SQL CREATE PROCEDURE statement. The source code can use all standard Java features as well as use Dharma SDK-supplied Java classes for processing any number of SQL statements.

Transaction
A group of operations whose changes can be made permanent or undone only as a unit.

Unicode
Unicode is a universal encoded character set that enables information from any language to be stored using a single character set.

UTF-8 Encoding
UTF-8 is a multibyte encoding in which each character can be encoded in as little as one byte and as many as four bytes.

Index