

# Contents

[System.Data.SqlClient](#)

[ApplicationIntent](#)

[OnChangeEventHandler](#)

[PoolBlockingPeriod](#)

[SortOrder](#)

[SqlAuthenticationInitializer](#)

[Initialize](#)

[SqlAuthenticationInitializer](#)

[SqlAuthenticationMethod](#)

[SqlAuthenticationParameters](#)

[AuthenticationMethod](#)

[Authority](#)

[ConnectionId](#)

[DatabaseName](#)

[Password](#)

[Resource](#)

[ServerName](#)

[SqlAuthenticationParameters](#)

[UserId](#)

[SqlAuthenticationProvider](#)

[AcquireTokenAsync](#)

[BeforeLoad](#)

[BeforeUnload](#)

[GetProvider](#)

[IsSupported](#)

[SetProvider](#)

[SqlAuthenticationProvider](#)

[SqlAuthenticationToken](#)

[AccessToken](#)

ExpiresOn  
SqlAuthenticationToken  
SqlBulkCopy  
BatchSize  
BulkCopyTimeout  
Close  
ColumnMappings  
DestinationTableName  
EnableStreaming  
IDisposable.Dispose  
NotifyAfter  
SqlBulkCopy  
SqlRowsCopied  
WriteToServer  
WriteToServerAsync  
SqlBulkCopyColumnMapping  
DestinationColumn  
DestinationOrdinal  
SourceColumn  
SourceOrdinal  
SqlBulkCopyColumnMapping  
SqlBulkCopyColumnMappingCollection  
Add  
Clear  
Contains  
CopyTo  
Count  
GetEnumerator  
ICollection.CopyTo  
ICollection.IsSynchronized  
ICollection.SyncRoot  
IList.Add

IList.Contains  
IList.IndexOf  
IList.Insert  
IList.IsFixedSize  
IList.IsReadOnly  
IList.Item[Int32]  
IList.Remove  
IndexOf  
Insert  
Item[Int32]  
Remove  
RemoveAt  
SqlBulkCopyOptions  
SqlClientFactory  
    CanCreateDataSourceEnumerator  
    .CreateCommand  
    .CreateCommandBuilder  
    .CreateConnection  
    .CreateConnectionStringBuilder  
    .CreateCommandAdapter  
    .CreateCommandSource  
    .CreateCommandParameter  
    .CreateCommandPermission  
    Instance  
    IServiceProvider.GetService  
SqlClientLogger  
    IsLoggingEnabled  
    LogAssert  
    .LogError  
    LogInfo  
    SqlClientLogger  
SqlClientMetaDataCollectionNames  
Columns

Databases

ForeignKeys

IndexColumns

Indexes

Parameters

ProcedureColumns

Procedures

Tables

UserDefinedTypes

Users

ViewColumns

Views

SqlClientPermission

Add

Copy

SqlClientPermission

SqlClientPermissionAttribute

CreatePermission

SqlClientPermissionAttribute

SqlColumnEncryptionCertificateStoreProvider

DecryptColumnEncryptionKey

EncryptColumnEncryptionKey

ProviderName

SignColumnMasterKeyMetadata

SqlColumnEncryptionCertificateStoreProvider

VerifyColumnMasterKeyMetadata

SqlColumnEncryptionCngProvider

DecryptColumnEncryptionKey

EncryptColumnEncryptionKey

ProviderName

SignColumnMasterKeyMetadata

SqlColumnEncryptionCngProvider

[VerifyColumnMasterKeyMetadata](#)

[SqlColumnEncryptionCspProvider](#)

[DecryptColumnEncryptionKey](#)

[EncryptColumnEncryptionKey](#)

[ProviderName](#)

[SignColumnMasterKeyMetadata](#)

[SqlColumnEncryptionCspProvider](#)

[VerifyColumnMasterKeyMetadata](#)

[SqlColumnEncryptionEnclaveProvider](#)

[CreateEnclaveSession](#)

[GetAttestationParameters](#)

[GetEnclaveSession](#)

[InvalidateEnclaveSession](#)

[SqlColumnEncryptionEnclaveProvider](#)

[SqlColumnEncryptionKeyStoreProvider](#)

[DecryptColumnEncryptionKey](#)

[EncryptColumnEncryptionKey](#)

[SignColumnMasterKeyMetadata](#)

[SqlColumnEncryptionKeyStoreProvider](#)

[VerifyColumnMasterKeyMetadata](#)

[SqlCommand](#)

[BeginExecuteNonQuery](#)

[BeginExecuteReader](#)

[BeginExecuteXmlReader](#)

[Cancel](#)

[Clone](#)

[ColumnEncryptionSetting](#)

[CommandText](#)

[CommandTimeout](#)

[CommandType](#)

[Connection](#)

[CreateParameter](#)

DesignTimeVisible  
EndExecuteNonQuery  
EndExecuteReader  
EndExecuteXmlReader  
ExecuteNonQuery  
ExecuteNonQueryAsync  
ExecuteReader  
ExecuteReaderAsync  
ExecuteScalar  
ExecuteScalarAsync  
ExecuteXmlReader  
ExecuteXmlReaderAsync  
ICloneable.Clone  
 IDbCommand.CreateParameter  
 IDbCommand.ExecuteReader  
Notification  
NotificationAutoEnlist  
Parameters  
Prepare  
ResetCommandTimeout  
SqlCommand  
StatementCompleted  
Transaction  
UpdatedRowSource  
SqlCommandBuilder  
CatalogLocation  
CatalogSeparator  
DataAdapter  
DeriveParameters  
GetDeleteCommand  
GetInsertCommand  
GetUpdateCommand

Quot\_identifier  
Quote\_Prefix  
Quote\_Suffix  
RefreshSchema  
SchemaSeparator  
SqlCommandBuilder  
Unquot\_identifier  
SqlCommandColumnEncryptionSetting  
SqlConnection  
AccessToken  
BeginTransaction  
ChangeDatabase  
ChangePassword  
ClearAllPools  
ClearPool  
ClientConnectionId  
Close  
ColumnEncryptionKeyCacheTtl  
ColumnEncryptionQueryMetadataCacheEnabled  
ColumnEncryptionTrustedMasterKeyPaths  
ConnectionString  
ConnectionTimeout  
CreateCommand  
Credential  
Credentials  
Database  
DataSource  
EnlistDistributedTransaction  
EnlistTransaction  
FireInfoMessageEventOnUserErrors  
GetSchema  
ICloneable.Clone

[IDbConnection.BeginTransaction](#)  
[IDbConnection.CreateCommand](#)  
[InfoMessage](#)  
[Open](#)  
[OpenAsync](#)  
[PacketSize](#)  
[RegisterColumnEncryptionKeyStoreProviders](#)  
[ResetStatistics](#)  
[RetrieveStatistics](#)  
[ServerVersion](#)  
[SqlConnection](#)  
[State](#)  
[StateChange](#)  
[StatisticsEnabled](#)  
[WorkstationId](#)  
[SqlConnectionColumnEncryptionSetting](#)  
[ConnectionStringBuilder](#)  
[ApplicationIntent](#)  
[ApplicationName](#)  
[AsynchronousProcessing](#)  
[AttachDBFilename](#)  
[Authentication](#)  
[Clear](#)  
[ColumnEncryptionSetting](#)  
[ConnectionReset](#)  
[ConnectRetryCount](#)  
[ConnectRetryInterval](#)  
[ConnectTimeout](#)  
[ContainsKey](#)  
[ContextConnection](#)  
[CurrentLanguage](#)  
[DataSource](#)

EnclaveAttestationUrl

Encrypt

Enlist

FailoverPartner

InitialCatalog

IntegratedSecurity

IsFixedSize

Item[String]

Keys

LoadBalanceTimeout

MaxPoolSize

MinPoolSize

MultipleActiveResultSets

MultiSubnetFailover

NetworkLibrary

PacketSize

Password

PersistSecurityInfo

PoolBlockingPeriod

Pooling

Remove

Replication

ShouldSerialize

SqlConnectionStringBuilder

TransactionBinding

TransparentNetworkIPResolution

TrustServerCertificate

TryGetValue

TypeSystemVersion

UserID

UserInstance

Values

WorkstationID  
SqlCredential  
Password  
SqlCredential  
UserId  
SqlDataAdapter  
DeleteCommand  
ICloneable.Clone  
 IDbDataAdapter.DeleteCommand  
 IDbDataAdapter.InsertCommand  
 IDbDataAdapter.SelectCommand  
 IDbDataAdapter.UpdateCommand  
InsertCommand  
RowUpdated  
RowUpdating  
SelectCommand  
SqlDataAdapter  
UpdateBatchSize  
UpdateCommand  
SqlDataReader  
Close  
Connection  
Depth  
FieldCount  
GetBoolean  
GetByte  
GetBytes  
GetChar  
GetChars  
GetColumnSchema  
GetData  
GetDataTypeName

[GetDateTime](#)  
[GetDateTimeOffset](#)  
[GetDecimal](#)  
[GetDouble](#)  
[GetEnumerator](#)  
[GetFieldType](#)  
[GetFieldValue](#)  
[GetFieldValueAsync](#)  
[GetFloat](#)  
[GetGuid](#)  
[GetInt16](#)  
[GetInt32](#)  
[GetInt64](#)  
[GetName](#)  
[GetOrdinal](#)  
[GetProviderSpecificFieldType](#)  
[GetProviderSpecificValue](#)  
[GetProviderSpecificValues](#)  
[GetSchemaTable](#)  
[GetSqlBinary](#)  
[GetSqlBoolean](#)  
[GetSqlByte](#)  
[GetSqlBytes](#)  
[GetSqlChars](#)  
[GetSqlDateTime](#)  
[GetSqlDecimal](#)  
[GetSqlDouble](#)  
[GetSqlGuid](#)  
[GetSqlInt16](#)  
[GetSqlInt32](#)  
[GetSqlInt64](#)  
[GetSqlMoney](#)

GetSqlSingle  
GetSqlString  
GetSqlValue  
GetSqlValues  
GetSqlXml  
GetStream  
GetString  
GetTextReader  
GetTimeSpan  
GetValue  
GetValues  
GetXmlReader  
HasRows  
IDataRecord.GetData  
IDisposable.Dispose  
IEnumerable.GetEnumerator  
IsClosed  
IsCommandBehavior  
IsDBNull  
IsDBNullAsync  
Item[Int32]  
NextResult  
NextResultAsync  
Read  
ReadAsync  
RecordsAffected  
VisibleFieldCount  
SQLDebugging  
SQLDebugging  
SqlDependency  
AddCommandDependency  
HasChanges

Id  
OnChange  
SqlDependency  
Start  
Stop  
SqlEnclaveAttestationParameters  
ClientDiffieHellmanKey  
GetInput  
Protocol  
SqlEnclaveAttestationParameters  
SqlEnclaveSession  
GetSessionKey  
SessionId  
SqlEnclaveSession  
SqlError  
Class  
LineNumber  
Message  
Number  
Procedure  
Server  
Source  
State  
ToString  
SqlErrorCollection  
CopyTo  
Count  
GetEnumerator  
ICollection.IsSynchronized  
ICollection.SyncRoot  
Item[Int32]  
SqlException  
Class

[ClientConnectionId](#)

[Errors](#)

[GetObjectData](#)

[LineNumber](#)

[Message](#)

[Number](#)

[Procedure](#)

[Server](#)

[Source](#)

[State](#)

[ToString](#)

[SqlInfoMessageEventArgs](#)

[Errors](#)

[Message](#)

[Source](#)

[ToString](#)

[SqlInfoMessageEventHandler](#)

[SqlNotificationEventArgs](#)

[Info](#)

[Source](#)

[SqlNotificationEventArgs](#)

[Type](#)

[SqlNotificationInfo](#)

[SqlNotificationSource](#)

[SqlNotificationType](#)

[SqlParameter](#)

[CompareInfo](#)

[DbType](#)

[Direction](#)

[ForceColumnEncryption](#)

[ICloneable.Clone](#)

[IsNullable](#)

LocaleId  
Offset  
ParameterName  
Precision  
ResetDbType  
ResetSqlDbType  
Scale  
Size  
SourceColumn  
SourceColumnNullMapping  
SourceVersion  
SqlDbType  
SqlParameter  
SqlValue  
ToString  
TypeName  
UdtTypeName  
Value  
XmlSchemaCollectionDatabase  
XmlSchemaCollectionName  
XmlSchemaCollectionOwningSchema  
SqlParameterCollection  
Add  
AddRange  
AddWithValue  
Clear  
Contains  
CopyTo  
Count  
GetEnumerator  
IndexOf  
Insert

- [IsFixedSize](#)
- [IsReadOnly](#)
- [IsSynchronized](#)
- [Item\[Int32\]](#)
- [Remove](#)
- [RemoveAt](#)
- [SyncRoot](#)

[SqlProviderServices](#)

- [SingletonInstance](#)

[SqlRowsCopiedEventArgs](#)

- [Abort](#)
- [RowsCopied](#)
- [SqlRowsCopiedEventArgs](#)

[SqlRowsCopiedEventHandler](#)

[SqlRowUpdatedEventArgs](#)

- [Command](#)
- [SqlRowUpdatedEventArgs](#)

[SqlRowUpdatedEventHandler](#)

[SqlRowUpdatingEventArgs](#)

- [Command](#)
- [SqlRowUpdatingEventArgs](#)

[SqlRowUpdatingEventHandler](#)

[SqlTransaction](#)

- [Commit](#)
- [Connection](#)
- [Dispose](#)
- [IsolationLevel](#)
- [Rollback](#)
- [Save](#)

# System.Data.SqlClient Namespace

The [System.Data.SqlClient](#) namespace is the .NET Data Provider for SQL Server.

## Introduction

The .NET Data Provider for SQL Server describes a collection of classes used to access a SQL Server database in the managed space. Using the [SqlDataAdapter](#), you can fill a memory-resident [DataSet](#) that you can use to query and update the database.

 **Note**

For conceptual information about using this namespace when programming with .NET, see [SQL Server and ADO.NET](#).

## Classes

|  |   |
|--|---|
| <a href="#">SqlAuthenticationInitializer</a>       | Defines the core behavior of authentication initializers that can be registered in the app.config file and provides a base for derived classes.       |
| <a href="#">SqlAuthenticationParameters</a>        | Represents AD authentication parameters passed by a driver to authentication providers.   |
| <a href="#">SqlAuthenticationProvider</a>          | Defines the core behavior of authentication providers and provides a base class for derived classes.  |
| <a href="#">SqlAuthenticationToken</a>             | Represents an AD authentication token.  |
| <a href="#">SqlBulkCopy</a>                        | Lets you efficiently bulk load a SQL Server table with data from another source.  |
| <a href="#">SqlBulkCopyColumnMapping</a>           | Defines the mapping between a column in a <a href="#">SqlBulkCopy</a> instance's data source and a column in the instance's destination table.        |
| <a href="#">SqlBulkCopyColumnMappingCollection</a> | Collection of <a href="#">SqlBulkCopyColumnMapping</a> objects that inherits from <a href="#">CollectionBase</a> .                                    |
| <a href="#">SqlClientFactory</a>                   | Represents a set of methods for creating instances of the <a href="#">System.Data.SqlClient</a> provider's implementation of the data source classes. |
| <a href="#">SqlClientLogger</a>                    | Represents a SQL client logger.   |

|   |   |
|---|---|
| <a href="#">SqlClientMetaDataCollectionNames</a>            | Provides a list of constants for use with the <b>GetSchema</b> method to retrieve metadata collections.   |
| <a href="#">SqlClientPermission</a>                         | Enables the .NET Framework Data Provider for SQL Server to help make sure that a user has a security level sufficient to access a data source.  |
| <a href="#">SqlClientPermissionAttribute</a>                | Associates a security action with a custom security attribute.  |
| <a href="#">SqlColumnEncryptionCertificateStoreProvider</a> | The implementation of the key store provider for Windows Certificate Store. This class enables using certificates stored in the Windows Certificate Store as column master keys. For details, see <a href="#">Always Encrypted</a> .  |
| <a href="#">SqlColumnEncryptionCngProvider</a>              | The CMK Store provider implementation for using the Microsoft Cryptography API: Next Generation (CNG) with <a href="#">Always Encrypted</a> .   |
| <a href="#">SqlColumnEncryptionCspProvider</a>              | The CMK Store provider implementation for using Microsoft CAPI based Cryptographic Service Providers (CSP) with <a href="#">Always Encrypted</a> .  |
| <a href="#">SqlColumnEncryptionEnclaveProvider</a>          | The base class that defines the interface for enclave providers for Always Encrypted.   |
| <a href="#">SqlColumnEncryptionKeyStoreProvider</a>         | Base class for all key store providers. A custom provider must derive from this class and override its member functions and then register it using <code>SqlConnection.RegisterColumnEncryptionKeyStoreProviders()</code> . For details see, <a href="#">Always Encrypted</a> . |
| <a href="#">SqlCommand</a>                                  | Represents a Transact-SQL statement or stored procedure to execute against a SQL Server database. This class cannot be inherited.   |
| <a href="#">SqlCommandBuilder</a>                           | Automatically generates single-table commands that are used to reconcile changes made to a <a href="#">DataSet</a> with the associated SQL Server database. This class cannot be inherited.   |
| <a href="#">SqlConnection</a>                               | Represents a connection to a SQL Server database. This class cannot be inherited.   |

|   |   |
|---|---|
| <a href="#">SqlConnectionStringBuilder</a>      | Provides a simple way to create and manage the contents of connection strings used by the <a href="#">SqlConnection</a> class.  |
| <a href="#">SqlCredential</a>                   | <p><a href="#">SqlCredential</a> provides a more secure way to specify the password for a login attempt using SQL Server Authentication.</p> <p><a href="#">SqlCredential</a> is comprised of a user id and a password that will be used for SQL Server Authentication. The password in a <a href="#">SqlCredential</a> object is of type <a href="#">SecureString</a>.</p> <p><a href="#">SqlCredential</a> cannot be inherited.</p> <p>Windows Authentication (<code>Integrated Security = true</code>) remains the most secure way to log in to a SQL Server database.</p> |
| <a href="#">SqlDataAdapter</a>                  | Represents a set of data commands and a database connection that are used to fill the <a href="#">DataSet</a> and update a SQL Server database. This class cannot be inherited.   |
| <a href="#">SqlDataReader</a>                   | Provides a way of reading a forward-only stream of rows from a SQL Server database. This class cannot be inherited.   |
| <a href="#">SQLDebugging</a>                    | Included to support debugging applications. Not intended for direct use.  |
| <a href="#">SqlDependency</a>                   | The <a href="#">SqlDependency</a> object represents a query notification dependency between an application and an instance of SQL Server. An application can create a <a href="#">SqlDependency</a> object and register to receive notifications via the <a href="#">OnChangeEventHandler</a> event handler.  |
| <a href="#">SqlEnclaveAttestationParameters</a> | Encapsulates the information SqlClient sends to SQL Server to initiate the process of attesting and creating a secure session with the enclave, SQL Server uses for computations on columns protected using Always Encrypted.   |
| <a href="#">SqlEnclaveSession</a>               | Encapsulates the state of a secure session between SqlClient and an enclave inside SQL Server, which can be used for computations on encrypted columns protected with Always Encrypted.   |
| <a href="#">SqlError</a>                        | Collects information relevant to a warning or error returned by SQL Server.   |

|  |   |
|--|---|
| <a href="#">SqlErrorCollection</a>       | Collects all errors generated by the .NET Framework Data Provider for SQL Server. This class cannot be inherited.   |
| <a href="#">SQLException</a>             | The exception that is thrown when SQL Server returns a warning or error. This class cannot be inherited.  |
| <a href="#">SqlInfoMessageEventArgs</a>  | Provides data for the <a href="#">InfoMessage</a> event.  |
| <a href="#">SqlNotificationEventArgs</a> | Represents the set of arguments passed to the notification event handler.   |
| <a href="#">SqlParameter</a>             | Represents a parameter to a <a href="#">SqlCommand</a> and optionally its mapping to <a href="#">DataSet</a> columns. This class cannot be inherited. For more information on parameters, see <a href="#">Configuring Parameters and Parameter Data Types</a> . |
| <a href="#">SqlParameterCollection</a>   | Represents a collection of parameters associated with a <a href="#">SqlCommand</a> and their respective mappings to columns in a <a href="#">DataSet</a> . This class cannot be inherited.  |
| <a href="#">SqlProviderServices</a>      | The DbProviderServices implementation for the SqlClient provider for SQL Server.  |
| <a href="#">SqlRowsCopiedEventArgs</a>   | Represents the set of arguments passed to the <a href="#">SqlRowsCopiedEventHandler</a> .   |
| <a href="#">SqlRowUpdatedEventArgs</a>   | Provides data for the <a href="#">RowUpdated</a> event.   |
| <a href="#">SqlRowUpdatingEventArgs</a>  | Provides data for the <a href="#">RowUpdating</a> event.  |
| <a href="#">SqlTransaction</a>           | Represents a Transact-SQL transaction to be made in a SQL Server database. This class cannot be inherited.  |

## Enums

|                                   |  |
|-----------------------------------|--|
| <a href="#">ApplicationIntent</a> | Specifies a value for <a href="#">ApplicationIntent</a> . Possible values are <a href="#">ReadWrite</a> and <a href="#">ReadOnly</a> . |
|                                   |  |

|   |   |
|---|---|
| <code>PoolBlockingPeriod</code>                   | Specifies a value for the <code>PoolBlockingPeriod</code> property.   |
| <code>SortOrder</code>                            | Specifies how rows of data are sorted.  |
| <code>SqlAuthenticationMethod</code>              | Describes the different SQL authentication methods that can be used by a client connecting to Azure SQL Database. For details, see <a href="#">Connecting to SQL Database By Using Azure Active Directory Authentication</a> .  |
| <code>SqlBulkCopyOptions</code>                   | Bitwise flag that specifies one or more options to use with an instance of <code>SqlBulkCopy</code> .   |
| <code>SqlCommandColumnEncryptionSetting</code>    | Specifies how data will be sent and received when reading and writing encrypted columns. Depending on your specific query, performance impact may be reduced by bypassing the Always Encrypted driver's processing when non-encrypted columns are being used. Note that these settings cannot be used to bypass encryption and gain access to plaintext data. For details, see <a href="#">Always Encrypted (Database Engine)</a> |
| <code>SqlConnectionColumnEncryptionSetting</code> | Specifies that Always Encrypted functionality is enabled in a connection. Note that these settings cannot be used to bypass encryption and gain access to plaintext data. For details, see <a href="#">Always Encrypted (Database Engine)</a> .   |
| <code>SqlNotificationInfo</code>                  | This enumeration provides additional information about the different notifications that can be received by the dependency event handler.  |
| <code>SqlNotificationSource</code>                | Indicates the source of the notification received by the dependency event handler.  |
| <code>SqlNotificationType</code>                  | Describes the different notification types that can be received by an <code>OnChangeEventHandler</code> event handler through the <code>SqlNotificationEventArgs</code> parameter.  |

## Delegates

|                                   |  |
|-----------------------------------|--|
| <code>OnChangeEventHandler</code> | Handles the <code>OnChange</code> event that is fired when a notification is received for any of the commands associated with a <code>SqlDependency</code> object. |
|                                   |  |

|   |  |
|---|--|
| <code>SqlInfoMessageEventHandler</code> | Represents the method that will handle the <a href="#">InfoMessage</a> event of a <a href="#">SqlConnection</a> .  |
| <code>SqlRowsCopiedEventHandler</code>  | Represents the method that handles the <a href="#">SqlRowsCopied</a> event of a <a href="#">SqlBulkCopy</a> .      |
| <code>SqlRowUpdatedEventHandler</code>  | Represents the method that will handle the <a href="#">RowUpdated</a> event of a <a href="#">SqlDataAdapter</a> .  |
| <code>SqlRowUpdatingEventHandler</code> | Represents the method that will handle the <a href="#">RowUpdating</a> event of a <a href="#">SqlDataAdapter</a> . |

# ApplicationIntent ApplicationIntent Enum

Specifies a value for [ApplicationIntent](#). Possible values are `ReadWrite` and `ReadOnly`.

## Declaration

```
[Serializable]
public enum ApplicationIntent

type ApplicationIntent =
```

## Inheritance Hierarchy



## Fields

`ReadOnly` `ReadOnly`

The application workload type when connecting to a server is read only.

`ReadWrite` `ReadWrite`

The application workload type when connecting to a server is read write.

## See Also

# OnChangeEvent Handler Delegate

Handles the [OnChange](#) event that is fired when a notification is received for any of the commands associated with a [SqlDependency](#) object.

## Declaration

```
public delegate void OnChangeEventHandler(object sender, SqlNotificationEventArgs e);  
type OnChangeEventHandler = delegate of obj * SqlNotificationEventArgs -> unit
```

## Inheritance Hierarchy

```
Object Object  
Delegate Delegate
```

## Remarks

The [OnChange](#) event does not necessarily imply a change in the data. Other circumstances, such as time-out expired and failure to set the notification request, also generate [OnChange](#).

## See Also

# PoolBlockingPeriod PoolBlockingPeriod Enum

Specifies a value for the [PoolBlockingPeriod](#) property.

## Declaration

```
[Serializable]
public enum PoolBlockingPeriod

type PoolBlockingPeriod =
```

## Inheritance Hierarchy



## Fields

`AlwaysBlock` `AlwaysBlock`

Blocking period ON for all SQL servers including Azure SQL servers.

`Auto` `Auto`

Blocking period OFF for Azure SQL servers, but ON for all other SQL servers.

`NeverBlock` `NeverBlock`

Blocking period OFF for all SQL servers including Azure SQL servers.

# SortOrder SortOrder Enum

Specifies how rows of data are sorted.

## Declaration

```
public enum SortOrder  
type SortOrder =
```

## Inheritance Hierarchy



## Fields

`Ascending` `Ascending`

Rows are sorted in ascending order.

`Descending` `Descending`

Rows are sorted in descending order.

`Unspecified` `Unspecified`

The default. No sort order is specified.

## See Also

# SqlAuthenticationInitializer Class

Defines the core behavior of authentication initializers that can be registered in the app.config file and provides a base for derived classes.

## Declaration

```
public abstract class SqlAuthenticationInitializer  
type SqlAuthenticationInitializer = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

[SqlAuthenticationInitializer\(\)](#)  
[SqlAuthenticationInitializer\(\)](#)

Called from constructors in derived classes to initialize the [SqlAuthenticationInitializer](#) class.

## Methods

[Initialize\(\)](#)  
[Initialize\(\)](#)

When overridden in a derived class, initializes the authentication initializer. This method is called by the [SqlAuthenticationInitializer\(\)](#) constructor during startup.

# SqlAuthenticationInitializer.Initialize SqlAuthenticationInitializer.Initialize

## In this Article

When overridden in a derived class, initializes the authentication initializer. This method is called by the [SqlAuthenticationInitializer\(\)](#) constructor during startup.

```
public abstract void Initialize ();  
abstract member Initialize : unit -> unit
```

# SqlAuthenticationInitializer

## In this Article

Called from constructors in derived classes to initialize the [SqlAuthenticationInitializer](#) class.

```
protected SqlAuthenticationInitializer ();
```

# SqlAuthenticationMethod SqlAuthenticationMethod Enum

Describes the different SQL authentication methods that can be used by a client connecting to Azure SQL Database. For details, see [Connecting to SQL Database By Using Azure Active Directory Authentication](#).

## Declaration

```
public enum SqlAuthenticationMethod  
type SqlAuthenticationMethod =
```

## Inheritance Hierarchy



## Fields

|                           |
|---------------------------|
| ActiveDirectoryIntegrated |
| ActiveDirectoryIntegrated |

The authentication method uses Active Directory Integrated. Use Active Directory Integrated to connect to a SQL Database using integrated Windows authentication.

|                            |
|----------------------------|
| ActiveDirectoryInteractive |
| ActiveDirectoryInteractive |

The authentication method uses Active Directory Password. Use Active Directory Password to connect to a SQL Database using an Azure AD principal name and password.

|              |
|--------------|
| NotSpecified |
| NotSpecified |

The authentication method is not specified.

|             |
|-------------|
| SqlPassword |
| SqlPassword |

The authentication method is Sql Password.

# SqlAuthenticationParameters SqlAuthenticationParameters Class

Represents AD authentication parameters passed by a driver to authentication providers.

## Declaration

```
public class SqlAuthenticationParameters  
type SqlAuthenticationParameters = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

`SqlAuthenticationParameters(SqlAuthenticationMethod, String, String, String, String, String, String, String, Guid)`

`SqlAuthenticationParameters(SqlAuthenticationMethod, String, String, String, String, String, String, String, Guid)`

Initializes a new instance of the [SqlAuthenticationParameters](#) class using the specified authentication method, server name, database name, resource URI, authority URI, user login name/ID, user password and connection ID.

## Properties

`AuthenticationMethod`

`AuthenticationMethod`

Gets the authentication method.

`Authority`

`Authority`

Gets the authority URI.

`ConnectionId`

`ConnectionId`

Gets the connection ID.

`DatabaseName`

`DatabaseName`

Gets the database name.

`Password`

`Password`

Gets the user password.

Resource

Resource

Gets the resource URI.

ServerName

ServerName

Gets the server name.

UserId

UserId

Gets the user login name/ID.

# SqlAuthenticationParameters.AuthenticationMethod SqlAuthenticationParameters.AuthenticationMethod

## In this Article

Gets the authentication method.

```
public System.Data.SqlClient.SqlAuthenticationMethod AuthenticationMethod { get; }
```

```
member this.AuthenticationMethod : System.Data.SqlClient.SqlAuthenticationMethod
```

Returns

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication method.

# SqlAuthenticationParameters.Authority SqlAuthenticationParameters.Authority

## In this Article

Gets the authority URI.

```
public string Authority { get; }  
member this.Authority : string
```

Returns

[String](#) [String](#)

The authority URI.

# SqlAuthenticationParameters.ConnectionId SqlAuthenticationParameters.ConnectionId

## In this Article

Gets the connection ID.

```
public Guid ConnectionId { get; }  
member this.ConnectionId : Guid
```

Returns

[Guid](#) [Guid](#)

The connection ID.

# SqlAuthenticationParameters.DatabaseName SqlAuthenticationParameters.DatabaseName

## In this Article

Gets the database name.

```
public string DatabaseName { get; }  
member this.DatabaseName : string
```

Returns

[String](#) [String](#)

The database name.

# SqlAuthenticationParameters.Password SqlAuthenticationParameters.Password

## In this Article

Gets the user password.

```
public string Password { get; }  
member this.Password : string
```

Returns

[String](#) [String](#)

The user password.

# SqlAuthenticationParameters.Resource SqlAuthenticationParameters.Resource

## In this Article

Gets the resource URI.

```
public string Resource { get; }  
member this.Resource : string
```

Returns

[String](#) [String](#)

The resource URI.

# SqlAuthenticationParameters.ServerName Sql AuthenticationParameters.ServerName

## In this Article

Gets the server name.

```
public string ServerName { get; }  
member this.ServerName : string
```

Returns

[String](#) [String](#)

The server name.

# SqlAuthenticationParameters

## SqlAuthenticationParameters

### In this Article

Initializes a new instance of the [SqlAuthenticationParameters](#) class using the specified authentication method, server name, database name, resource URI, authority URI, user login name/ID, user password and connection ID.

```
protected SqlAuthenticationParameters (System.Data.SqlClient.SqlAuthenticationMethod authenticationMethod, string serverName, string databaseName, string resource, string authority, string userId, string password, Guid connectionId);  
  
new System.Data.SqlClient.SqlAuthenticationParameters :  
System.Data.SqlClient.SqlAuthenticationMethod * string * string * string * string * string  
* Guid -> System.Data.SqlClient.SqlAuthenticationParameters
```

### Parameters

|   |   |   |
|---|---|---|
| authenticationMethod  | <a href="#">SqlAuthenticationMethod</a> | <a href="#">SqlAuthenticationMethod</a> |
| One of the enumeration values that specifies the authentication method. |   |   |
| serverName  |   | <a href="#">String</a>                  |
| The server name.  |   |   |
| databaseName  |   | <a href="#">String</a>                  |
| The database name.  |   |   |
| resource  |   | <a href="#">String</a>                  |
| The resource URI.   |   |   |
| authority   |   | <a href="#">String</a>                  |
| The authority URI.  |   |   |
| userId  |   | <a href="#">String</a>                  |
| The user login name/ID.   |   |   |
| password  |   | <a href="#">String</a>                  |
| The user password.  |   |   |
| connectionId  |   | <a href="#">Guid</a>                    |
| The connection ID.  |   |   |

# SqlAuthenticationParameters.UserId SqlAuthenticationParameters.UserId

## In this Article

Gets the user login name/ID.

```
public string UserId { get; }  
member this.UserId : string
```

Returns

[String](#) [String](#)

The user login name/ID.

# SqlAuthenticationProvider Class

Defines the core behavior of authentication providers and provides a base class for derived classes.

## Declaration

```
public abstract class SqlAuthenticationProvider  
type SqlAuthenticationProvider = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

Derived classes must provide a default constructor if they can be instantiated from the app.config file.

## Constructors

```
SqlAuthenticationProvider()  
SqlAuthenticationProvider()
```

Called from constructors in derived classes to initialize the [SqlAuthenticationProvider](#) class.

## Methods

```
AcquireTokenAsync(SqlAuthenticationParameters)  
AcquireTokenAsync(SqlAuthenticationParameters)
```

Acquires a security token from the authority.

```
BeforeLoad(SqlAuthenticationMethod)  
BeforeLoad(SqlAuthenticationMethod)
```

This method is called immediately before the provider is added to SQL drivers registry.

```
BeforeUnload(SqlAuthenticationMethod)  
BeforeUnload(SqlAuthenticationMethod)
```

This method is called immediately before the provider is removed from the SQL drivers registry.

```
GetProvider(SqlAuthenticationMethod)  
GetProvider(SqlAuthenticationMethod)
```

Gets an authentication provider by method.

```
IsSupported(SqlAuthenticationMethod)
```

```
IsSupported(SqlAuthenticationMethod)
```

Indicates whether the specified authentication method is supported.

```
SetProvider(SqlAuthenticationMethod, SqlAuthenticationProvider)
```

```
SetProvider(SqlAuthenticationMethod, SqlAuthenticationProvider)
```

Sets an authentication provider by method.

# SqlAuthenticationProvider.AcquireTokenAsync Sql AuthenticationProvider.AcquireTokenAsync

## In this Article

Acquires a security token from the authority.

```
public abstract System.Threading.Tasks.Task<System.Data.SqlClient.SqlAuthenticationToken>  
AcquireTokenAsync (System.Data.SqlClient.SqlAuthenticationParameters parameters);  
  
abstract member AcquireTokenAsync : System.Data.SqlClient.SqlAuthenticationParameters ->  
System.Threading.Tasks.Task<System.Data.SqlClient.SqlAuthenticationToken>
```

## Parameters

parameters [SqlAuthenticationParameters](#) [SqlAuthenticationParameters](#)

The Active Directory authentication parameters passed by the driver to authentication providers.

## Returns

[Task<SqlAuthenticationToken>](#)

Represents an asynchronous operation that returns the AD authentication token.

# SqlAuthenticationProvider.BeforeLoad SqlAuthentication Provider.BeforeLoad

## In this Article

This method is called immediately before the provider is added to SQL drivers registry.

```
public virtual void BeforeLoad (System.Data.SqlClient.SqlAuthenticationMethod authenticationMethod);  
abstract member BeforeLoad : System.Data.SqlClient.SqlAuthenticationMethod -> unit  
override this.BeforeLoad : System.Data.SqlClient.SqlAuthenticationMethod -> unit
```

## Parameters

authenticationMethod

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication method.

## Remarks

Avoid performing long-waiting tasks in this method, since it can block other threads from accessing the provider registry.

# SqlAuthenticationProvider.BeforeUnload SqlAuthenticationProvider.BeforeUnload

## In this Article

This method is called immediately before the provider is removed from the SQL drivers registry.

```
public virtual void BeforeUnload (System.Data.SqlClient.SqlAuthenticationMethod authenticationMethod);  
  
abstract member BeforeUnload : System.Data.SqlClient.SqlAuthenticationMethod -> unit  
override this.BeforeUnload : System.Data.SqlClient.SqlAuthenticationMethod -> unit
```

## Parameters

authenticationMethod

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication method.

## Remarks

For example, this method is called when a different provider with the same authentication method overrides this provider in the SQL drivers registry. Avoid performing long-waiting task in this method, since it can block other threads from accessing the provider registry.

# SqlAuthenticationProvider.GetProvider SqlAuthenticationProvider.GetProvider

## In this Article

Gets an authentication provider by method.

```
public static System.Data.SqlClient.SqlAuthenticationProvider GetProvider  
(System.Data.SqlClient.SqlAuthenticationMethod authenticationMethod);  
  
static member GetProvider : System.Data.SqlClient.SqlAuthenticationMethod ->  
System.Data.SqlClient.SqlAuthenticationProvider
```

## Parameters

authenticationMethod

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication method.

## Returns

[SqlAuthenticationProvider](#) [SqlAuthenticationProvider](#)

The authentication provider or `null` if not found.

# SqlAuthenticationProvider.IsSupported SqlAuthenticationProvider.IsSupported

## In this Article

Indicates whether the specified authentication method is supported.

```
public abstract bool IsSupported (System.Data.SqlClient.SqlAuthenticationMethod authenticationMethod);
```

```
abstract member IsSupported : System.Data.SqlClient.SqlAuthenticationMethod -> bool
```

## Parameters

authenticationMethod

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication method.

## Returns

[Boolean](#) [Boolean](#)

`true` if the specified authentication method is supported; otherwise, `false`.

# SqlAuthenticationProvider.SetProvider SqlAuthenticationProvider.SetProvider

## In this Article

Sets an authentication provider by method.

```
public static bool SetProvider (System.Data.SqlClient.SqlAuthenticationMethod authenticationMethod,  
System.Data.SqlClient.SqlAuthenticationProvider provider);  
  
static member SetProvider : System.Data.SqlClient.SqlAuthenticationMethod *  
System.Data.SqlClient.SqlAuthenticationProvider -> bool
```

## Parameters

authenticationMethod

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication method.

provider

[SqlAuthenticationProvider](#) [SqlAuthenticationProvider](#)

The authentication provider.

## Returns

[Boolean](#) [Boolean](#)

`true` if the operation succeeded; otherwise, `false` (for example, the existing provider disallows overriding).

# SqlAuthenticationProvider

## In this Article

Called from constructors in derived classes to initialize the [SqlAuthenticationProvider](#) class.

```
protected SqlAuthenticationProvider ();
```

# SqlAuthenticationToken SqlAuthenticationToken Class

Represents an AD authentication token.

## Declaration

```
public class SqlAuthenticationToken  
type SqlAuthenticationToken = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

`SqlAuthenticationToken(String, DateTimeOffset)`

`SqlAuthenticationToken(String, DateTimeOffset)`

Initializes a new instance of the [SqlAuthenticationToken](#) class.

## Properties

`AccessToken`

`AccessToken`

Gets the token string.

`ExpiresOn`

`ExpiresOn`

Gets the token expiration time.

# SqlAuthenticationToken AccessToken SqlAuthenticationToken AccessToken

## In this Article

Gets the token string.

```
public string AccessToken { get; }  
member this.AccessToken : string
```

Returns

[String String](#)

The token string.

# SqlAuthenticationToken.ExpiresOn SqlAuthenticationToken.ExpiresOn

## In this Article

Gets the token expiration time.

```
public DateTimeOffset ExpiresOn { get; }  
member this.ExpiresOn : DateTimeOffset
```

Returns

[DateTimeOffset](#) [DateTimeOffset](#)

The token expiration time.

# SqlAuthenticationToken SqlAuthenticationToken

## In this Article

Initializes a new instance of the [SqlAuthenticationToken](#) class.

```
public SqlAuthenticationToken (string accessToken, DateTimeOffset expiresOn);  
new System.Data.SqlClient.SqlAuthenticationToken : string * DateTimeOffset ->  
System.Data.SqlClient.SqlAuthenticationToken
```

## Parameters

|             |                        |
|-------------|------------------------|
| accessToken | <a href="#">String</a> |
|-------------|------------------------|

The access token.

|           |                                |
|-----------|--------------------------------|
| expiresOn | <a href="#">DateTimeOffset</a> |
|-----------|--------------------------------|

The token expiration time.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The `accessToken` parameter is `null` or empty.

# SqlBulkCopy SqlBulkCopy Class

Lets you efficiently bulk load a SQL Server table with data from another source.

## Declaration

```
public sealed class SqlBulkCopy : IDisposable  
type SqlBulkCopy = class  
    interface IDisposable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

Microsoft SQL Server includes a popular command-prompt utility named **bcp** for moving data from one table to another, whether on a single server or between servers. The [SqlBulkCopy](#) class lets you write managed code solutions that provide similar functionality. There are other ways to load data into a SQL Server table (INSERT statements, for example), but [SqlBulkCopy](#) offers a significant performance advantage over them.

The [SqlBulkCopy](#) class can be used to write data only to SQL Server tables. However, the data source is not limited to SQL Server; any data source can be used, as long as the data can be loaded to a [DataTable](#) instance or read with a [IDataReader](#) instance.

[SqlBulkCopy](#) will fail when bulk loading a [DataTable](#) column of type [SqlDateTime](#) into a SQL Server column whose type is one of the date/time types added in SQL Server 2008.

## Constructors

[SqlBulkCopy\(SqlConnection\)](#)

[SqlBulkCopy\(SqlConnection\)](#)

Initializes a new instance of the [SqlBulkCopy](#) class using the specified open instance of [SqlConnection](#).

[SqlBulkCopy\(String\)](#)

[SqlBulkCopy\(String\)](#)

Initializes and opens a new instance of [SqlConnection](#) based on the supplied [connectionString](#). The constructor uses the [SqlConnection](#) to initialize a new instance of the [SqlBulkCopy](#) class.

[SqlBulkCopy\(String, SqlBulkCopyOptions\)](#)

[SqlBulkCopy\(String, SqlBulkCopyOptions\)](#)

Initializes and opens a new instance of [SqlConnection](#) based on the supplied [connectionString](#). The constructor uses that [SqlConnection](#) to initialize a new instance of the [SqlBulkCopy](#) class. The [SqlConnection](#) instance behaves according to options supplied in the [copyOptions](#) parameter.

[SqlBulkCopy\(SqlConnection, SqlBulkCopyOptions, SqlTransaction\)](#)

[SqlBulkCopy\(SqlConnection, SqlBulkCopyOptions, SqlTransaction\)](#)

Initializes a new instance of the [SqlBulkCopy](#) class using the supplied existing open instance of [SqlConnection](#). The [SqlBulkCopy](#) instance behaves according to options supplied in the `copyOptions` parameter. If a non-null [SqlTransaction](#) is supplied, the copy operations will be performed within that transaction.

## Properties

BatchSize

**BatchSize**

Number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server.

BulkCopyTimeout

**BulkCopyTimeout**

Number of seconds for the operation to complete before it times out.

ColumnMappings

**ColumnMappings**

Returns a collection of [SqlBulkCopyColumnMapping](#) items. Column mappings define the relationships between columns in the data source and columns in the destination.

DestinationTableName

**DestinationTableName**

Name of the destination table on the server.

EnableStreaming

**EnableStreaming**

Enables or disables a [SqlBulkCopy](#) object to stream data from an [IDataReader](#) object

NotifyAfter

**NotifyAfter**

Defines the number of rows to be processed before generating a notification event.

## Methods

**Close()**

**Close()**

Closes the [SqlBulkCopy](#) instance.

```
WriteToServer(DataTable, DataRowState)  
WriteToServer(DataTable, DataRowState)
```

Copies only rows that match the supplied row state in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
WriteToServer(IDataReader)  
WriteToServer(IDataReader)
```

Copies all rows in the supplied [IDataReader](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
WriteToServer(DataTable)  
WriteToServer(DataTable)
```

Copies all rows in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
WriteToServer(DbDataReader)  
WriteToServer(DbDataReader)
```

Copies all rows from the supplied [DbDataReader](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
WriteToServer(DataRow[])  
WriteToServer(DataRow[])
```

Copies all rows from the supplied [DataRow](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
WriteToServerAsync(IDataReader, CancellationToken)  
WriteToServerAsync(IDataReader, CancellationToken)
```

The asynchronous version of [WriteToServer\(IDataReader\)](#), which copies all rows in the supplied [IDataReader](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
WriteToServerAsync(DbDataReader)  
WriteToServerAsync(DbDataReader)
```

The asynchronous version of [WriteToServer\(DbDataReader\)](#), which copies all rows from the supplied [DbDataReader](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
WriteToServerAsync(DataRow[])
WriteToServerAsync(DataRow[])
```

The asynchronous version of [WriteToServer\(DataRow\[\]\)](#), which copies all rows from the supplied `DataRow` array to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

```
WriteToServerAsync(DataTable)
WriteToServerAsync(DataTable)
```

The asynchronous version of [WriteToServer\(DataTable\)](#), which copies all rows in the supplied `DataTable` to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

```
WriteToServerAsync(IDataReader)
WriteToServerAsync(IDataReader)
```

The asynchronous version of [WriteToServer\(IDataReader\)](#), which copies all rows in the supplied `IDataReader` to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

```
WriteToServerAsync(DbDataReader, CancellationToken)
WriteToServerAsync(DbDataReader, CancellationToken)
```

The asynchronous version of [WriteToServer\(DbDataReader\)](#), which copies all rows from the supplied `DbDataReader` array to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

```
WriteToServerAsync(DataRow[], CancellationToken)
WriteToServerAsync(DataRow[], CancellationToken)
```

The asynchronous version of [WriteToServer\(DataRow\[\]\)](#), which copies all rows from the supplied `DataRow` array to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
WriteToServerAsync(DataTable, DataRowState)
WriteToServerAsync(DataTable, DataRowState)
```

The asynchronous version of [WriteToServer\(DataTable, DataRowState\)](#), which copies only rows that match the supplied row state in the supplied `DataTable` to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

```
WriteToServerAsync(DataTable, CancellationToken)
WriteToServerAsync(DataTable, CancellationToken)
```

The asynchronous version of [WriteToServer\(DataTable\)](#), which copies all rows in the supplied `DataTable` to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
WriteToServerAsync(DataTable, DataRowState, CancellationToken)
WriteToServerAsync(DataTable, DataRowState, CancellationToken)
```

The asynchronous version of [WriteToServer\(DataTable, DataRowState\)](#), which copies only rows that match the supplied row state in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

## Events

```
SqlRowsCopied
SqlRowsCopied
```

Occurs every time that the number of rows specified by the [NotifyAfter](#) property have been processed.

```
IDisposable.Dispose()
IDisposable.Dispose()
```

Releases all resources used by the current instance of the [SqlBulkCopy](#) class.

## See Also

# SqlBulkCopy.BatchSize SqlBulkCopy.BatchSize

## In this Article

Number of rows in each batch. At the end of each batch, the rows in the batch are sent to the server.

```
public int BatchSize { get; set; }  
member this.BatchSize : int with get, set
```

Returns

[Int32](#) [Int32](#)

The integer value of the [BatchSize](#) property, or zero if no value has been set.

## Examples

The following console application demonstrates how to bulk load data in batches of 50 rows. For an example illustrating how [BatchSize](#) works with a transaction, see [Transaction and Bulk Copy Operations](#).

In this example, the source data is first read from a SQL Server table to a [SqlDataReader](#) instance. The source data does not have to be located on SQL Server; you can use any data source that can be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoMatchingColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Create the SqlBulkCopy object using a connection string.  
            // In the real world you would not use SqlBulkCopy to move  
            // data from one database to another.
```

```

// data from one table to the other in the same database.
using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connectionString))
{
    bulkCopy.DestinationTableName =
        "dbo.BulkCopyDemoMatchingColumns";

    // Set the BatchSize.
    bulkCopy.BatchSize = 50;

    try
    {
        // Write from the source to the destination.
        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

A batch is complete when [BatchSize](#) rows have been processed or there are no more rows to send to the destination data source.

Zero (the default) indicates that each [WriteToServer](#) operation is a single batch.

If the [SqlBulkCopy](#) instance has been declared without the [UseInternalTransaction](#) option in effect, rows are sent to the server [BatchSize](#) rows at a time, but no transaction-related action is taken. If [UseInternalTransaction](#) is in effect, each batch of rows is inserted as a separate transaction.

The [BatchSize](#) property can be set at any time. If a bulk copy is already in progress, the current batch is sized according to the previous batch size. Subsequent batches use the new size. If the [BatchSize](#) is initially zero and changed while a [WriteToServer](#) operation is already in progress, that operation loads the data as a single batch. Any subsequent

[WriteToServer](#) operations on the same [SqlBulkCopy](#) instance use the new [BatchSize](#).

See

Also

[Bulk Copy Operations in SQL Server](#)

[ADO.NET Overview](#)

# SqlBulkCopy.BulkCopyTimeout SqlBulkCopy.BulkCopyTimeout

## In this Article

Number of seconds for the operation to complete before it times out.

```
public int BulkCopyTimeout { get; set; }  
member this.BulkCopyTimeout : int with get, set
```

## Returns

Int32 Int32

The integer value of the [BulkCopyTimeout](#) property. The default is 30 seconds. A value of 0 indicates no limit; the bulk copy will wait indefinitely.

## Examples

The following console application demonstrates how to modify the time-out to 60 seconds when bulk loading data.

In this example, the source data is first read from a SQL Server table to a [SqlDataReader](#) instance. The source data does not have to be located on SQL Server; you can use any data source that can be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL [INSERT ... SELECT](#) statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoMatchingColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();
```

```

// Create the SqlBulkCopy object using a connection string.
// In the real world you would not use SqlBulkCopy to move
// data from one table to the other in the same database.
using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connectionString))
{
    bulkCopy.DestinationTableName =
        "dbo.BulkCopyDemoMatchingColumns";

    // Set the timeout.
    bulkCopy.BulkCopyTimeout = 60;

    try
    {
        // Write from the source to the destination.
        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

If the operation does time out, the transaction is not committed and all copied rows are removed from the destination table.

See

Also

[Performing Bulk Copy Operations](#)  
[ADO.NET Overview](#)

# SqlBulkCopy.Close SqlBulkCopy.Close

## In this Article

Closes the [SqlBulkCopy](#) instance.

```
public void Close ();  
member this.Close : unit -> unit
```

## Examples

The following example uses the same [SqlBulkCopy](#) instance to add sales orders and their associated details to two destination tables. Because the **AdventureWorks** sales order tables are large, the sample reads only orders placed by a certain account number and bulk copies those orders and details to the destination tables. The [Close](#) method is used only after both bulk copy operations are complete.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a connection to the AdventureWorks database.  
        using (SqlConnection connection =  
            new SqlConnection(connectionString))  
        {  
            connection.Open();  
  
            // Empty the destination tables.  
            SqlCommand deleteHeader = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            deleteHeader.ExecuteNonQuery();  
            SqlCommand deleteDetail = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderDetail;",  
                connection);  
            deleteDetail.ExecuteNonQuery();  
  
            // Perform an initial count on the destination  
            // table with matching columns.  
            SqlCommand countRowHeader = new SqlCommand(  
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            long countStartHeader = System.Convert.ToInt32(  
                countRowHeader.ExecuteScalar());  
            Console.WriteLine(  
                "Starting row count for Header table = {0}",  
                countStartHeader);  
  
            // Perform an initial count on the destination  
            // table with different column positions.  
            SqlCommand countRowDetail = new SqlCommand(  
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;",  
                connection);  
            long countStartDetail = System.Convert.ToInt32(  
                countRowDetail.ExecuteScalar());  
            Console.WriteLine(  
                "Starting row count for Detail table = {0}",  
                countStartDetail);  
        }  
    }  
}
```

```

        countRowDetail.ExecuteScalar());
Console.WriteLine(
    "Starting row count for Detail table = {0}",
    countStartDetail);

// Get data from the source table as a SqlDataReader.
// The Sales.SalesOrderHeader and Sales.SalesOrderDetail
// tables are quite large and could easily cause a timeout
// if all data from the tables is added to the destination.
// To keep the example simple and quick, a parameter is
// used to select only orders for a particular account
// as the source for the bulk insert.
SqlCommand headerData = new SqlCommand(
    "SELECT [SalesOrderID], [OrderDate], " +
    "[AccountNumber] FROM [Sales].[SalesOrderHeader] " +
    "WHERE [AccountNumber] = @accountNumber;",
    connection);
SqlParameter parameterAccount = new SqlParameter();
parameterAccount.ParameterName = "@accountNumber";
parameterAccount.SqlDbType = SqlDbType.NVarChar;
parameterAccount.Direction = ParameterDirection.Input;
parameterAccount.Value = "10-4020-000034";
headerData.Parameters.Add(parameterAccount);
SqlDataReader readerHeader = headerData.ExecuteReader();

// Get the Detail data in a separate connection.
using (SqlConnection connection2 = new SqlConnection(connectionString))
{
    connection2.Open();
    SqlCommand sourceDetailData = new SqlCommand(
        "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], [SalesOrderDetailID], " +
        "[OrderQty], [ProductID], [UnitPrice] FROM [Sales].[SalesOrderDetail] " +
        "INNER JOIN [Sales].[SalesOrderHeader] ON [Sales].[SalesOrderDetail]." +
        "[SalesOrderID] = [Sales].[SalesOrderHeader].[SalesOrderID] " +
        "WHERE [AccountNumber] = @accountNumber;", connection2);

    SqlParameter accountDetail = new SqlParameter();
    accountDetail.ParameterName = "@accountNumber";
    accountDetail.SqlDbType = SqlDbType.NVarChar;
    accountDetail.Direction = ParameterDirection.Input;
    accountDetail.Value = "10-4020-000034";
    sourceDetailData.Parameters.Add(accountDetail);
    SqlDataReader readerDetail = sourceDetailData.ExecuteReader();

    // Create the SqlBulkCopy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoOrderHeader";

        // Write readerHeader to the destination.
        try
        {
            bulkCopy.WriteToServer(readerHeader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            readerHeader.Close();
        }
    }
}

```

```

        // Set up a different destination and
        // map columns.
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoOrderDetail";

        // Write readerDetail to the destination.
        try
        {
            bulkCopy.WriteToServer(readerDetail);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            readerDetail.Close();
        }
    }

    // Perform a final count on the destination
    // tables to see how many rows were added.
    long countEndHeader = System.Convert.ToInt32(
        countRowHeader.ExecuteScalar());
    Console.WriteLine("{0} rows were added to the Header table.",
        countEndHeader - countStartHeader);
    long countEndDetail = System.Convert.ToInt32(
        countRowDetail.ExecuteScalar());
    Console.WriteLine("{0} rows were added to the Detail table.",
        countEndDetail - countStartDetail);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}
}
}

private static string GetConnectionString()
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

After you call a [Close](#) on the [SqlBulkCopy](#) object, no other operation will succeed. Calls to the [WriteToServer](#) method will throw an [InvalidOperationException](#).

Calling the [Close](#) method from the [SqlRowsCopied](#) event causes an [InvalidOperationException](#) to be thrown.

Note that open [SqlBulkCopy](#) instances are closed implicitly at the end of a `using` block.

See

Also

[Performing Bulk Copy Operations](#)  
[ADO.NET Overview](#)

# SqlBulkCopy.ColumnMappings SqlBulkCopy.ColumnMappings

## In this Article

Returns a collection of [SqlBulkCopyColumnMapping](#) items. Column mappings define the relationships between columns in the data source and columns in the destination.

```
public System.Data.SqlClient.SqlBulkCopyColumnMappingCollection ColumnMappings { get; }  
member this.ColumnMappings : System.Data.SqlClient.SqlBulkCopyColumnMappingCollection
```

Returns

[SqlBulkCopyColumnMappingCollection](#) [SqlBulkCopyColumnMappingCollection](#)

A collection of column mappings. By default, it is an empty collection.

## Remarks

If the data source and the destination table have the same number of columns, and the ordinal position of each source column within the data source matches the ordinal position of the corresponding destination column, the [ColumnMappings](#) collection is unnecessary. However, if the column counts differ, or the ordinal positions are not consistent, you must use [ColumnMappings](#) to make sure that data is copied into the correct columns.

During the execution of a bulk copy operation, this collection can be accessed, but it cannot be changed. Any attempt to change it will throw an [InvalidOperationException](#).

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopy.DestinationTableName SqlBulkCopy.DestinationTableName

## In this Article

Name of the destination table on the server.

```
public string DestinationTableName { get; set; }  
member this.DestinationTableName : string with get, set
```

## Returns

[String](#)

The string value of the [DestinationTableName](#) property, or null if none as been supplied.

## Examples

The following console application demonstrates how to bulk load data using a connection that is already open. The destination table is a table in the **AdventureWorks** database.

In this example, the connection is first used to read data from a SQL Server table to a [SqlDataReader](#) instance. The source data does not have to be located on SQL Server; you can use any data source that can be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoMatchingColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();
```

```

// Open the destination connection. In the real world you would
// not use SqlBulkCopy to move data from one table to the other
// in the same database. This is for demonstration purposes only.
using (SqlConnection destinationConnection =
    new SqlConnection(connectionString))
{
    destinationConnection.Open();

    // Set up the bulk copy object.
    // Note that the column positions in the source
    // data reader match the column positions in
    // the destination table so there is no need to
    // map columns.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(destinationConnection))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoMatchingColumns";

        try
        {
            // Write from the source to the destination.
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}
}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

If `DestinationTableName` has not been set when `WriteToServer` is called, an `ArgumentNullException` is thrown.

If `DestinationTableName` is modified while a `WriteToServer` operation is running, the change does not affect the

current operation. The new [DestinationTableName](#) value is used the next time a [WriteToServer](#) method is called.

[DestinationTableName](#) is a three-part name (`<database>.<owningschema>.<name>`). You can qualify the table name with its database and owning schema if you choose. However, if the table name uses an underscore ("\_") or any other special characters, you must escape the name using surrounding brackets as in (`[<database>.<owningschema>.<name_01>]`). For more information, see [Database Identifiers](#).

You can bulk-copy data to a temporary table by using a value such as `tempdb..#table` or `tempdb.<owner>.#table` for the [DestinationTableName](#) property.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopy.EnableStreaming SqlBulkCopy.EnableStreaming

## In this Article

Enables or disables a [SqlBulkCopy](#) object to stream data from an [IDataReader](#) object

```
public bool EnableStreaming { get; set; }  
member this.EnableStreaming : bool with get, set
```

Returns

[Boolean](#) Boolean

`true` if a [SqlBulkCopy](#) object can stream data from an [IDataReader](#) object; otherwise, false. The default is `false`.

## Remarks

When `EnableStreaming` is `true`, [SqlBulkCopy](#) reads from an [IDataReader](#) object using [SequentialAccess](#), optimizing memory usage by using the [IDataReader](#) streaming capabilities. When it's set to false, the [SqlBulkCopy](#) class loads all the data returned by the [IDataReader](#) object into memory before sending it to SQL Server or SQL Azure.

# SqlBulkCopy.IDisposable.Dispose

## In this Article

Releases all resources used by the current instance of the [SqlBulkCopy](#) class.

```
void IDisposable.Dispose();
```

## Remarks

Call `Dispose` when you are finished using the [SqlBulkCopy](#). The `Dispose` method leaves the [SqlBulkCopy](#) in an unusable state. After calling `Dispose`, you must release all references to the [SqlBulkCopy](#) so the garbage collector can reclaim the memory that the [SqlBulkCopy](#) was occupying.

For more information, see [Cleaning Up Unmanaged Resources](#) and [Implementing a Dispose Method](#).

**Note**

Always call `Dispose` before you release your last reference to the [SqlBulkCopy](#). Otherwise, the resources it is using will not be freed until the garbage collector calls the [SqlBulkCopy](#) object's `Finalize` method.

See

[ADO.NET Overview](#)

Also

# SqlBulkCopy.NotifyAfter SqlBulkCopy.NotifyAfter

## In this Article

Defines the number of rows to be processed before generating a notification event.

```
public int NotifyAfter { get; set; }  
member this.NotifyAfter : int with get, set
```

Returns

[Int32](#)

The integer value of the [NotifyAfter](#) property, or zero if the property has not been set.

## Examples

The following console application demonstrates how to bulk load data using a connection that is already open. The [NotifyAfter](#) property is set so that the event handler is called after every 50 rows copied to the table.

In this example, the connection is first used to read data from a SQL Server table to a [SqlDataReader](#) instance. Then a second connection is opened to bulk copy the data. Note that the source data does not have to be located on SQL Server; you can use any data source that can be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoMatchingColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("NotifyAfter Sample");  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Create the SqlBulkCopy object using a connection string.  
            // ...
```

```

// In the real world you would not use SqlBulkCopy to move
// data from one table to the other in the same database.
using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connectionString))
{
    bulkCopy.DestinationTableName =
        "dbo.BulkCopyDemoMatchingColumns";

    // Set up the event handler to notify after 50 rows.
    bulkCopy.SqlRowsCopied +=
        new SqlRowsCopiedEventHandler(OnSqlRowsCopied);
    bulkCopy.NotifyAfter = 50;

    try
    {
        // Write from the source to the destination.
        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}

private static void OnSqlRowsCopied(
    object sender, SqlRowsCopiedEventArgs e)
{
    Console.WriteLine("Copied {0} so far...", e.RowsCopied);
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

This property is designed for user interface components that illustrate the progress of a bulk copy operation. It indicates the number of rows to be processed before generating a notification event. The [NotifyAfter](#) property can be set at any time, even while a bulk copy operation is underway. Changes made during a bulk copy operation take effect after the next notification. The new setting applies to all subsequent operations on the same instance.

If [NotifyAfter](#) is set to a number less than zero, an [ArgumentOutOfRangeException](#) is thrown.

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

# SqlBulkCopy SqlBulkCopy

In this Article

## Overloads

|  |   |
|--|---|
| <code>SqlBulkCopy(SqlConnection) SqlBulkCopy(SqlConnection)</code>   | Initializes a new instance of the <code>SqlBulkCopy</code> class using the specified open instance of <code>SqlConnection</code> .  |
| <code>SqlBulkCopy(String) SqlBulkCopy(String)</code>   | Initializes and opens a new instance of <code>SqlConnection</code> based on the supplied <code>connectionString</code> . The constructor uses the <code>SqlConnection</code> to initialize a new instance of the <code>SqlBulkCopy</code> class.  |
| <code>SqlBulkCopy(String, SqlBulkCopyOptions) SqlBulkCopy(String, SqlBulkCopyOptions)</code>   | Initializes and opens a new instance of <code>SqlConnection</code> based on the supplied <code>connectionString</code> . The constructor uses that <code>SqlConnection</code> to initialize a new instance of the <code>SqlBulkCopy</code> class. The <code>SqlConnection</code> instance behaves according to options supplied in the <code>copyOptions</code> parameter.              |
| <code>SqlBulkCopy(SqlConnection, SqlBulkCopyOptions, SqlTransaction) SqlBulkCopy(SqlConnection, SqlBulkCopyOptions, SqlTransaction)</code> | Initializes a new instance of the <code>SqlBulkCopy</code> class using the supplied existing open instance of <code>SqlConnection</code> . The <code>SqlBulkCopy</code> instance behaves according to options supplied in the <code>copyOptions</code> parameter. If a non-null <code>SqlTransaction</code> is supplied, the copy operations will be performed within that transaction. |

## SqlBulkCopy(SqlConnection) SqlBulkCopy(SqlConnection)

Initializes a new instance of the `SqlBulkCopy` class using the specified open instance of `SqlConnection`.

```
public SqlBulkCopy (System.Data.SqlClient.SqlConnection connection);  
new System.Data.SqlClient.SqlBulkCopy : System.Data.SqlClient.SqlConnection ->  
System.Data.SqlClient.SqlBulkCopy
```

### Parameters

connection

`SqlConnection` `SqlConnection`

The already open `SqlConnection` instance that will be used to perform the bulk copy operation. If your connection string does not use `Integrated Security = true`, you can use `SqlCredential` to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

### Examples

The following console application demonstrates how to bulk load data using a connection that is already open. In this example, a `SqlDataReader` is used to copy data from the **Production.Product** table in the SQL Server **AdventureWorks** database to a similar table in the same database. This example is for demonstration purposes only. You would not use `SqlBulkCopy` to move data from one table to another in the same database in a production application. Note that the source data does not have to be located on SQL Server; you can use any data source that can

be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoMatchingColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Open the destination connection. In the real world you would
            // not use SqlBulkCopy to move data from one table to the other
            // in the same database. This is for demonstration purposes only.
            using (SqlConnection destinationConnection =
                new SqlConnection(connectionString))
            {
                destinationConnection.Open();

                // Set up the bulk copy object.
                // Note that the column positions in the source
                // data reader match the column positions in
                // the destination table so there is no need to
                // map columns.
                using (SqlBulkCopy bulkCopy =
                    new SqlBulkCopy(destinationConnection))
                {
                    bulkCopy.DestinationTableName =
                        "dbo.BulkCopyDemoMatchingColumns";

                    try
                    {
                        // Write from the source to the destination.
                        bulkCopy.WriteToServer(reader);
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(ex.Message);
                    }
                }
            }
        }
    }
}
```

```

        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

Because the connection is already open when the [SqlBulkCopy](#) instance is initialized, the connection remains open after the [SqlBulkCopy](#) instance is closed.

If the `connection` argument is null, an [ArgumentNullException](#) is thrown.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## SqlBulkCopy(String) SqlBulkCopy(String)

Initializes and opens a new instance of [SqlConnection](#) based on the supplied `connectionString`. The constructor uses the [SqlConnection](#) to initialize a new instance of the [SqlBulkCopy](#) class.

```

public SqlBulkCopy (string connectionString);

new System.Data.SqlClient.SqlBulkCopy : string -> System.Data.SqlClient.SqlBulkCopy

```

### Parameters

`connectionString`

[String](#) [String](#)

The string defining the connection that will be opened for use by the [SqlBulkCopy](#) instance. If your connection string does not use `Integrated Security = true`, you can use [SqlBulkCopy\(SqlConnection\)](#) or [SqlBulkCopy\(SqlConnection, SqlBulkCopyOptions, SqlTransaction\)](#) and [SqlCredential](#) to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

### Examples

The following console application demonstrates how to bulk load data by using a connection specified as a string. The connection is automatically closed when the [SqlBulkCopy](#) instance is closed.

In this example, the source data is first read from a SQL Server table to a [SqlDataReader](#) instance. The source data does not have to be located on SQL Server; you can use any data source that can be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoMatchingColumns",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Set up the bulk copy object using a connection string.
            // In the real world you would not use SqlBulkCopy to move
            // data from one table to the other in the same database.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoMatchingColumns";

                try
                {
                    // Write from the source to the destination.
                    bulkCopy.WriteToServer(reader);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
                finally
                {
                    // Close the SqlDataReader. The SqlBulkCopy
                }
            }
        }
    }
}
```

```

        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

The connection is automatically closed at the end of the bulk copy operation.

If `connectionString` is null, an [ArgumentNullException](#) is thrown. If `connectionString` is an empty string, an [ArgumentException](#) is thrown.

See

[Bulk Copy Operations in SQL Server](#)

Also

[ADO.NET Overview](#)

## SqlBulkCopy(String, SqlBulkCopyOptions) SqlBulkCopy(String, SqlBulkCopyOptions)

Initializes and opens a new instance of [SqlConnection](#) based on the supplied `connectionString`. The constructor uses that [SqlConnection](#) to initialize a new instance of the [SqlBulkCopy](#) class. The [SqlConnection](#) instance behaves according to options supplied in the `copyOptions` parameter.

```

public SqlBulkCopy (string connectionString, System.Data.SqlClient.SqlBulkCopyOptions copyOptions);

new System.Data.SqlClient.SqlBulkCopy : string * System.Data.SqlClient.SqlBulkCopyOptions ->
System.Data.SqlClient.SqlBulkCopy

```

## Parameters

connectionString

[String](#)

The string defining the connection that will be opened for use by the [SqlBulkCopy](#) instance. If your connection string does not use `Integrated Security = true`, you can use [SqlBulkCopy\(SqlConnection\)](#) or [SqlBulkCopy\(SqlConnection, SqlBulkCopyOptions, SqlTransaction\)](#) and [SqlCredential](#) to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

copyOptions

[SqlBulkCopyOptions](#)

A combination of values from the [SqlBulkCopyOptions](#) enumeration that determines which data source rows are

copied to the destination table.

## Examples

The following console application demonstrates how to perform a bulk load by using a connection specified as a string. An option is set to use the value in the identity column of the source table when you load the destination table. In this example, the source data is first read from a SQL Server table to a `SqlDataReader` instance. The source table and destination table each include an Identity column. By default, a new value for the **Identity** column is generated in the destination table for each row added. In this example, an option is set when the connection is opened that forces the bulk load process to use the **Identity** values from the source table instead. To see how the option changes the way the bulk load works, run the sample with the `dbo.BulkCopyDemoMatchingColumns` table empty. All rows load from the source. Then run the sample again without emptying the table. An exception is thrown and the code writes a message to the console notifying you that rows weren't added because of primary key constraint violations.

### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using `SqlBulkCopy` only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoMatchingColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Create the SqlBulkCopy object using a connection string
            // and the KeepIdentity option.
            // In the real world you would not use SqlBulkCopy to move
            // data from one table to the other in the same database.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString, SqlBulkCopyOptions.KeepIdentity))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoMatchingColumns";

                try
                {
                    // Write from the source to the destination.
                    bulkCopy.WriteToServer(reader);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
}
```

```

        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

You can obtain detailed information about all the bulk copy options in the [SqlBulkCopyOptions](#) topic.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## **SqlBulkCopy(SqlConnection, SqlBulkCopyOptions, SqlTransaction)**

## **SqlBulkCopy(SqlConnection, SqlBulkCopyOptions, SqlTransaction)**

Initializes a new instance of the [SqlBulkCopy](#) class using the supplied existing open instance of [SqlConnection](#). The [SqlBulkCopy](#) instance behaves according to options supplied in the `copyOptions` parameter. If a non-null [SqlTransaction](#) is supplied, the copy operations will be performed within that transaction.

```

public SqlBulkCopy (System.Data.SqlClient.SqlConnection connection,
System.Data.SqlClient.SqlBulkCopyOptions copyOptions, System.Data.SqlClient.SqlTransaction
externalTransaction);

new System.Data.SqlClient.SqlBulkCopy : System.Data.SqlClient.SqlConnection *
System.Data.SqlClient.SqlBulkCopyOptions * System.Data.SqlClient.SqlTransaction ->
System.Data.SqlClient.SqlBulkCopy

```

## Parameters

connection

[SqlConnection](#) [SqlConnection](#)

The already open [SqlConnection](#) instance that will be used to perform the bulk copy. If your connection string does not

use `Integrated Security = true`, you can use [SqlCredential](#) to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

copyOptions

[SqlBulkCopyOptions](#) [SqlBulkCopyOptions](#)

A combination of values from the [SqlBulkCopyOptions](#) enumeration that determines which data source rows are copied to the destination table.

externalTransaction

[SqlTransaction](#) [SqlTransaction](#)

An existing [SqlTransaction](#) instance under which the bulk copy will occur.

## Remarks

If options include `UseInternalTransaction` and the `externalTransaction` argument is not null, an **InvalidOperationException** is thrown.

For examples demonstrating how to use [SqlBulkCopy](#) in a transaction, see [Transaction and Bulk Copy Operations](#).

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopy.SqlRowsCopied SqlBulkCopy.SqlRowsCopied

## In this Article

Occurs every time that the number of rows specified by the [NotifyAfter](#) property have been processed.

```
public event System.Data.SqlClient.SqlRowsCopiedEventHandler SqlRowsCopied;  
member this.SqlRowsCopied : System.Data.SqlClient.SqlRowsCopiedEventHandler
```

## Examples

The following console application demonstrates how to bulk load data using a connection that is already open. The [NotifyAfter](#) property is set so that the event handler is called after every 50 rows copied to the table.

In this example, the connection is first used to read data from a SQL Server table to a [SqlDataReader](#) instance. Note that the source data does not have to be located on SQL Server; you can use any data source that can be read to an [IDataReader](#) or loaded to a [DataTable](#).

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
              new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoMatchingColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("NotifyAfter Sample");  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Create the SqlBulkCopy object using a connection string.  
            // In the real world you would not use SqlBulkCopy to move  
            // data from one table to the other in the same database.  
            using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connectionString))  
            {
```

```

        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoMatchingColumns";

        // Set up the event handler to notify after 50 rows.
        bulkCopy.SqlRowsCopied +=
            new SqlRowsCopiedEventHandler(OnSqlRowsCopied);
        bulkCopy.NotifyAfter = 50;

        try
        {
            // Write from the source to the destination.
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static void OnSqlRowsCopied(
    object sender, SqlRowsCopiedEventArgs e)
{
    Console.WriteLine("Copied {0} so far...", e.RowsCopied);
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

Note that the settings of [NotifyAfter](#) and [BatchSize](#) are independent. Receipt of a [SqlRowsCopied](#) event does not imply that any rows have been sent to the server or committed.

You cannot call `SqlBulkCopy.Close (Close)` or `SqlConnection.Close (Close)` from this event. Doing this will cause an [InvalidOperationException](#) being thrown, and the [SqlBulkCopy](#) object state will not change. If the user wants to cancel the operation from the event, the [Abort](#) property of the [SqlRowsCopiedEventArgs](#) can be used. (See [Transaction and Bulk Copy Operations](#) for examples that use the [Abort](#) property.)

No action, such as transaction activity, is supported in the connection during the execution of the bulk copy operation, and it is recommended that you not use the same connection used during the [SqlRowsCopied](#) event. However, you can open a different connection.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopy.WriteToServer SqlBulkCopy.WriteToServer

In this Article

## Overloads

|  |  |
|--|--|
| <code>WriteToServer(DataTable, DataRowState)</code><br><code>WriteToServer(DataTable, DataRowState)</code> | Copies only rows that match the supplied row state in the supplied <code>DataTable</code> to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object. |
| <code>WriteToServer(IDataReader)</code><br><code>WriteToServer(IDataReader)</code>                         | Copies all rows in the supplied <code>IDataReader</code> to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.                                  |
| <code>WriteToServer(DataTable)</code><br><code>WriteToServer(DataTable)</code>                             | Copies all rows in the supplied <code>DataTable</code> to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.                                    |
| <code>WriteToServer(DbDataReader)</code><br><code>WriteToServer(DbDataReader)</code>                       | Copies all rows from the supplied <code>DbDataReader</code> array to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.                         |
| <code>WriteToServer(DataRow[])</code><br><code>WriteToServer(DataRow[])</code>                             | Copies all rows from the supplied <code>DataRow</code> array to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.                              |

## Remarks

If multiple active result sets (MARS) is disabled, `WriteToServer` makes the connection busy. If MARS is enabled, you can interleave calls to `WriteToServer` with other commands in the same connection.

### **WriteToServer(DataTable, DataRowState)**

### **WriteToServer(DataTable, DataRowState)**

Copies only rows that match the supplied row state in the supplied `DataTable` to a destination table specified by the `DestinationTableName` property of the `SqlBulkCopy` object.

```
public void WriteToServer (System.Data.DataTable table, System.Data.DataRowState rowState);  
member this.WriteToServer : System.Data.DataTable * System.Data.DataRowState -> unit
```

Parameters

table

`DataTable` `DataTable`

A `DataTable` whose rows will be copied to the destination table.

rowState

`DataRowState` `DataRowState`

A value from the `DataRowState` enumeration. Only rows matching the row state are copied to the destination.

## Examples

The following Console application demonstrates how to bulk load only the rows in a [DataTable](#) that match a specified state. In this case, only unchanged rows are added. The destination table is a table in the **AdventureWorks** database.

In this example, a [DataTable](#) is created at run time and three rows are added to it. Before the [WriteToServer](#) method is executed, one of the rows is edited. The [WriteToServer](#) method is called with a `DataRowState.Unchanged` `rowState` argument, so only the two unchanged rows are bulk copied to the destination.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a connection to the AdventureWorks database.
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            connection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoMatchingColumns;",
                connection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Create a table with some rows.
            DataTable newProducts = MakeTable();

            // Make a change to one of the rows in the DataTable.
            DataRow row = newProducts.Rows[0];
            row.BeginEdit();
            row["Name"] = "AAA";
            row.EndEdit();

            // Create the SqlBulkCopy object.
            // Note that the column positions in the source DataTable
            // match the column positions in the destination table so
            // there is no need to map columns.
            using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connection))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoMatchingColumns";

                try
                {
                    // Write unchanged rows from the source to the destination.
                    bulkCopy.WriteToServer(newProducts, DataRowState.Unchanged);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
}
```

```

        }

        // Perform a final count on the destination
        // table to see how many rows were added.
        long countEnd = System.Convert.ToInt32(
            commandRowCount.ExecuteScalar());
        Console.WriteLine("Ending row count = {0}", countEnd);
        Console.WriteLine("{0} rows were added.", countEnd - countStart);
        Console.WriteLine("Press Enter to finish.");
        Console.ReadLine();
    }
}

private static DataTable MakeTable()
    // Create a new DataTable named NewProducts.
{
    DataTable newProducts = new DataTable("NewProducts");

    // Add three column objects to the table.
    DataColumn productID = new DataColumn();
    productID.DataType = System.Type.GetType("System.Int32");
    productID.ColumnName = "ProductID";
    productID.AutoIncrement = true;
    newProducts.Columns.Add(productID);

    DataColumn productName = new DataColumn();
    productName.DataType = System.Type.GetType("System.String");
    productName.ColumnName = "Name";
    newProducts.Columns.Add(productName);

    DataColumn productNumber = new DataColumn();
    productNumber.DataType = System.Type.GetType("System.String");
    productNumber.ColumnName = "ProductNumber";
    newProducts.Columns.Add(productNumber);

    // Create an array for DataColumn objects.
    DataColumn[] keys = new DataColumn[1];
    keys[0] = productID;
    newProducts.PrimaryKey = keys;

    // Add some new rows to the collection.
    DataRow row = newProducts.NewRow();
    row["Name"] = "CC-101-WH";
    row["ProductNumber"] = "Cyclocomputer - White";

    newProducts.Rows.Add(row);
    row = newProducts.NewRow();
    row["Name"] = "CC-101-BK";
    row["ProductNumber"] = "Cyclocomputer - Black";

    newProducts.Rows.Add(row);
    row = newProducts.NewRow();
    row["Name"] = "CC-101-ST";
    row["ProductNumber"] = "Cyclocomputer - Stainless";
    newProducts.Rows.Add(row);
    newProducts.AcceptChanges();

    // Return the new DataTable.
    return newProducts;
}
private static string GetConnectionString()
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +

```

```
        "Initial Catalog=AdventureWorks;";  
    }  
}
```

## Remarks

Only rows in the [DataTable](#) that are in the states indicated in the `rowState` argument and have not been deleted are copied to the destination table.

### [Note](#)

If [Deleted](#) is specified, any [Unchanged](#), [Added](#), and [Modified](#) rows will also be copied to the server. No exception will be raised.

While the bulk copy operation is in progress, the associated destination [SqlConnection](#) is busy serving it, and no other operations can be performed on the connection.

The [ColumnMappings](#) collection maps from the [DataTable](#) columns to the destination database table.

### See

[DataRowState](#)[DataRowState](#)

### Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

## WriteToServer([IDataReader](#)) WriteToServer([IDataReader](#))

Copies all rows in the supplied [IDataReader](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public void WriteToServer (System.Data.IDataReader reader);  
member this.WriteToServer : System.Data.IDataReader -> unit
```

### Parameters

#### reader

[IDataReader](#) [IDataReader](#)

A [IDataReader](#) whose rows will be copied to the destination table.

### Examples

The following console application demonstrates how to bulk load data from a [SqlDataReader](#). The destination table is a table in the [AdventureWorks](#) database.

#### [Important](#)

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
              new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.
```

```

// Perform an initial count on the destination table.
SqlCommand commandRowCount = new SqlCommand(
    "SELECT COUNT(*) FROM " +
    "dbo.BulkCopyDemoMatchingColumns;",
    sourceConnection);
long countStart = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Starting row count = {0}", countStart);

// Get data from the source table as a SqlDataReader.
SqlCommand commandSourceData = new SqlCommand(
    "SELECT ProductID, Name, " +
    "ProductNumber " +
    "FROM Production.Product;", sourceConnection);
SqlDataReader reader =
    commandSourceData.ExecuteReader();

// Set up the bulk copy object using a connection string.
// In the real world you would not use SqlBulkCopy to move
// data from one table to the other in the same database.
using (SqlBulkCopy bulkCopy =
    new SqlBulkCopy(connectionString))
{
    bulkCopy.DestinationTableName =
        "dbo.BulkCopyDemoMatchingColumns";

    try
    {
        // Write from the source to the destination.
        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}

}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        "Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

The copy operation starts at the next available row in the reader. Most of the time, the reader was just returned by [ExecuteReader](#) or a similar call, so the next available row is the first row. To process multiple results, call [NextResult](#) on the data reader and call [WriteToServer](#) again.

Note that using [WriteToServer](#) modifies the state of the reader. The method will call [Read](#) until it returns false, the operation is aborted, or an error occurs. This means that the data reader will be in a different state, probably at the end of the result set, when the [WriteToServer](#) operation is complete.

While the bulk copy operation is in progress, the associated destination [SqlConnection](#) is busy serving it, and no other operations can be performed on the connection.

The [ColumnMappings](#) collection maps from the data reader columns to the destination database table.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## WriteToServer(DataTable) WriteToServer(DataTable)

Copies all rows in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public void WriteToServer (System.Data.DataTable table);  
member this.WriteToServer : System.Data.DataTable -> unit
```

### Parameters

table

[DataTable](#) [DataTable](#)

A [DataTable](#) whose rows will be copied to the destination table.

### Examples

The following Console application demonstrates how to bulk load data from a [DataTable](#). The destination table is a table in the **AdventureWorks** database.

In this example, a [DataTable](#) is created at run time and is the source of the [SqlBulkCopy](#) operation.

#### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL [INSERT ... SELECT](#) statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a connection to the AdventureWorks database.  
        using (SqlConnection connection =  
              new SqlConnection(connectionString))  
        {  
            connection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +
```

```

        "dbo.BulkCopyDemoMatchingColumns",
        connection);
long countStart = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Starting row count = {0}", countStart);

// Create a table with some rows.
DataTable newProducts = MakeTable();

// Create the SqlBulkCopy object.
// Note that the column positions in the source DataTable
// match the column positions in the destination table so
// there is no need to map columns.
using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connection))
{
    bulkCopy.DestinationTableName =
        "dbo.BulkCopyDemoMatchingColumns";

    try
    {
        // Write from the source to the destination.
        bulkCopy.WriteToServer(newProducts);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}

}

private static DataTable MakeTable()
    // Create a new DataTable named NewProducts.
{
    DataTable newProducts = new DataTable("NewProducts");

    // Add three column objects to the table.
    DataColumn productID = new DataColumn();
    productID.DataType = System.Type.GetType("System.Int32");
    productID.ColumnName = "ProductID";
    productID.AutoIncrement = true;
    newProducts.Columns.Add(productID);

    DataColumn productName = new DataColumn();
    productName.DataType = System.Type.GetType("System.String");
    productName.ColumnName = "Name";
    newProducts.Columns.Add(productName);

    DataColumn productNumber = new DataColumn();
    productNumber.DataType = System.Type.GetType("System.String");
    productNumber.ColumnName = "ProductNumber";
    newProducts.Columns.Add(productNumber);

    // Create an array for DataColumn objects.
    DataColumn[] keys = new DataColumn[1];
    keys[0] = productID;
    newProducts.PrimaryKey = keys;
}

```

```

newProducts.PrimaryKey = keys;

// Add some new rows to the collection.
DataRow row = newProducts.NewRow();
row["Name"] = "CC-101-WH";
row["ProductNumber"] = "Cyclocomputer - White";

newProducts.Rows.Add(row);
row = newProducts.NewRow();
row["Name"] = "CC-101-BK";
row["ProductNumber"] = "Cyclocomputer - Black";

newProducts.Rows.Add(row);
row = newProducts.NewRow();
row["Name"] = "CC-101-ST";
row["ProductNumber"] = "Cyclocomputer - Stainless";
newProducts.Rows.Add(row);
newProducts.AcceptChanges();

// Return the new DataTable.
return newProducts;
}

private static string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}
}

```

## Remarks

All rows in the [DataTable](#) are copied to the destination table except those that have been deleted.

While the bulk copy operation is in progress, the associated destination [SqlConnection](#) is busy serving it, and no other operations can be performed on the connection.

The [ColumnMappings](#) collection maps from the [DataTable](#) columns to the destination database table.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## WriteToServer(DbDataReader) WriteToServer(DbDataReader)

Copies all rows from the supplied [DbDataReader](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```

public void WriteToServer (System.Data.Common.DbDataReader reader);
member this.WriteToServer : System.Data.Common.DbDataReader -> unit

```

### Parameters

reader

[DbDataReader](#) [DbDataReader](#)

A [DbDataReader](#) whose rows will be copied to the destination table.

## WriteToServer(DataRow[]) WriteToServer(DataRow[])

Copies all rows from the supplied [DataRow](#) array to a destination table specified by the [DestinationTableName](#)

property of the [SqlBulkCopy](#) object.

```
public void WriteToServer (System.Data.DataRow[] rows);
member this.WriteToServer : System.Data.DataRow[] -> unit
```

## Parameters

rows [DataRow](#)[]

An array of [DataRow](#) objects that will be copied to the destination table.

## Examples

The following console application demonstrates how to bulk load data from a [DataRow](#) array. The destination table is a table in the **AdventureWorks** database.

In this example, a [DataTable](#) is created at run time. A single row is selected from the [DataTable](#) to copy to the destination table.

### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a connection to the AdventureWorks database.
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            connection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoMatchingColumns",
                connection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Create a table with some rows.
            DataTable newProducts = MakeTable();

            // Get a reference to a single row in the table.
            DataRow[] rowArray = newProducts.Select(
                "Name='CC-101-BK'");

            // Create the SqlBulkCopy object.
            // Note that the column positions in the source DataTable
            // match the column positions in the destination table so
            // there is no need to map columns.
            using (SqlBulkCopy bulkCopy = new SqlBulkCopy(connection))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoMatchingColumns";
            }
        }
    }
}
```

```

        }

        // Write the array of rows to the destination.
        bulkCopy.WriteToServer(rowArray);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}
}

private static DataTable MakeTable()
    // Create a new DataTable named NewProducts.
{
    DataTable newProducts = new DataTable("NewProducts");

    // Add three column objects to the table.
    DataColumn productID = new DataColumn();
    productID.DataType = System.Type.GetType("System.Int32");
    productID.ColumnName = "ProductID";
    productID.AutoIncrement = true;
    newProducts.Columns.Add(productID);

    DataColumn productName = new DataColumn();
    productName.DataType = System.Type.GetType("System.String");
    productName.ColumnName = "Name";
    newProducts.Columns.Add(productName);

    DataColumn productNumber = new DataColumn();
    productNumber.DataType = System.Type.GetType("System.String");
    productNumber.ColumnName = "ProductNumber";
    newProducts.Columns.Add(productNumber);

    // Create an array for DataColumn objects.
    DataColumn[] keys = new DataColumn[1];
    keys[0] = productID;
    newProducts.PrimaryKey = keys;

    // Add some new rows to the collection.
    DataRow row = newProducts.NewRow();
    row["Name"] = "CC-101-WH";
    row["ProductNumber"] = "Cyclocomputer - White";

    newProducts.Rows.Add(row);
    row = newProducts.NewRow();
    row["Name"] = "CC-101-BK";
    row["ProductNumber"] = "Cyclocomputer - Black";

    newProducts.Rows.Add(row);
    row = newProducts.NewRow();
    row["Name"] = "CC-101-ST";
    row["ProductNumber"] = "Cyclocomputer - Stainless";
    newProducts.Rows.Add(row);
    newProducts.AcceptChanges();
}

```

```
// Return the new DataTable.  
    return newProducts;  
}  
  
private static string GetConnectionString()  
    // To avoid storing the connection string in your code,  
    // you can retrieve it from a configuration file.  
{  
    return "Data Source=(local); " +  
        " Integrated Security=true;" +  
        "Initial Catalog=AdventureWorks;";  
}  
}
```

## Remarks

While the bulk copy operation is in progress, the associated destination [SqlConnection](#) is busy serving it, and no other operations can be performed on the connection.

The [ColumnMappings](#) collection maps from the [DataRow](#) columns to the destination database table.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopy.WriteToServerAsync SqlBulkCopy.WriteToServerAsync

In this Article

## Overloads

|  |  |
|--|--|
| <code>WriteToServerAsync(IDataReader, CancellationToken)</code><br><code>WriteToServerAsync(IDataReader, CancellationToken)</code>   | <p>The asynchronous version of <a href="#">WriteToServer(IDataReader)</a>, which copies all rows in the supplied <a href="#">IDataReader</a> to a destination table specified by the <a href="#">DestinationTableName</a> property of the <a href="#">SqlBulkCopy</a> object.</p> <p>The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.</p> |
| <code>WriteToServerAsync(DbDataReader)</code><br><code>WriteToServerAsync(DbDataReader)</code>                                       | <p>The asynchronous version of <a href="#">WriteToServer(DbDataReader)</a>, which copies all rows from the supplied <a href="#">DbDataReader</a> array to a destination table specified by the <a href="#">DestinationTableName</a> property of the <a href="#">SqlBulkCopy</a> object.</p>  |
| <code>WriteToServerAsync(DataRow[])</code><br><code>WriteToServerAsync(DataRow[])</code>   | <p>The asynchronous version of <a href="#">WriteToServer(DataRow[])</a>, which copies all rows from the supplied <a href="#">DataRow</a> array to a destination table specified by the <a href="#">DestinationTableName</a> property of the <a href="#">SqlBulkCopy</a> object.</p>  |
| <code>WriteToServerAsync(DataTable)</code><br><code>WriteToServerAsync(DataTable)</code>   | <p>The asynchronous version of <a href="#">WriteToServer(DataTable)</a>, which copies all rows in the supplied <a href="#">DataTable</a> to a destination table specified by the <a href="#">DestinationTableName</a> property of the <a href="#">SqlBulkCopy</a> object.</p>  |
| <code>WriteToServerAsync(IDataReader)</code><br><code>WriteToServerAsync(IDataReader)</code>   | <p>The asynchronous version of <a href="#">WriteToServer(IDataReader)</a>, which copies all rows in the supplied <a href="#">IDataReader</a> to a destination table specified by the <a href="#">DestinationTableName</a> property of the <a href="#">SqlBulkCopy</a> object.</p>  |
| <code>WriteToServerAsync(DbDataReader, CancellationToken)</code><br><code>WriteToServerAsync(DbDataReader, CancellationToken)</code> | <p>The asynchronous version of <a href="#">WriteToServer(DbDataReader)</a>, which copies all rows from the supplied <a href="#">DbDataReader</a> array to a destination table specified by the <a href="#">DestinationTableName</a> property of the <a href="#">SqlBulkCopy</a> object.</p>  |

|  |  |
|--|--|
| <code>WriteToServerAsync(DataRow[], CancellationToken)</code><br><code>WriteToServerAsync(DataRow[], CancellationToken)</code>                             | <p>The asynchronous version of <a href="#">WriteToServer(DataRow[])</a>, which copies all rows from the supplied <code>DataRow</code> array to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.</p> <p>The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.</p>  |
| <code>WriteToServerAsync(DataTable, DataRowState)</code><br><code>WriteToServerAsync(DataTable, DataRowState)</code>                                       | <p>The asynchronous version of <a href="#">WriteToServer(DataTable, DataRowState)</a>, which copies only rows that match the supplied row state in the supplied <code>DataTable</code> to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.</p>  |
| <code>WriteToServerAsync(DataTable, CancellationToken)</code><br><code>WriteToServerAsync(DataTable, CancellationToken)</code>                             | <p>The asynchronous version of <a href="#">WriteToServer(DataTable)</a>, which copies all rows in the supplied <code>DataTable</code> to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.</p> <p>The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.</p>  |
| <code>WriteToServerAsync(DataTable, DataRowState, CancellationToken)</code><br><code>WriteToServerAsync(DataTable, DataRowState, CancellationToken)</code> | <p>The asynchronous version of <a href="#">WriteToServer(DataTable, DataRowState)</a>, which copies only rows that match the supplied row state in the supplied <code>DataTable</code> to a destination table specified by the <code>DestinationTableName</code> property of the <code>SqlBulkCopy</code> object.</p> <p>The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.</p> |

## Remarks

If multiple active result sets (MARS) is disabled, [WriteToServer](#) makes the connection busy. If MARS is enabled, you can interleave calls to [WriteToServer](#) with other commands in the same connection.

The number of rows that are rolled back when one fails depends on several things:

- If `UseInternalTransaction` is specified.
- If you have your own transaction.
- The value of `BatchSize`.

When there is an error while sending data to the server, the current batch (as specified by `BatchSize`) will be rolled back. If `UseInternalTransaction` is not specified and you have your own transaction, the entire transaction will be rolled back (which includes all previous batches as well).

Use `SqlRowsCopied` to know how many rows were copied to the server.

## **WriteToServerAsync(IDataReader, CancellationToken)**

## **WriteToServerAsync(IDataReader, CancellationToken)**

The asynchronous version of [WriteToServer\(IDataReader\)](#), which copies all rows in the supplied [IDataReader](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.IDataReader reader,
System.Threading.CancellationToken cancellationToken);

member this.WriteToServerAsync : System.Data.IDataReader * System.Threading.CancellationToken ->
System.Threading.Tasks.Task
```

### Parameters

reader [IDataReader](#) [IDataReader](#)

A [IDataReader](#) whose rows will be copied to the destination table.

cancellationToken [CancellationToken](#) [CancellationToken](#)

The cancellation instruction. A [None](#) value in this parameter makes this method equivalent to [WriteToServerAsync\(DataTable\)](#).

### Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(IDataReader\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(IDataReader\)](#) and [WriteToServer\(IDataReader\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(IDataReader\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

The [IDataReader](#) was closed before the completed [Task](#) returned.

The [IDataReader](#)'s associated connection was closed before the completed [Task](#) returned.

`Context Connection=true` is specified in the connection string.

### [SQLException](#) [SQLException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## **WriteToServerAsync(DbDataReader)**

## **WriteToServerAsync(DbDataReader)**

The asynchronous version of [WriteToServer\(DbDataReader\)](#), which copies all rows from the supplied [DbDataReader](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.Common.DbDataReader reader);  
member this.WriteToServerAsync : System.Data.Common.DbDataReader -> System.Threading.Tasks.Task
```

Parameters

reader [DbDataReader](#) [DbDataReader](#)

A [DbDataReader](#) whose rows will be copied to the destination table.

Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

## **WriteToServerAsync(DataRow[])** **WriteToServerAsync(DataRow[])**

The asynchronous version of [WriteToServer\(DataRow\[\]\)](#), which copies all rows from the supplied [DataRow](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.DataRow[] rows);  
member this.WriteToServerAsync : System.Data.DataRow[] -> System.Threading.Tasks.Task
```

Parameters

rows [DataRow](#)[] [DataRow](#)[]

An array of [DataRow](#) objects that will be copied to the destination table.

Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(DataRow\[\]\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(DataRow\[\]\)](#) and [WriteToServer\(DataRow\[\]\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(DataRow\[\]\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

[Context Connection=true](#) is specified in the connection string.

[SqlException](#) [SqlException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

## Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## **WriteToServerAsync(DataTable) WriteToServerAsync(DataTable)**

The asynchronous version of [WriteToServer\(DataTable\)](#), which copies all rows in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.DataTable table);  
member this.WriteToServerAsync : System.Data.DataTable -> System.Threading.Tasks.Task
```

## Parameters

table [DataTable](#) [DataTable](#)

A [DataTable](#) whose rows will be copied to the destination table.

## Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(DataTable\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(DataTable\)](#) and [WriteToServer\(DataTable\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(DataTable\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

`Context Connection=true` is specified in the connection string.

[SQLException](#) [SQLException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

## Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## **WriteToServerAsync(IDataReader) WriteToServerAsync(IDataReader)**

The asynchronous version of [WriteToServer\(IDataReader\)](#), which copies all rows in the supplied [IDataReader](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.IDataReader reader);  
member this.WriteToServerAsync : System.Data.IDataReader -> System.Threading.Tasks.Task
```

#### Parameters

reader [IDataReader](#) [IDataReader](#)

A [IDataReader](#) whose rows will be copied to the destination table.

#### Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

#### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(IDataReader\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(IDataReader\)](#) and [WriteToServer\(IDataReader\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(IDataReader\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

The [IDataReader](#) was closed before the completed [Task](#) returned.

The [IDataReader](#)'s associated connection was closed before the completed [Task](#) returned.

`Context Connection=true` is specified in the connection string.

[SqlException](#) [SqlException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

#### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## **WriteToServerAsync(DbDataReader, CancellationToken)** **WriteToServerAsync(DbDataReader, CancellationToken)**

The asynchronous version of [WriteToServer\(DbDataReader\)](#), which copies all rows from the supplied [DbDataReader](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.Common.DbDataReader reader,  
System.Threading.CancellationToken cancellationToken);  
member this.WriteToServerAsync : System.Data.Common.DbDataReader *  
System.Threading.CancellationToken -> System.Threading.Tasks.Task
```

#### Parameters

reader [DbDataReader](#) [DbDataReader](#)

A [DbDataReader](#) whose rows will be copied to the destination table.

cancellationToken

CancellationToken CancellationToken

The cancellation instruction. A [None](#) value in this parameter makes this method equivalent to [WriteToServerAsync\(DbDataReader\)](#).

Returns

[Task](#) Task

Returns [Task](#).

## **WriteToServerAsync(DataRow[], CancellationToken)**

## **WriteToServerAsync(DataRow[], CancellationToken)**

The asynchronous version of [WriteToServer\(DataRow\[\]\)](#), which copies all rows from the supplied [DataRow](#) array to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned [Task](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.DataRow[] rows,
System.Threading.CancellationToken cancellationToken);

member this.WriteToServerAsync : System.Data.DataRow[] * System.Threading.CancellationToken ->
System.Threading.Tasks.Task
```

Parameters

rows

[DataRow](#)[]

An array of [DataRow](#) objects that will be copied to the destination table.

cancellationToken

CancellationToken CancellationToken

The cancellation instruction. A [None](#) value in this parameter makes this method equivalent to [WriteToServerAsync\(DataTable\)](#).

Returns

[Task](#) Task

A task representing the asynchronous operation.

Exceptions

[InvalidOperationException](#) InvalidOperationException

Calling [WriteToServerAsync\(DataRow\[\]\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(DataRow\[\]\)](#) and [WriteToServer\(DataRow\[\]\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(DataRow\[\]\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

`Context Connection=true` is specified in the connection string.

## [SqlException](#) [SqlException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## **WriteToServerAsync(DataTable, DataRowState)** **WriteToServerAsync(DataTable, DataRowState)**

The asynchronous version of [WriteToServer\(DataTable, DataRowState\)](#), which copies only rows that match the supplied row state in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.DataTable table,  
System.Data.DataRowState rowState);  
  
member this.WriteToServerAsync : System.Data.DataTable * System.Data.DataRowState ->  
System.Threading.Tasks.Task
```

### Parameters

table [DataTable](#) [DataTable](#)

A [DataTable](#) whose rows will be copied to the destination table.

rowState [DataRowState](#) [DataRowState](#)

A value from the [DataRowState](#) enumeration. Only rows matching the row state are copied to the destination.

### Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(DataTable, DataRowState\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(DataTable, DataRowState\)](#) and [WriteToServer\(DataTable, DataRowState\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(DataTable, DataRowState\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

`Context Connection=true` is specified in the connection string.

## [SqlException](#) [SqlException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## WriteToServerAsync(DataTable, CancellationToken)

## WriteToServerAsync(DataTable, CancellationToken)

The asynchronous version of [WriteToServer\(DataTable\)](#), which copies all rows in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.DataTable table,  
System.Threading.CancellationToken cancellationToken);  
  
member this.WriteToServerAsync : System.Data.DataTable * System.Threading.CancellationToken ->  
System.Threading.Tasks.Task
```

### Parameters

table [DataTable](#) [DataTable](#)

A [DataTable](#) whose rows will be copied to the destination table.

cancellationToken [CancellationToken](#) [CancellationToken](#)

The cancellation instruction. A [None](#) value in this parameter makes this method equivalent to [WriteToServerAsync\(DataTable\)](#).

### Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(DataTable\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(DataTable\)](#) and [WriteToServer\(DataTable\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(DataTable\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

`Context Connection=true` is specified in the connection string.

[SQLException](#) [SQLException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## **WriteToServerAsync(DataTable, DataRowState, CancellationToken)**

## **WriteToServerAsync(DataTable, DataRowState, CancellationToken)**

The asynchronous version of [WriteToServer\(DataTable, DataRowState\)](#), which copies only rows that match the supplied row state in the supplied [DataTable](#) to a destination table specified by the [DestinationTableName](#) property of the [SqlBulkCopy](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task WriteToServerAsync (System.Data.DataTable table,  
System.Data.DataRowState rowState, System.Threading.CancellationToken cancellationToken);  
  
member this.WriteToServerAsync : System.Data.DataTable * System.Data.DataRowState *  
System.Threading.CancellationToken -> System.Threading.Tasks.Task
```

### Parameters

table [DataTable](#) [DataTable](#)

A [DataTable](#) whose rows will be copied to the destination table.

rowState [DataRowState](#) [DataRowState](#)

A value from the [DataRowState](#) enumeration. Only rows matching the row state are copied to the destination.

cancellationToken [CancellationToken](#) [CancellationToken](#)

The cancellation instruction. A [None](#) value in this parameter makes this method equivalent to [WriteToServerAsync\(DataTable\)](#).

### Returns

[Task](#) [Task](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [WriteToServerAsync\(DataTable, DataRowState\)](#) multiple times for the same instance before task completion.

Calling [WriteToServerAsync\(DataTable, DataRowState\)](#) and [WriteToServer\(DataTable, DataRowState\)](#) for the same instance before task completion.

The connection drops or is closed during [WriteToServerAsync\(DataTable, DataRowState\)](#) execution.

Returned in the task object, the [SqlBulkCopy](#) object was closed during the method execution.

Returned in the task object, there was a connection pool timeout.

Returned in the task object, the [SqlConnection](#) object is closed before method execution.

`Context Connection=true` is specified in the connection string.

[SQLException](#) [SQLException](#)

Returned in the task object, any error returned by SQL Server that occurred while opening the connection.

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see

Asynchronous Programming.

# SqlBulkCopyColumnMapping SqlBulkCopyColumn Mapping Class

Defines the mapping between a column in a [SqlBulkCopy](#) instance's data source and a column in the instance's destination table.

## Declaration

```
public sealed class SqlBulkCopyColumnMapping  
type SqlBulkCopyColumnMapping = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

Column mappings define the mapping between data source and the target table.

If mappings are not defined—that is, the [ColumnMappings](#) collection is empty—the columns are mapped implicitly based on ordinal position. For this to work, source and target schemas must match. If they do not, an [InvalidOperationException](#) will be thrown.

If the [ColumnMappings](#) collection is not empty, not every column present in the data source has to be specified. Those not mapped by the collection are ignored.

You can refer to source and target columns by either name or ordinal. You can also mix by-name and by-ordinal column references in the same mappings collection.

## Constructors

[SqlBulkCopyColumnMapping\(\)](#)

[SqlBulkCopyColumnMapping\(\)](#)

Default constructor that initializes a new [SqlBulkCopyColumnMapping](#) object.

[SqlBulkCopyColumnMapping\(Int32, Int32\)](#)

[SqlBulkCopyColumnMapping\(Int32, Int32\)](#)

Creates a new column mapping, using column ordinals to refer to source and destination columns.

[SqlBulkCopyColumnMapping\(Int32, String\)](#)

[SqlBulkCopyColumnMapping\(Int32, String\)](#)

Creates a new column mapping, using a column ordinal to refer to the source column and a column name for the target column.

[SqlBulkCopyColumnMapping\(String, Int32\)](#)

[SqlBulkCopyColumnMapping\(String, Int32\)](#)

Creates a new column mapping, using a column name to refer to the source column and a column ordinal for the

target column.

```
SqlBulkCopyColumnMapping(String, String)  
SqlBulkCopyColumnMapping(String, String)
```

Creates a new column mapping, using column names to refer to source and destination columns.

## Properties

DestinationColumn

DestinationColumn

Name of the column being mapped in the destination database table.

DestinationOrdinal

DestinationOrdinal

Ordinal value of the destination column within the destination table.

SourceColumn

SourceColumn

Name of the column being mapped in the data source.

SourceOrdinal

SourceOrdinal

The ordinal position of the source column within the data source.

## See Also

# SqlBulkCopyColumnMapping.DestinationColumn SqlBulkCopyColumnMapping.DestinationColumn

## In this Article

Name of the column being mapped in the destination database table.

```
public string DestinationColumn { get; set; }  
member this.DestinationColumn : string with get, set
```

Returns

[String](#)

The string value of the [DestinationColumn](#) property.

## Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoDifferentColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Set up the bulk copy object.  
            using (SqlBulkCopy bulkCopy =
```

```

        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings source and destination.
        SqlBulkCopyColumnMapping mapID = new SqlBulkCopyColumnMapping();
        mapID.SourceColumn = "ProductID";
        mapID.DestinationColumn = "ProdID";
        bulkCopy.ColumnMappings.Add(mapID);

        SqlBulkCopyColumnMapping mapName = new SqlBulkCopyColumnMapping();
        mapName.SourceColumn = "Name";
        mapName.DestinationColumn = "ProdName";
        bulkCopy.ColumnMappings.Add(mapName);

        SqlBulkCopyColumnMapping mapNumber = new SqlBulkCopyColumnMapping();
        mapNumber.SourceColumn = "ProductNumber";
        mapNumber.DestinationColumn = "ProdNum";
        bulkCopy.ColumnMappings.Add(mapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

The [DestinationColumn](#) and [DestinationOrdinal](#) properties are mutually exclusive. The last value set takes precedence.

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

# SqlBulkCopyColumnMapping.DestinationOrdinal SqlBulkCopyColumnMapping.DestinationOrdinal

## In this Article

Ordinal value of the destination column within the destination table.

```
public int DestinationOrdinal { get; set; }  
member this.DestinationOrdinal : int with get, set
```

Returns

Int32 Int32

The integer value of the [DestinationOrdinal](#) property, or -1 if the property has not been set.

## Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoDifferentColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Set up the bulk copy object.  
            using (SqlBulkCopy bulkCopy =
```

```

        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings source and destination.
        SqlBulkCopyColumnMapping mapID = new SqlBulkCopyColumnMapping();
        mapID.SourceOrdinal = 0;
        mapID.DestinationOrdinal = 0;
        bulkCopy.ColumnMappings.Add(mapID);

        SqlBulkCopyColumnMapping mapName = new SqlBulkCopyColumnMapping();
        mapName.SourceOrdinal = 1;
        mapName.DestinationOrdinal = 2;
        bulkCopy.ColumnMappings.Add(mapName);

        SqlBulkCopyColumnMapping mapNumber = new SqlBulkCopyColumnMapping();
        mapNumber.SourceOrdinal = 2;
        mapNumber.DestinationOrdinal = 1;
        bulkCopy.ColumnMappings.Add(mapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

The [DestinationColumn](#) and [DestinationOrdinal](#) properties are mutually exclusive. The last value set takes precedence.

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

# SqlBulkCopyColumnMapping.SourceColumn SqlBulkCopyColumnMapping.SourceColumn

## In this Article

Name of the column being mapped in the data source.

```
public string SourceColumn { get; set; }  
member this.SourceColumn : string with get, set
```

## Returns

[String](#)

The string value of the [SourceColumn](#) property.

## Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoDifferentColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Set up the bulk copy object.  
            using (SqlBulkCopy bulkCopy =
```

```

        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings source and destination.
        SqlBulkCopyColumnMapping mapID = new SqlBulkCopyColumnMapping();
        mapID.SourceColumn = "ProductID";
        mapID.DestinationColumn = "ProdID";
        bulkCopy.ColumnMappings.Add(mapID);

        SqlBulkCopyColumnMapping mapName = new SqlBulkCopyColumnMapping();
        mapName.SourceColumn = "Name";
        mapName.DestinationColumn = "ProdName";
        bulkCopy.ColumnMappings.Add(mapName);

        SqlBulkCopyColumnMapping mapNumber = new SqlBulkCopyColumnMapping();
        mapNumber.SourceColumn = "ProductNumber";
        mapNumber.DestinationColumn = "ProdNum";
        bulkCopy.ColumnMappings.Add(mapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

The [SourceColumn](#) and [SourceOrdinal](#) properties are mutually exclusive. The last value set takes precedence.

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

# SqlBulkCopyColumnMapping.SourceOrdinal SqlBulkCopyColumnMapping.SourceOrdinal

## In this Article

The ordinal position of the source column within the data source.

```
public int SourceOrdinal { get; set; }  
member this.SourceOrdinal : int with get, set
```

Returns

[Int32](#) [Int32](#)

The integer value of the [SourceOrdinal](#) property.

## Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a sourceConnection to the AdventureWorks database.  
        using (SqlConnection sourceConnection =  
            new SqlConnection(connectionString))  
        {  
            sourceConnection.Open();  
  
            // Perform an initial count on the destination table.  
            SqlCommand commandRowCount = new SqlCommand(  
                "SELECT COUNT(*) FROM " +  
                "dbo.BulkCopyDemoDifferentColumns;",  
                sourceConnection);  
            long countStart = System.Convert.ToInt32(  
                commandRowCount.ExecuteScalar());  
            Console.WriteLine("Starting row count = {0}", countStart);  
  
            // Get data from the source table as a SqlDataReader.  
            SqlCommand commandSourceData = new SqlCommand(  
                "SELECT ProductID, Name, " +  
                "ProductNumber " +  
                "FROM Production.Product;", sourceConnection);  
            SqlDataReader reader =  
                commandSourceData.ExecuteReader();  
  
            // Set up the bulk copy object.  
            using (SqlBulkCopy bulkCopy =
```

```

        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings source and destination.
        SqlBulkCopyColumnMapping mapID = new SqlBulkCopyColumnMapping();
        mapID.SourceOrdinal = 0;
        mapID.DestinationOrdinal = 0;
        bulkCopy.ColumnMappings.Add(mapID);

        SqlBulkCopyColumnMapping mapName = new SqlBulkCopyColumnMapping();
        mapName.SourceOrdinal = 1;
        mapName.DestinationOrdinal = 2;
        bulkCopy.ColumnMappings.Add(mapName);

        SqlBulkCopyColumnMapping mapNumber = new SqlBulkCopyColumnMapping();
        mapNumber.SourceOrdinal = 2;
        mapNumber.DestinationOrdinal = 1;
        bulkCopy.ColumnMappings.Add(mapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

The [SourceColumn](#) and [SourceOrdinal](#) properties are mutually exclusive. The last value set takes precedence.

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

# SqlBulkCopyColumnMapping SqlBulkCopyColumn Mapping

In this Article

## Overloads

|  |   |
|--|---|
| <code>SqlBulkCopyColumnMapping()</code>  | Default constructor that initializes a new <a href="#">SqlBulkCopyColumnMapping</a> object.                                 |
| <code>SqlBulkCopyColumnMapping(Int32, Int32) SqlBulkCopyColumnMapping(Int32, Int32)</code>     | Creates a new column mapping, using column ordinals to refer to source and destination columns.                             |
| <code>SqlBulkCopyColumnMapping(Int32, String) SqlBulkCopyColumnMapping(Int32, String)</code>   | Creates a new column mapping, using a column ordinal to refer to the source column and a column name for the target column. |
| <code>SqlBulkCopyColumnMapping(String, Int32) SqlBulkCopyColumnMapping(String, Int32)</code>   | Creates a new column mapping, using a column name to refer to the source column and a column ordinal for the target column. |
| <code>SqlBulkCopyColumnMapping(String, String) SqlBulkCopyColumnMapping(String, String)</code> | Creates a new column mapping, using column names to refer to source and destination columns.                                |

## SqlBulkCopyColumnMapping()

Default constructor that initializes a new [SqlBulkCopyColumnMapping](#) object.

```
public SqlBulkCopyColumnMapping();
```

### Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

#### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the Adventureworks database.
```

```
// Open a source connection to the Adventureworks database.
using (SqlConnection sourceConnection =
    new SqlConnection(connectionString))
{
    sourceConnection.Open();

    // Perform an initial count on the destination table.
    SqlCommand commandRowCount = new SqlCommand(
        "SELECT COUNT(*) FROM " +
        "dbo.BulkCopyDemoDifferentColumns;",
        sourceConnection);
    long countStart = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Starting row count = {0}", countStart);

    // Get data from the source table as a SqlDataReader.
    SqlCommand commandSourceData = new SqlCommand(
        "SELECT ProductID, Name, " +
        "ProductNumber " +
        "FROM Production.Product;", sourceConnection);
    SqlDataReader reader =
        commandSourceData.ExecuteReader();

    // Set up the bulk copy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings by name.
        SqlBulkCopyColumnMapping mapID =
            new SqlBulkCopyColumnMapping("ProductID", "ProdID");
        bulkCopy.ColumnMappings.Add(mapID);

        SqlBulkCopyColumnMapping mapName =
            new SqlBulkCopyColumnMapping("Name", "ProdName");
        bulkCopy.ColumnMappings.Add(mapName);

        SqlBulkCopyColumnMapping mapNumber =
            new SqlBulkCopyColumnMapping("ProductNumber", "ProdNum");
        bulkCopy.ColumnMappings.Add(mapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
}
```

```

        Console.WriteLine("{0} rows were added.", countEnd - countStart);
        Console.WriteLine("Press Enter to finish.");
        Console.ReadLine();
    }

    private static string GetConnectionString()
        // To avoid storing the sourceConnection string in your code,
        // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

If you use this constructor, you must then define the source for the mapping using the [SourceColumn](#) property or the [SourceOrdinal](#) property, and define the destination for the mapping using the [DestinationColumn](#) property or the [DestinationOrdinal](#) property.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## SqlBulkCopyColumnMapping(Int32, Int32) SqlBulkCopyColumnMapping(Int32, Int32)

Creates a new column mapping, using column ordinals to refer to source and destination columns.

```

public SqlBulkCopyColumnMapping (int sourceColumnOrdinal, int destinationOrdinal);

new System.Data.SqlClient.SqlBulkCopyColumnMapping : int * int ->
System.Data.SqlClient.SqlBulkCopyColumnMapping

```

### Parameters

sourceColumnOrdinal

[Int32](#) [Int32](#)

The ordinal position of the source column within the data source.

destinationOrdinal

[Int32](#) [Int32](#)

The ordinal position of the destination column within the destination table.

### Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy based on the ordinal positions of the columns.

#### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```

using System.Data.SqlClient;

class Program
{

```

```
static void Main()
{
    string connectionString = GetConnectionString();
    // Open a sourceConnection to the AdventureWorks database.
    using (SqlConnection sourceConnection =
        new SqlConnection(connectionString))
    {
        sourceConnection.Open();

        // Perform an initial count on the destination table.
        SqlCommand commandRowCount = new SqlCommand(
            "SELECT COUNT(*) FROM " +
            "dbo.BulkCopyDemoDifferentColumns;",
            sourceConnection);
        long countStart = System.Convert.ToInt32(
            commandRowCount.ExecuteScalar());
        Console.WriteLine("Starting row count = {0}", countStart);

        // Get data from the source table as a SqlDataReader.
        SqlCommand commandSourceData = new SqlCommand(
            "SELECT ProductID, Name, " +
            "ProductNumber " +
            "FROM Production.Product;", sourceConnection);
        SqlDataReader reader =
            commandSourceData.ExecuteReader();

        // Set up the bulk copy object.
        using (SqlBulkCopy bulkCopy =
            new SqlBulkCopy(connectionString))
        {
            bulkCopy.DestinationTableName =
                "dbo.BulkCopyDemoDifferentColumns";

            // Set up the column mappings by ordinal.
            SqlBulkCopyColumnMapping columnMapID =
                new SqlBulkCopyColumnMapping(0, 0);
            bulkCopy.ColumnMappings.Add(columnMapID);

            SqlBulkCopyColumnMapping columnMapName =
                new SqlBulkCopyColumnMapping(1, 2);
            bulkCopy.ColumnMappings.Add(columnMapName);

            SqlBulkCopyColumnMapping columnMapNumber =
                new SqlBulkCopyColumnMapping(2, 1);
            bulkCopy.ColumnMappings.Add(columnMapNumber);

            // Write from the source to the destination.
            try
            {
                bulkCopy.WriteToServer(reader);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                // Close the SqlDataReader. The SqlBulkCopy
                // object is automatically closed at the end
                // of the using block.
                reader.Close();
            }
        }

        // Perform a final count on the destination
        // table.
```

```

        // table to see how many rows were added.
        long countEnd = System.Convert.ToInt32(
            commandRowCount.ExecuteScalar());
        Console.WriteLine("Ending row count = {0}", countEnd);
        Console.WriteLine("{0} rows were added.", countEnd - countStart);
        Console.WriteLine("Press Enter to finish.");
        Console.ReadLine();
    }
}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}

```

See

[ADO.NET Overview](#)

Also

## **SqlBulkCopyColumnMapping(Int32, String)**

## **SqlBulkCopyColumnMapping(Int32, String)**

Creates a new column mapping, using a column ordinal to refer to the source column and a column name for the target column.

```

public SqlBulkCopyColumnMapping (int sourceColumnOrdinal, string destinationColumn);

new System.Data.SqlClient.SqlBulkCopyColumnMapping : int * string ->
System.Data.SqlClient.SqlBulkCopyColumnMapping

```

Parameters

sourceColumnOrdinal

**Int32** **Int32**

The ordinal position of the source column within the data source.

destinationColumn

**String** **String**

The name of the destination column within the destination table.

Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {

```

```

    string connectionString = GetConnectionString();
    // Open a sourceConnection to the AdventureWorks database.
    using (SqlConnection sourceConnection =
        new SqlConnection(connectionString))
    {
        sourceConnection.Open();

        // Perform an initial count on the destination table.
        SqlCommand commandRowCount = new SqlCommand(
            "SELECT COUNT(*) FROM " +
            "dbo.BulkCopyDemoDifferentColumns",
            sourceConnection);
        long countStart = System.Convert.ToInt32(
            commandRowCount.ExecuteScalar());
        Console.WriteLine("Starting row count = {0}", countStart);

        // Get data from the source table as a SqlDataReader.
        SqlCommand commandSourceData = new SqlCommand(
            "SELECT ProductID, Name, " +
            "ProductNumber " +
            "FROM Production.Product;", sourceConnection);
        SqlDataReader reader =
            commandSourceData.ExecuteReader();

        // Set up the bulk copy object.
        using (SqlBulkCopy bulkCopy =
            new SqlBulkCopy(connectionString))
        {
            bulkCopy.DestinationTableName =
                "dbo.BulkCopyDemoDifferentColumns";

            // Set up the column mappings by ordinal and name.
            SqlBulkCopyColumnMapping columnMapID =
                new SqlBulkCopyColumnMapping(0, "ProdID");
            bulkCopy.ColumnMappings.Add(columnMapID);

            SqlBulkCopyColumnMapping columnMapName =
                new SqlBulkCopyColumnMapping(1, "ProdName");
            bulkCopy.ColumnMappings.Add(columnMapName);

            SqlBulkCopyColumnMapping columnMapNumber =
                new SqlBulkCopyColumnMapping(2, "ProdNum");
            bulkCopy.ColumnMappings.Add(columnMapNumber);

            // Write from the source to the destination.
            try
            {
                bulkCopy.WriteToServer(reader);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            finally
            {
                // Close the SqlDataReader. The SqlBulkCopy
                // object is automatically closed at the end
                // of the using block.
                reader.Close();
            }
        }

        // Perform a final count on the destination
        // table to see how many rows were added.
        long countEnd = System.Convert.ToInt32(

```

```

        commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
    }
}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

## SqlBulkCopyColumnMapping(String, Int32)

## SqlBulkCopyColumnMapping(String, Int32)

Creates a new column mapping, using a column name to refer to the source column and a column ordinal for the target column.

```

public SqlBulkCopyColumnMapping (string sourceColumn, int destinationOrdinal);

new System.Data.SqlClient.SqlBulkCopyColumnMapping : string * int ->
System.Data.SqlClient.SqlBulkCopyColumnMapping

```

Parameters

sourceColumn

[String](#)

The name of the source column within the data source.

destinationOrdinal

[Int32](#)

The ordinal position of the destination column within the destination table.

Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();

```

```

// Open a sourceConnection to the AdventureWorks database.
using (SqlConnection sourceConnection =
    new SqlConnection(connectionString))
{
    sourceConnection.Open();

    // Perform an initial count on the destination table.
    SqlCommand commandRowCount = new SqlCommand(
        "SELECT COUNT(*) FROM " +
        "dbo.BulkCopyDemoDifferentColumns;",
        sourceConnection);
    long countStart = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Starting row count = {0}", countStart);

    // Get data from the source table as a SqlDataReader.
    SqlCommand commandSourceData = new SqlCommand(
        "SELECT ProductID, Name, " +
        "ProductNumber " +
        "FROM Production.Product;", sourceConnection);
    SqlDataReader reader =
        commandSourceData.ExecuteReader();

    // Set up the bulk copy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings by name and ordinal.
        SqlBulkCopyColumnMapping columnMapID =
            new SqlBulkCopyColumnMapping("ProductID", 0);
        bulkCopy.ColumnMappings.Add(columnMapID);

        SqlBulkCopyColumnMapping columnMapName =
            new SqlBulkCopyColumnMapping("Name", 2);
        bulkCopy.ColumnMappings.Add(columnMapName);

        SqlBulkCopyColumnMapping columnMapNumber =
            new SqlBulkCopyColumnMapping("ProductNumber", 1);
        bulkCopy.ColumnMappings.Add(columnMapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
}

```

```

        Console.WriteLine("Ending row count = {0}", countEnd);
        Console.WriteLine("{0} rows were added.", countEnd - countStart);
        Console.WriteLine("Press Enter to finish.");
        Console.ReadLine();
    }
}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

## SqlBulkCopyColumnMapping(String, String)

## SqlBulkCopyColumnMapping(String, String)

Creates a new column mapping, using column names to refer to source and destination columns.

```

public SqlBulkCopyColumnMapping (string sourceColumn, string destinationColumn);

new System.Data.SqlClient.SqlBulkCopyColumnMapping : string * string ->
System.Data.SqlClient.SqlBulkCopyColumnMapping

```

Parameters

|              |                        |
|--------------|------------------------|
| sourceColumn | <a href="#">String</a> |
|--------------|------------------------|

The name of the source column within the data source.

|                   |                        |
|-------------------|------------------------|
| destinationColumn | <a href="#">String</a> |
|-------------------|------------------------|

The name of the destination column within the destination table.

Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =

```

```

        new SqlConnection(connectionString))
{
    sourceConnection.Open();

    // Perform an initial count on the destination table.
    SqlCommand commandRowCount = new SqlCommand(
        "SELECT COUNT(*) FROM " +
        "dbo.BulkCopyDemoDifferentColumns;",
        sourceConnection);
    long countStart = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Starting row count = {0}", countStart);

    // Get data from the source table as a SqlDataReader.
    SqlCommand commandSourceData = new SqlCommand(
        "SELECT ProductID, Name, " +
        "ProductNumber " +
        "FROM Production.Product;", sourceConnection);
    SqlDataReader reader =
        commandSourceData.ExecuteReader();

    // Set up the bulk copy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoDifferentColumns";

        // Set up the column mappings by name.
        SqlBulkCopyColumnMapping mapID =
            new SqlBulkCopyColumnMapping("ProductID", "ProdID");
        bulkCopy.ColumnMappings.Add(mapID);

        SqlBulkCopyColumnMapping mapName =
            new SqlBulkCopyColumnMapping("Name", "ProdName");
        bulkCopy.ColumnMappings.Add(mapName);

        SqlBulkCopyColumnMapping mapNumber =
            new SqlBulkCopyColumnMapping("ProductNumber", "ProdNum");
        bulkCopy.ColumnMappings.Add(mapNumber);

        // Write from the source to the destination.
        try
        {
            bulkCopy.WriteToServer(reader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
}

```

```
        Console.ReadLine();
    }
}

private static string GetConnectionString()
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
```

See

Also

[Performing Bulk Copy Operations](#)

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection SqlBulkCopy ColumnMappingCollection Class

Collection of [SqlBulkCopyColumnMapping](#) objects that inherits from [CollectionBase](#).

## Declaration

```
public sealed class SqlBulkCopyColumnMappingCollection : System.Collections.CollectionBase  
type SqlBulkCopyColumnMappingCollection = class  
    inherit CollectionBase
```

## Inheritance Hierarchy



## Remarks

Column mappings define the mapping between data source and the target table.

If mappings are not defined—that is, the [ColumnMappings](#) collection is empty—the columns are mapped implicitly based on ordinal position. For this to work, source and target schemas must match. If they do not, an [InvalidOperationException](#) is thrown.

If the [ColumnMappings](#) collection is not empty, not every column present in the data source has to be specified. Those not mapped by the collection are ignored.

You can refer to source and target columns by either name or ordinal. You can mix by-name and by-ordinal column references in the same mappings collection.

## Properties

Count

Count

Item[Int32]

Item[Int32]

Gets the [SqlBulkCopyColumnMapping](#) object at the specified index.

## Methods

Add(SqlBulkCopyColumnMapping)

Add(SqlBulkCopyColumnMapping)

Adds the specified mapping to the [SqlBulkCopyColumnMappingCollection](#).

Add(Int32, Int32)

Add(Int32, Int32)

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using ordinals to specify both source and destination columns.

```
Add(Int32, String)  
Add(Int32, String)
```

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using an ordinal for the source column and a string for the destination column.

```
Add(String, Int32)  
Add(String, Int32)
```

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using a column name to describe the source column and an ordinal to specify the destination column.

```
Add(String, String)  
Add(String, String)
```

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using column names to specify both source and destination columns.

```
Clear()  
Clear()
```

Clears the contents of the collection.

```
Contains(SqlBulkCopyColumnMapping)  
Contains(SqlBulkCopyColumnMapping)
```

Gets a value indicating whether a specified [SqlBulkCopyColumnMapping](#) object exists in the collection.

```
CopyTo(SqlBulkCopyColumnMapping[], Int32)  
CopyTo(SqlBulkCopyColumnMapping[], Int32)
```

Copies the elements of the [SqlBulkCopyColumnMappingCollection](#) to an array of [SqlBulkCopyColumnMapping](#) items, starting at a particular index.

```
GetEnumerator()  
GetEnumerator()
```

```
IndexOf(SqlBulkCopyColumnMapping)  
IndexOf(SqlBulkCopyColumnMapping)
```

Gets the index of the specified [SqlBulkCopyColumnMapping](#) object.

```
Insert(Int32, SqlBulkCopyColumnMapping)
```

```
Insert(Int32, SqlBulkCopyColumnMapping)
```

Insert a new [SqlBulkCopyColumnMapping](#) at the index specified.

```
Remove(SqlBulkCopyColumnMapping)
```

```
Remove(SqlBulkCopyColumnMapping)
```

Removes the specified [SqlBulkCopyColumnMapping](#) element from the [SqlBulkCopyColumnMappingCollection](#).

```
RemoveAt(Int32)
```

```
RemoveAt(Int32)
```

Removes the mapping at the specified index from the collection.

```
ICollection.CopyTo(Array, Int32)
```

```
ICollection.CopyTo(Array, Int32)
```

```
ICollection.IsSynchronized
```

```
ICollection.IsSynchronized
```

```
ICollection.SyncRoot
```

```
ICollection.SyncRoot
```

```
IList.Add(Object)
```

```
IList.Add(Object)
```

```
IList.Contains(Object)
```

```
IList.Contains(Object)
```

```
IList.IndexOf(Object)
```

```
IList.IndexOf(Object)
```

```
IList.Insert(Int32, Object)
```

```
IList.Insert(Int32, Object)
```

```
IListFixedSize
```

```
IListFixedSize
```

`IList.IsReadOnly`

`IList.IsReadOnly`

`IList.Item[Int32]`

`IList.Item[Int32]`

`IList.Remove(Object)`

`IList.Remove(Object)`

## See Also

# SqlBulkCopyColumnMappingCollection.Add SqlBulkCopyColumnMappingCollection.Add

In this Article

## Overloads

|   |   |
|---|---|
| Add(SqlBulkCopyColumnMapping) Add(SqlBulkCopyColumnMapping) | Adds the specified mapping to the <a href="#">SqlBulkCopyColumnMappingCollection</a> .  |
| Add(Int32, Int32) Add(Int32, Int32)                         | Creates a new <a href="#">SqlBulkCopyColumnMapping</a> and adds it to the collection, using ordinals to specify both source and destination columns.                                      |
| Add(Int32, String) Add(Int32, String)                       | Creates a new <a href="#">SqlBulkCopyColumnMapping</a> and adds it to the collection, using an ordinal for the source column and a string for the destination column.                     |
| Add(String, Int32) Add(String, Int32)                       | Creates a new <a href="#">SqlBulkCopyColumnMapping</a> and adds it to the collection, using a column name to describe the source column and an ordinal to specify the destination column. |
| Add(String, String) Add(String, String)                     | Creates a new <a href="#">SqlBulkCopyColumnMapping</a> and adds it to the collection, using column names to specify both source and destination columns.                                  |

## Add(SqlBulkCopyColumnMapping) Add(SqlBulkCopyColumnMapping)

Adds the specified mapping to the [SqlBulkCopyColumnMappingCollection](#).

```
public System.Data.SqlClient.SqlBulkCopyColumnMapping Add  
(System.Data.SqlClient.SqlBulkCopyColumnMapping bulkCopyColumnMapping);  
  
member this.Add : System.Data.SqlClient.SqlBulkCopyColumnMapping ->  
System.Data.SqlClient.SqlBulkCopyColumnMapping
```

Parameters

bulkCopyColumnMapping

[SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

The [SqlBulkCopyColumnMapping](#) object that describes the mapping to be added to the collection.

Returns

[SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

A [SqlBulkCopyColumnMapping](#) object.

Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. **SqlBulkCopyColumnMapping** objects are used to create a column map for the bulk copy.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoDifferentColumns",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Set up the bulk copy object.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoDifferentColumns";

                // Set up the column mappings by name.
                SqlBulkCopyColumnMapping mapID =
                    new SqlBulkCopyColumnMapping("ProductID", "ProdID");
                bulkCopy.ColumnMappings.Add(mapID);

                SqlBulkCopyColumnMapping mapName =
                    new SqlBulkCopyColumnMapping("Name", "ProdName");
                bulkCopy.ColumnMappings.Add(mapName);

                SqlBulkCopyColumnMapping mapNumber =
                    new SqlBulkCopyColumnMapping("ProductNumber", "ProdNum");
                bulkCopy.ColumnMappings.Add(mapNumber);

                // Write from the source to the destination.
                try
                {
                    bulkCopy.BulkCopy...
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
}
```

```

        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## Add(Int32, Int32) Add(Int32, Int32)

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using ordinals to specify both source and destination columns.

```

public System.Data.SqlClient.SqlBulkCopyColumnMapping Add (int sourceColumnIndex, int
destinationColumnIndex);

member this.Add : int * int -> System.Data.SqlClient.SqlBulkCopyColumnMapping

```

Parameters

sourceColumnIndex

[Int32](#) [Int32](#)

The ordinal position of the source column within the data source.

destinationColumnIndex

[Int32](#) [Int32](#)

The ordinal position of the destination column within the destination table.

Returns

[SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

A column mapping.

## Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy using the ordinal position of the source and destination columns.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoDifferentColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Set up the bulk copy object.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoDifferentColumns";

                // The column order in the source doesn't match the order
                // in the destination, so ColumnMappings must be defined.
                bulkCopy.ColumnMappings.Add(0, 0);
                bulkCopy.ColumnMappings.Add(1, 2);
                bulkCopy.ColumnMappings.Add(2, 1);

                // Write from the source to the destination.
                try
                {
                    bulkCopy.WriteToServer(reader);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
    }
}
```

```

        }

        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

Mappings in a collection must be uniform: either all integer/integer pairs, all string/string pairs, all integer/string pairs, or all string/integer pairs. If you try to add a mapping that is different from others already in the collection, an [InvalidOperationException](#) is thrown.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## Add(Int32, String) Add(Int32, String)

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using an ordinal for the source column and a string for the destination column.

```

public System.Data.SqlClient.SqlBulkCopyColumnMapping Add (int sourceColumnIndex, string
destinationColumn);

member this.Add : int * string -> System.Data.SqlClient.SqlBulkCopyColumnMapping

```

### Parameters

sourceColumnIndex

[Int32](#) [Int32](#)

The ordinal position of the source column within the data source.

destinationColumn

[String](#) [String](#)

The name of the destination column within the destination table.

### Returns

[SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

A column mapping.

## Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoDifferentColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Set up the bulk copy object.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoDifferentColumns";

                // The column order in the source doesn't match the order
                // in the destination, so ColumnMappings must be defined.
                bulkCopy.ColumnMappings.Add(0, "ProdID");
                bulkCopy.ColumnMappings.Add(1, "ProdName");
                bulkCopy.ColumnMappings.Add(2, "ProdNum");

                // Write from the source to the destination.
                try
                {
                    bulkCopy.WriteToServer(reader);
                }
                catch (Exception ex)
                {

```

```

        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

Mappings in a collection must be uniform: either all integer/integer pairs, all string/string pairs, all integer/string pairs, or all string/integer pairs. If you try to add a mapping that is different from others already in the collection, an [InvalidOperationException](#) is thrown.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## Add(String, Int32) Add(String, Int32)

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using a column name to describe the source column and an ordinal to specify the destination column.

```

public System.Data.SqlClient.SqlBulkCopyColumnMapping Add (string sourceColumn, int
destinationColumnIndex);

member this.Add : string * int -> System.Data.SqlClient.SqlBulkCopyColumnMapping

```

### Parameters

sourceColumn

[String](#)

The name of the source column within the data source.

destinationColumnIndex

[Int32](#)

The ordinal position of the destination column within the destination table.

### Returns

## [SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

A column mapping.

### Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. [SqlBulkCopyColumnMapping](#) objects are used to create a column map for the bulk copy.

 **Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using [SqlBulkCopy](#) only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoDifferentColumns",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Set up the bulk copy object.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoDifferentColumns";

                // The column order in the source doesn't match the order
                // in the destination, so ColumnMappings must be defined.
                bulkCopy.ColumnMappings.Add("ProductID", 0);
                bulkCopy.ColumnMappings.Add("ProductNumber", 1);
                bulkCopy.ColumnMappings.Add("Name", 2);

                // Write from the source to the destination.
                try
                {
                    bulkCopy.WriteToServer(reader);
                }
            }
        }
    }
}
```

```

        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            // Close the SqlDataReader. The SqlBulkCopy
            // object is automatically closed at the end
            // of the using block.
            reader.Close();
        }
    }

    // Perform a final count on the destination
    // table to see how many rows were added.
    long countEnd = System.Convert.ToInt32(
        commandRowCount.ExecuteScalar());
    Console.WriteLine("Ending row count = {0}", countEnd);
    Console.WriteLine("{0} rows were added.", countEnd - countStart);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}
}

private static string GetConnectionString()
{
    // To avoid storing the sourceConnection string in your code,
    // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}

```

## Remarks

Mappings in a collection must be uniform: either all integer/integer pairs, all string/string pairs, all integer/string pairs, or all string/integer pairs. If you try to add a mapping that is different from others already in the collection, an [InvalidOperationException](#) is thrown.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

## Add(String, String) Add(String, String)

Creates a new [SqlBulkCopyColumnMapping](#) and adds it to the collection, using column names to specify both source and destination columns.

```

public System.Data.SqlClient.SqlBulkCopyColumnMapping Add (string sourceColumn, string
destinationColumn);

member this.Add : string * string -> System.Data.SqlClient.SqlBulkCopyColumnMapping

```

### Parameters

sourceColumn

[String](#) [String](#)

The name of the source column within the data source.

destinationColumn

[String](#) [String](#)

The name of the destination column within the destination table.

Returns

## SqlBulkCopyColumnMapping SqlBulkCopyColumnMapping

A column mapping.

### Examples

The following example bulk copies data from a source table in the **AdventureWorks** sample database to a destination table in the same database. Although the number of columns in the destination matches the number of columns in the source, the column names and ordinal positions do not match. The code creates a [SqlBulkCopyColumnMapping](#) object by specifying the column names.

#### Important

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string connectionString = GetConnectionString();
        // Open a sourceConnection to the AdventureWorks database.
        using (SqlConnection sourceConnection =
            new SqlConnection(connectionString))
        {
            sourceConnection.Open();

            // Perform an initial count on the destination table.
            SqlCommand commandRowCount = new SqlCommand(
                "SELECT COUNT(*) FROM " +
                "dbo.BulkCopyDemoDifferentColumns;",
                sourceConnection);
            long countStart = System.Convert.ToInt32(
                commandRowCount.ExecuteScalar());
            Console.WriteLine("Starting row count = {0}", countStart);

            // Get data from the source table as a SqlDataReader.
            SqlCommand commandSourceData = new SqlCommand(
                "SELECT ProductID, Name, " +
                "ProductNumber " +
                "FROM Production.Product;", sourceConnection);
            SqlDataReader reader =
                commandSourceData.ExecuteReader();

            // Set up the bulk copy object.
            using (SqlBulkCopy bulkCopy =
                new SqlBulkCopy(connectionString))
            {
                bulkCopy.DestinationTableName =
                    "dbo.BulkCopyDemoDifferentColumns";

                // The column order in the source doesn't match the order
                // in the destination, so ColumnMappings must be defined.
                bulkCopy.ColumnMappings.Add("ProductID", "ProdID");
                bulkCopy.ColumnMappings.Add("Name", "ProdName");
                bulkCopy.ColumnMappings.Add("ProductNumber", "ProdNum");

                // Write from the source to the destination.
                try
                {

```

```

        bulkCopy.WriteToServer(reader);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        // Close the SqlDataReader. The SqlBulkCopy
        // object is automatically closed at the end
        // of the using block.
        reader.Close();
    }
}

// Perform a final count on the destination
// table to see how many rows were added.
long countEnd = System.Convert.ToInt32(
    commandRowCount.ExecuteScalar());
Console.WriteLine("Ending row count = {0}", countEnd);
Console.WriteLine("{0} rows were added.", countEnd - countStart);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}
}

private static string GetConnectionString()
    // To avoid storing the sourceConnectionString string in your code,
    // you can retrieve it from a configuration file.
{
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

Mappings in a collection must be uniform: either all integer/integer pairs, all string/string pairs, all integer/string pairs, or all string/integer pairs. If you try to add a mapping that is different from others already in the collection, an [InvalidOperationException](#) is thrown.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.Clear SqlBulkCopyColumnMappingCollection.Clear

## In this Article

Clears the contents of the collection.

```
public void Clear ();  
member this.Clear : unit -> unit
```

## Examples

The following example performs two bulk copy operations. The first operation copies sales order header information, and the second copies sales order details. Although not strictly necessary in this example (because the ordinal positions of the source and destination columns do match), the example defines column mappings for each bulk copy operation. The [Clear](#) method must be used after the first bulk copy is performed and before the next bulk copy's column mappings are defined.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a connection to the AdventureWorks database.  
        using (SqlConnection connection =  
            new SqlConnection(connectionString))  
        {  
            connection.Open();  
  
            // Empty the destination tables.  
            SqlCommand deleteHeader = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            deleteHeader.ExecuteNonQuery();  
            SqlCommand deleteDetail = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderDetail;",  
                connection);  
            deleteDetail.ExecuteNonQuery();  
  
            // Perform an initial count on the destination  
            // table with matching columns.  
            SqlCommand countRowHeader = new SqlCommand(  
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            long countStartHeader = System.Convert.ToInt32(  
                countRowHeader.ExecuteScalar());  
            Console.WriteLine(  
                "Starting row count for Header table = {0}",  
                countStartHeader);  
  
            // Perform an initial count on the destination  
            // table with different column positions.  
            SqlCommand countRowDetail = new SqlCommand(  
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;",  
                connection);  
            long countStartDetail = System.Convert.ToInt32(  
                countRowDetail.ExecuteScalar());  
            Console.WriteLine(  
                "Starting row count for Detail table = {0}",  
                countStartDetail);  
        }  
    }  
}
```

```

SqlCommand countRowDetail = new SqlCommand(
    "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;",
    connection);
long countStartDetail = System.Convert.ToInt32(
    countRowDetail.ExecuteScalar());
Console.WriteLine(
    "Starting row count for Detail table = {0}",
    countStartDetail);

// Get data from the source table as a SqlDataReader.
// The Sales.SalesOrderHeader and Sales.SalesOrderDetail
// tables are quite large and could easily cause a timeout
// if all data from the tables is added to the destination.
// To keep the example simple and quick, a parameter is
// used to select only orders for a particular account
// as the source for the bulk insert.
SqlCommand headerData = new SqlCommand(
    "SELECT [SalesOrderID], [OrderDate], " +
    "[AccountNumber] FROM [Sales].[SalesOrderHeader] " +
    "WHERE [AccountNumber] = @accountNumber;",
    connection);
SqlParameter parameterAccount = new SqlParameter();
parameterAccount.ParameterName = "@accountNumber";
parameterAccount.SqlDbType = SqlDbType.NVarChar;
parameterAccount.Direction = ParameterDirection.Input;
parameterAccount.Value = "10-4020-000034";
headerData.Parameters.Add(parameterAccount);
SqlDataReader readerHeader = headerData.ExecuteReader();

// Get the Detail data in a separate connection.
using (SqlConnection connection2 = new SqlConnection(connectionString))
{
    connection2.Open();
    SqlCommand sourceDetailData = new SqlCommand(
        "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], [SalesOrderDetailID], " +
        "[OrderQty], [ProductID], [UnitPrice] FROM [Sales].[SalesOrderDetail] " +
        "INNER JOIN [Sales].[SalesOrderHeader] ON [Sales].[SalesOrderDetail]." +
        "[SalesOrderID] = [Sales].[SalesOrderHeader].[SalesOrderID] " +
        "WHERE [AccountNumber] = @accountNumber;", connection2);

    SqlParameter accountDetail = new SqlParameter();
    accountDetail.ParameterName = "@accountNumber";
    accountDetail.SqlDbType = SqlDbType.NVarChar;
    accountDetail.Direction = ParameterDirection.Input;
    accountDetail.Value = "10-4020-000034";
    sourceDetailData.Parameters.Add(accountDetail);
    SqlDataReader readerDetail = sourceDetailData.ExecuteReader();

    // Create the SqlBulkCopy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoOrderHeader";

        // Guarantee that columns are mapped correctly by
        // defining the column mappings for the order.
        bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID");
        bulkCopy.ColumnMappings.Add("OrderDate", "OrderDate");
        bulkCopy.ColumnMappings.Add("AccountNumber", "AccountNumber");

        // Write readerHeader to the destination.
        try
        {
            bulkCopy.WriteToServer(readerHeader);
        }
    }
}

```

```

        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    finally
    {
        readerHeader.Close();
    }

    // Set up the order details destination.
    bulkCopy.DestinationTableName = "dbo.BulkCopyDemoOrderDetail";

    // Clear the ColumnMappingCollection.
    bulkCopy.ColumnMappings.Clear();

    // Add order detail column mappings.
    bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID");
    bulkCopy.ColumnMappings.Add("SalesOrderDetailID", "SalesOrderDetailID");
    bulkCopy.ColumnMappings.Add("OrderQty", "OrderQty");
    bulkCopy.ColumnMappings.Add("ProductID", "ProductID");
    bulkCopy.ColumnMappings.Add("UnitPrice", "UnitPrice");

    // Write readerDetail to the destination.
    try
    {
        bulkCopy.WriteToServer(readerDetail);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    finally
    {
        readerDetail.Close();
    }
}

// Perform a final count on the destination
// tables to see how many rows were added.
long countEndHeader = System.Convert.ToInt32(
    countRowHeader.ExecuteScalar());
Console.WriteLine("{0} rows were added to the Header table.",
    countEndHeader - countStartHeader);
long countEndDetail = System.Convert.ToInt32(
    countRowDetail.ExecuteScalar());
Console.WriteLine("{0} rows were added to the Detail table.",
    countEndDetail - countStartDetail);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}

}

private static string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
    return "Data Source=(local); " +
        " Integrated Security=true;" +
        "Initial Catalog=AdventureWorks;";
}
}

```

## Remarks

The [Clear](#) method is most commonly used when you use a single [SqlBulkCopy](#) instance to process more than one bulk copy operation. If you create column mappings for one bulk copy operation, you must clear the [SqlBulkCopyColumnMappingCollection](#) after the [WriteToServer](#) method and before processing the next bulk copy.

Performing several bulk copies using the same [SqlBulkCopy](#) instance will usually be more efficient from a performance point of view than using a separate [SqlBulkCopy](#) for each operation.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.Contains SqlBulkCopyColumnMappingCollection.Contains

## In this Article

Gets a value indicating whether a specified [SqlBulkCopyColumnMapping](#) object exists in the collection.

```
public bool Contains (System.Data.SqlClient.SqlBulkCopyColumnMapping value);  
member this.Contains : System.Data.SqlClient.SqlBulkCopyColumnMapping -> bool
```

### Parameters

value [SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

A valid [SqlBulkCopyColumnMapping](#) object.

### Returns

[Boolean](#) [Boolean](#)

`true` if the specified mapping exists in the collection; otherwise `false`.

### See

[Performing Bulk Copy Operations](#)

### Also

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.CopyTo SqlBulkCopyColumnMappingCollection.CopyTo

## In this Article

Copies the elements of the [SqlBulkCopyColumnMappingCollection](#) to an array of [SqlBulkCopyColumnMapping](#) items, starting at a particular index.

```
public void CopyTo (System.Data.SqlClient.SqlBulkCopyColumnMapping[] array, int index);  
member this.CopyTo : System.Data.SqlClient.SqlBulkCopyColumnMapping[] * int -> unit
```

## Parameters

array [SqlBulkCopyColumnMapping\[\]](#)

The one-dimensional [SqlBulkCopyColumnMapping](#) array that is the destination of the elements copied from [SqlBulkCopyColumnMappingCollection](#). The array must have zero-based indexing.

index [Int32](#) [Int32](#)

The zero-based index in `array` at which copying begins.

See [Performing Bulk Copy Operations](#)

Also [ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.Count SqlBulkCopyColumnMappingCollection.Count

## In this Article

```
public int Count { get; }  
member this.Count : int
```

## Returns

[Int32](#) [Int32](#)

# SqlBulkCopyColumnMappingCollection.GetEnumerator

## SqlBulkCopyColumnMappingCollection.GetEnumerator

### In this Article

```
public System.Collections.IEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

### Returns

[IEnumerator](#) [IEnumerator](#)

# SqlBulkCopyColumnMappingCollection.ICollection.CopyTo

## In this Article

```
void ICollection.CopyTo (Array array, int index);
```

## Parameters

|       |       |
|-------|-------|
| array | Array |
|-------|-------|

|       |       |
|-------|-------|
| index | Int32 |
|-------|-------|

# SqlBulkCopyColumnMappingCollection.ICollection.IsSynchronized

In this Article

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Returns

[Boolean](#)

# SqlBulkCopyColumnMappingCollection.ICollection.SyncRoot

## In this Article

```
object System.Collections.ICollection.SyncRoot { get; }
```

## Returns

[Object](#)

# SqlBulkCopyColumnMappingCollection\_IList.Add

## In this Article

```
int IList.Add (object value);
```

### Parameters

|       |        |
|-------|--------|
| value | Object |
|-------|--------|

### Returns

[Int32](#)

# SqlBulkCopyColumnMappingCollection.ICollection.Contains

## In this Article

```
bool ICollection.Contains (object value);
```

### Parameters

|       |        |
|-------|--------|
| value | Object |
|-------|--------|

### Returns

[Boolean](#)

# SqlBulkCopyColumnMappingCollection\_IList.IndexOf

## In this Article

```
int IList.IndexOf (object value);
```

### Parameters

value Object

### Returns

[Int32](#)

# SqlBulkCopyColumnMappingCollection\_IList.Insert

## In this Article

```
void IList.Insert (int index, object value);
```

### Parameters

|       |                        |
|-------|------------------------|
| index | <a href="#">Int32</a>  |
| value | <a href="#">Object</a> |

# SqlBulkCopyColumnMappingCollection\_IList.IsFixedSize

## In this Article

```
bool System.Collections.IList.IsFixedSize { get; }
```

### Returns

[Boolean](#)

# SqlBulkCopyColumnMappingCollection\_IList.IsReadOnly

## In this Article

```
bool System.Collections.IList.IsReadOnly { get; }
```

### Returns

[Boolean](#)

# SqlBulkCopyColumnMappingCollection\_IList.Item[Int32]

## In this Article

```
object System.Collections.IList.Item[int index] { get; set; }
```

### Parameters

index Int32

### Returns

[Object](#)

# SqlBulkCopyColumnMappingCollection\_IList.Remove

## In this Article

```
void IList.Remove (object value);
```

## Parameters

|       |                        |
|-------|------------------------|
| value | <a href="#">Object</a> |
|-------|------------------------|

# SqlBulkCopyColumnMappingCollection.IndexOf SqlBulkCopyColumnMappingCollection.IndexOf

## In this Article

Gets the index of the specified [SqlBulkCopyColumnMapping](#) object.

```
public int IndexOf (System.Data.SqlClient.SqlBulkCopyColumnMapping value);
```

```
member this.IndexOf : System.Data.SqlClient.SqlBulkCopyColumnMapping -> int
```

### Parameters

value

[SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

The [SqlBulkCopyColumnMapping](#) object for which to search.

### Returns

[Int32](#) [Int32](#)

The zero-based index of the column mapping, or -1 if the column mapping is not found in the collection.

### See

[Performing Bulk Copy Operations](#)

### Also

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.Insert SqlBulkCopyColumnMappingCollection.Insert

## In this Article

Insert a new [SqlBulkCopyColumnMapping](#) at the index specified.

```
public void Insert (int index, System.Data.SqlClient.SqlBulkCopyColumnMapping value);  
member this.Insert : int * System.Data.SqlClient.SqlBulkCopyColumnMapping -> unit
```

## Parameters

index [Int32](#) [Int32](#)

Integer value of the location within the [SqlBulkCopyColumnMappingCollection](#) at which to insert the new [SqlBulkCopyColumnMapping](#).

value [SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

[SqlBulkCopyColumnMapping](#) object to be inserted in the collection.

See [Performing Bulk Copy Operations](#)

Also [ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.Item[Int32] Sql BulkCopyColumnMappingCollection.Item[Int32]

## In this Article

Gets the [SqlBulkCopyColumnMapping](#) object at the specified index.

```
public System.Data.SqlClient.SqlBulkCopyColumnMapping this[int index] { get; }  
member this.Item(int) : System.Data.SqlClient.SqlBulkCopyColumnMapping
```

## Parameters

index [Int32](#) [Int32](#)

The zero-based index of the [SqlBulkCopyColumnMapping](#) to find.

## Returns

[SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

A [SqlBulkCopyColumnMapping](#) object.

## See

[Performing Bulk Copy Operations](#)

## Also

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.Remove SqlBulkCopyColumnMappingCollection.Remove

## In this Article

Removes the specified [SqlBulkCopyColumnMapping](#) element from the [SqlBulkCopyColumnMappingCollection](#).

```
public void Remove (System.Data.SqlClient.SqlBulkCopyColumnMapping value);  
member this.Remove : System.Data.SqlClient.SqlBulkCopyColumnMapping -> unit
```

## Parameters

value [SqlBulkCopyColumnMapping](#) [SqlBulkCopyColumnMapping](#)

[SqlBulkCopyColumnMapping](#) object to be removed from the collection.

## Examples

The following example performs two bulk copy operations. The first operation copies sales order header information, and the second copies sales order details. Although not strictly necessary in this example (because the ordinal positions of the source and destination columns do match), the example defines column mappings for each bulk copy operation. Both bulk copies include a mapping for the **SalesOrderID**, so rather than clearing the entire collection between bulk copy operations, the example removes all mappings except for the **SalesOrderID** mapping and then adds the appropriate mappings for the second bulk copy operation.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a connection to the AdventureWorks database.  
        using (SqlConnection connection =  
            new SqlConnection(connectionString))  
        {  
            connection.Open();  
  
            // Empty the destination tables.  
            SqlCommand deleteHeader = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            deleteHeader.ExecuteNonQuery();  
            SqlCommand deleteDetail = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderDetail;",  
                connection);  
            deleteDetail.ExecuteNonQuery();  
  
            // Perform an initial count on the destination  
            // table with matching columns.  
            SqlCommand countRowHeader = new SqlCommand(  
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            long countStartHeader = System.Convert.ToInt32(
```

```

        countRowHeader.ExecuteScalar());
Console.WriteLine(
    "Starting row count for Header table = {0}",
    countStartHeader);

// Perform an initial count on the destination
// table with different column positions.
SqlCommand countRowDetail = new SqlCommand(
    "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;",
    connection);
long countStartDetail = System.Convert.ToInt32(
    countRowDetail.ExecuteScalar());
Console.WriteLine(
    "Starting row count for Detail table = {0}",
    countStartDetail);

// Get data from the source table as a SqlDataReader.
// The Sales.SalesOrderHeader and Sales.SalesOrderDetail
// tables are quite large and could easily cause a timeout
// if all data from the tables is added to the destination.
// To keep the example simple and quick, a parameter is
// used to select only orders for a particular account
// as the source for the bulk insert.
SqlCommand headerData = new SqlCommand(
    "SELECT [SalesOrderID], [OrderDate], " +
    "[AccountNumber] FROM [Sales].[SalesOrderHeader] " +
    "WHERE [AccountNumber] = @accountNumber;",
    connection);
SqlParameter parameterAccount = new SqlParameter();
parameterAccount.ParameterName = "@accountNumber";
parameterAccount.SqlDbType = SqlDbType.NVarChar;
parameterAccount.Direction = ParameterDirection.Input;
parameterAccount.Value = "10-4020-000034";
headerData.Parameters.Add(parameterAccount);
SqlDataReader readerHeader = headerData.ExecuteReader();

// Get the Detail data in a separate connection.
using (SqlConnection connection2 = new SqlConnection(connectionString))
{
    connection2.Open();
    SqlCommand sourceDetailData = new SqlCommand(
        "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], [SalesOrderDetailID], " +
        "[OrderQty], [ProductID], [UnitPrice] FROM [Sales].[SalesOrderDetail] " +
        "INNER JOIN [Sales].[SalesOrderHeader] ON [Sales].[SalesOrderDetail]." +
        "[SalesOrderID] = [Sales].[SalesOrderHeader].[SalesOrderID] " +
        "WHERE [AccountNumber] = @accountNumber;", connection2);

    SqlParameter accountDetail = new SqlParameter();
    accountDetail.ParameterName = "@accountNumber";
    accountDetail.SqlDbType = SqlDbType.NVarChar;
    accountDetail.Direction = ParameterDirection.Input;
    accountDetail.Value = "10-4020-000034";
    sourceDetailData.Parameters.Add(accountDetail);
    SqlDataReader readerDetail = sourceDetailData.ExecuteReader();

    // Create the SqlBulkCopy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoOrderHeader";

        // Guarantee that columns are mapped correctly by
        // defining the column mappings for the order.
        bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID");
    }
}

```

```

// Define SqlBulkCopyColumnMapping objects so they
// can be referenced later.
SqlBulkCopyColumnMapping columnMappingDate =
    bulkCopy.ColumnMappings.Add("OrderDate", "OrderDate");
SqlBulkCopyColumnMapping columnMappingAccount =
    bulkCopy.ColumnMappings.Add("AccountNumber", "AccountNumber");

// Write readerHeader to the destination.
try
{
    bulkCopy.WriteToServer(readerHeader);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
finally
{
    readerHeader.Close();
}

// Set up the order details destination.
bulkCopy.DestinationTableName = "dbo.BulkCopyDemoOrderDetail";

// Rather than clearing mappings that are not necessary
// for the next bulk copy operation, the unneeded mappings
// are removed with the Remove method.
bulkCopy.ColumnMappings.Remove(columnMappingDate);
bulkCopy.ColumnMappings.Remove(columnMappingAccount);

// Add order detail column mappings.
bulkCopy.ColumnMappings.Add("SalesOrderDetailID", "SalesOrderDetailID");
bulkCopy.ColumnMappings.Add("OrderQty", "OrderQty");
bulkCopy.ColumnMappings.Add("ProductID", "ProductID");
bulkCopy.ColumnMappings.Add("UnitPrice", "UnitPrice");
bulkCopy.WriteToServer(readerDetail);

// Write readerDetail to the destination.
try
{
    bulkCopy.WriteToServer(readerDetail);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
finally
{
    readerDetail.Close();
}

// Perform a final count on the destination
// tables to see how many rows were added.
long countEndHeader = System.Convert.ToInt32(
    countRowHeader.ExecuteScalar());
Console.WriteLine("{0} rows were added to the Header table.",
    countEndHeader - countStartHeader);
long countEndDetail = System.Convert.ToInt32(
    countRowDetail.ExecuteScalar());
Console.WriteLine("{0} rows were added to the Detail table.",
    countEndDetail - countStartDetail);
Console.WriteLine("Press Enter to finish.");
Console.ReadLine();
}

```

```
        }

    private static string GetConnectionString()
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
    {
        return "Data Source=(local); " +
            " Integrated Security=true;" +
            "Initial Catalog=AdventureWorks;";
    }
}
```

## Remarks

The [Remove](#) method is most commonly used when you use a single [SqlBulkCopy](#) instance to process more than one bulk copy operation. If you create column mappings for one bulk copy operation, you must remove mappings that no longer apply after the [WriteToServer](#) method is called and before defining mapping for the next bulk copy. You can clear the entire collection by using the [Clear](#) method, or remove mappings individually using the [Remove](#) method or the [RemoveAt](#) method.

Performing several bulk copies using the same [SqlBulkCopy](#) instance will usually be more efficient from a performance point of view than using a separate [SqlBulkCopy](#) for each operation.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopyColumnMappingCollection.RemoveAt SqlBulkCopyColumnMappingCollection.RemoveAt

## In this Article

Removes the mapping at the specified index from the collection.

```
public void RemoveAt (int index);  
member this.RemoveAt : int -> unit
```

## Parameters

index Int32 Int32

The zero-based index of the [SqlBulkCopyColumnMapping](#) object to be removed from the collection.

## Examples

The following example performs two bulk copy operations. The first operation copies sales order header information, and the second copies sales order details. Although not strictly necessary in this example (because the ordinal positions of the source and destination columns do match), the example defines column mappings for each bulk copy operation. Both bulk copies include a mapping for the **SalesOrderID**, so rather than clearing the entire collection between bulk copy operations, the example removes all mappings except for the **SalesOrderID** mapping and then adds the appropriate mappings for the second bulk copy operation.

**Important**

This sample will not run unless you have created the work tables as described in [Bulk Copy Example Setup](#). This code is provided to demonstrate the syntax for using **SqlBulkCopy** only. If the source and destination tables are in the same SQL Server instance, it is easier and faster to use a Transact-SQL `INSERT ... SELECT` statement to copy the data.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        string connectionString = GetConnectionString();  
        // Open a connection to the AdventureWorks database.  
        using (SqlConnection connection =  
            new SqlConnection(connectionString))  
        {  
            connection.Open();  
  
            // Empty the destination tables.  
            SqlCommand deleteHeader = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            deleteHeader.ExecuteNonQuery();  
            SqlCommand deleteDetail = new SqlCommand(  
                "DELETE FROM dbo.BulkCopyDemoOrderDetail;",  
                connection);  
            deleteDetail.ExecuteNonQuery();  
  
            // Perform an initial count on the destination  
            // table with matching columns.  
            SqlCommand countRowHeader = new SqlCommand(  
                "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderHeader;",  
                connection);  
            long countStartHeader = System.Convert.ToInt32(
```

```

        countRowHeader.ExecuteScalar());
Console.WriteLine(
    "Starting row count for Header table = {0}",
    countStartHeader);

// Perform an initial count on the destination
// table with different column positions.
SqlCommand countRowDetail = new SqlCommand(
    "SELECT COUNT(*) FROM dbo.BulkCopyDemoOrderDetail;",
    connection);
long countStartDetail = System.Convert.ToInt32(
    countRowDetail.ExecuteScalar());
Console.WriteLine(
    "Starting row count for Detail table = {0}",
    countStartDetail);

// Get data from the source table as a SqlDataReader.
// The Sales.SalesOrderHeader and Sales.SalesOrderDetail
// tables are quite large and could easily cause a timeout
// if all data from the tables is added to the destination.
// To keep the example simple and quick, a parameter is
// used to select only orders for a particular account
// as the source for the bulk insert.
SqlCommand headerData = new SqlCommand(
    "SELECT [SalesOrderID], [OrderDate], " +
    "[AccountNumber] FROM [Sales].[SalesOrderHeader] " +
    "WHERE [AccountNumber] = @accountNumber;",
    connection);
SqlParameter parameterAccount = new SqlParameter();
parameterAccount.ParameterName = "@accountNumber";
parameterAccount.SqlDbType = SqlDbType.NVarChar;
parameterAccount.Direction = ParameterDirection.Input;
parameterAccount.Value = "10-4020-000034";
headerData.Parameters.Add(parameterAccount);
SqlDataReader readerHeader = headerData.ExecuteReader();

// Get the Detail data in a separate connection.
using (SqlConnection connection2 = new SqlConnection(connectionString))
{
    connection2.Open();
    SqlCommand sourceDetailData = new SqlCommand(
        "SELECT [Sales].[SalesOrderDetail].[SalesOrderID], [SalesOrderDetailID], " +
        "[OrderQty], [ProductID], [UnitPrice] FROM [Sales].[SalesOrderDetail] " +
        "INNER JOIN [Sales].[SalesOrderHeader] ON [Sales].[SalesOrderDetail]." +
        "[SalesOrderID] = [Sales].[SalesOrderHeader].[SalesOrderID] " +
        "WHERE [AccountNumber] = @accountNumber;", connection2);

    SqlParameter accountDetail = new SqlParameter();
    accountDetail.ParameterName = "@accountNumber";
    accountDetail.SqlDbType = SqlDbType.NVarChar;
    accountDetail.Direction = ParameterDirection.Input;
    accountDetail.Value = "10-4020-000034";
    sourceDetailData.Parameters.Add(accountDetail);
    SqlDataReader readerDetail = sourceDetailData.ExecuteReader();

    // Create the SqlBulkCopy object.
    using (SqlBulkCopy bulkCopy =
        new SqlBulkCopy(connectionString))
    {
        bulkCopy.DestinationTableName =
            "dbo.BulkCopyDemoOrderHeader";

        // Guarantee that columns are mapped correctly by
        // defining the column mappings for the order.
        bulkCopy.ColumnMappings.Add("SalesOrderID", "SalesOrderID");
        bulkCopy.ColumnMappings.Add("OrderDate", "OrderDate");
    }
}

```

```

        bulkCopy.ColumnMappings.Add("OrderDate", "OrderDate");
        bulkCopy.ColumnMappings.Add("AccountNumber", "AccountNumber");

        // Write readerHeader to the destination.
        try
        {
            bulkCopy.WriteToServer(readerHeader);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            readerHeader.Close();
        }

        // Set up the order details destination.
        bulkCopy.DestinationTableName = "dbo.BulkCopyDemoOrderDetail";

        // Rather than clearing mappings that are not necessary
        // for the next bulk copy operation, the unneeded mappings
        // are removed with the RemoveAt method.
        bulkCopy.ColumnMappings.RemoveAt(2);
        bulkCopy.ColumnMappings.RemoveAt(1);

        // Add order detail column mappings.
        bulkCopy.ColumnMappings.Add("SalesOrderDetailID", "SalesOrderDetailID");
        bulkCopy.ColumnMappings.Add("OrderQty", "OrderQty");
        bulkCopy.ColumnMappings.Add("ProductID", "ProductID");
        bulkCopy.ColumnMappings.Add("UnitPrice", "UnitPrice");
        bulkCopy.WriteToServer(readerDetail);

        // Write readerDetail to the destination.
        try
        {
            bulkCopy.WriteToServer(readerDetail);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
        finally
        {
            readerDetail.Close();
        }
    }

    // Perform a final count on the destination
    // tables to see how many rows were added.
    long countEndHeader = System.Convert.ToInt32(
        countRowHeader.ExecuteScalar());
    Console.WriteLine("{0} rows were added to the Header table.",
        countEndHeader - countStartHeader);
    long countEndDetail = System.Convert.ToInt32(
        countRowDetail.ExecuteScalar());
    Console.WriteLine("{0} rows were added to the Detail table.",
        countEndDetail - countStartDetail);
    Console.WriteLine("Press Enter to finish.");
    Console.ReadLine();
}

private static string GetConnectionString()
    // To avoid storing the connection string in your code,
}

```

```
// you can retrieve it from a configuration file.  
{  
    return "Data Source=(local); " +  
        " Integrated Security=true;" +  
        "Initial Catalog=AdventureWorks;";  
}  
}
```

## Remarks

The [RemoveAt](#) method is most commonly used when you use a single [SqlBulkCopy](#) instance to process more than one bulk copy operation. If you create column mappings for one bulk copy operation, you must remove mappings that no longer apply after the [WriteToServer](#) method is called and before defining mapping for the next bulk copy. You can clear the entire collection by using the [Clear](#) method, or remove mappings individually using the [Remove](#) method or the [RemoveAt](#) method.

Performing several bulk copies using the same [SqlBulkCopy](#) instance will usually be more efficient from a performance point of view than using a separate [SqlBulkCopy](#) for each operation.

See

[Performing Bulk Copy Operations](#)

Also

[ADO.NET Overview](#)

# SqlBulkCopyOptions SqlBulkCopyOptions Enum

Bitwise flag that specifies one or more options to use with an instance of [SqlBulkCopy](#).

## Declaration

```
[System.Flags]  
public enum SqlBulkCopyOptions  
  
type SqlBulkCopyOptions =
```

## Inheritance Hierarchy



## Remarks

You can use the [SqlBulkCopyOptions](#) enumeration when you construct a [SqlBulkCopy](#) instance to change how the [WriteToServer](#) methods for that instance behave.

## Fields

|                                  |
|----------------------------------|
| AllowEncryptedValueModifications |
| AllowEncryptedValueModifications |

When specified, **AllowEncryptedValueModifications** enables bulk copying of encrypted data between tables or databases, without decrypting the data. Typically, an application would select data from encrypted columns from one table without decrypting the data (the app would connect to the database with the column encryption setting keyword set to disabled) and then would use this option to bulk insert the data, which is still encrypted. For more information, see [Always Encrypted](#).

Use caution when specifying **AllowEncryptedValueModifications** as this may lead to corrupting the database because the driver does not check if the data is indeed encrypted, or if it is correctly encrypted using the same encryption type, algorithm and key as the target column.

|                  |
|------------------|
| CheckConstraints |
| CheckConstraints |

Check constraints while data is being inserted. By default, constraints are not checked.

|         |
|---------|
| Default |
| Default |

Use the default values for all options.

|              |
|--------------|
| FireTriggers |
| FireTriggers |

When specified, cause the server to fire the insert triggers for the rows being inserted into the database.

|              |
|--------------|
| KeepIdentity |
| KeepIdentity |

Preserve source identity values. When not specified, identity values are assigned by the destination.

`KeepNulls` `KeepNulls`

Preserve null values in the destination table regardless of the settings for default values. When not specified, null values are replaced by default values where applicable.

`TableLock` `TableLock`

Obtain a bulk update lock for the duration of the bulk copy operation. When not specified, row locks are used.

`UseInternalTransaction`  
`UseInternalTransaction`

When specified, each batch of the bulk-copy operation will occur within a transaction. If you indicate this option and also provide a `SqlTransaction` object to the constructor, an `ArgumentException` occurs.

## See Also

# SqlClientFactory SqlClientFactory Class

Represents a set of methods for creating instances of the [System.Data.SqlClient](#) provider's implementation of the data source classes.

## Declaration

```
public sealed class SqlClientFactory : System.Data.Common.DbProviderFactory  
type SqlClientFactory = class  
    inherit DbProviderFactory
```

## Inheritance Hierarchy

[Object](#) [Object](#)  
[DbProviderFactory](#) [DbProviderFactory](#)

## Fields

Instance

Instance

Gets an instance of the [SqlClientFactory](#). This can be used to retrieve strongly typed data objects.

## Properties

CanCreateDataSourceEnumerator

CanCreateDataSourceEnumerator

Returns `true` if a [SqlDataSourceEnumerator](#) can be created; otherwise `false`.

## Methods

CreateCommand()

CreateCommand()

Returns a strongly typed [DbCommand](#) instance.

CreateCommandBuilder()

CreateCommandBuilder()

Returns a strongly typed [DbCommandBuilder](#) instance.

CreateConnection()

CreateConnection()

Returns a strongly typed [DbConnection](#) instance.

CreateConnectionStringBuilder()

`CreateConnectionStringBuilder()`

Returns a strongly typed [DbConnectionStringBuilder](#) instance.

`CreateDataAdapter()`

`CreateDataAdapter()`

Returns a strongly typed [DbDataAdapter](#) instance.

`CreateDataSourceEnumerator()`

`CreateDataSourceEnumerator()`

Returns a new [SqlDataSourceEnumerator](#).

`CreateParameter()`

`CreateParameter()`

Returns a strongly typed [DbParameter](#) instance.

`CreatePermission(PermissionState)`

`CreatePermission(PermissionState)`

Returns a new [CodeAccessPermission](#).

`IServiceProvider.GetService(Type)`

`IServiceProvider.GetService(Type)`

For a description of this member, see [GetService\(Type\)](#).

## See Also

# SqlClientFactory.CanCreateDataSourceEnumerator SqlClientFactory.CanCreateDataSourceEnumerator

## In this Article

Returns `true` if a [SqlDataSourceEnumerator](#) can be created; otherwise `false`.

```
public override bool CanCreateDataSourceEnumerator { get; }  
member this.CanCreateDataSourceEnumerator : bool
```

Returns

**Boolean Boolean**

`true` if a [SqlDataSourceEnumerator](#) can be created; otherwise `false`.

## Examples

The following example displays a list of all available SQL Server data sources, using code that could enumerate data sources for any provider.

```
using System;  
using System.Data;  
using System.Data.Common;  
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        // List all SQL Server instances:  
        ListServers(SqlClientFactory.Instance);  
  
        Console.WriteLine();  
        Console.WriteLine("Press any key to continue...");  
        Console.ReadKey();  
    }  
    private static void ListServers(DbProviderFactory factory)  
    {  
        // This procedure is provider-agnostic, and can list  
        // instances of any provider's servers. Of course,  
        // not all providers can create a data source enumerator,  
        // so it's best to check the CanCreateDataSourceEnumerator  
        // property before attempting to list the data sources.  
        if (factory.CanCreateDataSourceEnumerator)  
        {  
            DbDataSourceEnumerator instance =  
                factory.CreateDataSourceEnumerator();  
            DataTable table = instance.GetDataSources();  
  
            foreach (DataRow row in table.Rows)  
            {  
                Console.WriteLine("{0}\\{1}",  
                    row["ServerName"], row["InstanceName"]);  
            }  
        }  
    }  
}
```

## Remarks

The [DbProviderFactory](#) class provides the [CanCreateDataSourceEnumerator](#) property so that inheritors can indicate whether they can provide a data source enumerator. The [SqlClientFactory](#) displays this property, but its value is always `true`.

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateCommand SqlClientFactory.CreateCommand

## In this Article

Returns a strongly typed [DbCommand](#) instance.

```
public override System.Data.Common.DbCommand CreateCommand ();  
override this.CreateCommand : unit -> System.Data.Common.DbCommand
```

Returns

[DbCommand](#) [DbCommand](#)

A new strongly typed instance of [DbCommand](#).

## Examples

The following code fragment returns a strongly typed [DbCommand](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;  
DbCommand cmd = newFactory.CreateCommand();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateCommandBuilder SqlClientFactory.CreateCommandBuilder

## In this Article

Returns a strongly typed [DbCommandBuilder](#) instance.

```
public override System.Data.Common.DbCommandBuilder CreateCommandBuilder ();  
override this.CreateCommandBuilder : unit -> System.Data.Common.DbCommandBuilder
```

Returns

[DbCommandBuilder](#) [DbCommandBuilder](#)

A new strongly typed instance of [DbCommandBuilder](#).

## Examples

The following code fragment returns a strongly typed [DbCommandBuilder](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;  
DbCommandBuilder cmd = newFactory.CreateCommandBuilder();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateConnection SqlClientFactory.CreateConnection

## In this Article

Returns a strongly typed [DbConnection](#) instance.

```
public override System.Data.Common.DbConnection CreateConnection ();
override this.CreateConnection : unit -> System.Data.Common.DbConnection
```

Returns

[DbConnection](#) [DbConnection](#)

A new strongly typed instance of [DbConnection](#).

## Examples

The following code fragment returns a strongly typed [DbConnection](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;
DbConnection cmd = newFactory.CreateConnection();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateConnectionStringBuilder SqlClientFactory.CreateConnectionStringBuilder

## In this Article

Returns a strongly typed [DbConnectionStringBuilder](#) instance.

```
public override System.Data.Common.DbConnectionStringBuilder CreateConnectionStringBuilder ();
override this.CreateConnectionStringBuilder : unit -> System.Data.Common.DbConnectionStringBuilder
```

Returns

[DbConnectionStringBuilder](#) [DbConnectionStringBuilder](#)

A new strongly typed instance of [DbConnectionStringBuilder](#).

## Examples

The following code fragment returns a strongly typed [DbConnectionStringBuilder](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;
DbConnectionStringBuilder cmd =
    newFactory.CreateConnectionStringBuilder();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateDataAdapter SqlClientFactory.CreateDataAdapter

## In this Article

Returns a strongly typed [DbDataAdapter](#) instance.

```
public override System.Data.Common.DbDataAdapter CreateDataAdapter ();  
override this.CreateDataAdapter : unit -> System.Data.Common.DbDataAdapter
```

Returns

[DbDataAdapter](#) [DbDataAdapter](#)

A new strongly typed instance of [DbDataAdapter](#).

## Examples

The following code fragment returns a strongly typed [DbDataAdapter](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;  
DbDataAdapter cmd = newFactory.CreateDataAdapter();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateDataSourceEnumerator SqlClientFactory.CreateDataSourceEnumerator

## In this Article

Returns a new [SqlDataSourceEnumerator](#).

```
public override System.Data.Common.DbDataSourceEnumerator CreateDataSourceEnumerator ();  
override this.CreateDataSourceEnumerator : unit -> System.Data.Common.DbDataSourceEnumerator
```

Returns

[DbDataSourceEnumerator](#) [DbDataSourceEnumerator](#)

A new data source enumerator.

## Examples

The following example displays a list of all available SQL Server data sources, using code that could enumerate data sources for any provider.

```
using System;  
using System.Data;  
using System.Data.Common;  
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        // List all SQL Server instances:  
        ListServers(SqlClientFactory.Instance);  
  
        Console.WriteLine();  
        Console.WriteLine("Press any key to continue...");  
        Console.ReadKey();  
    }  
    private static void ListServers(DbProviderFactory factory)  
    {  
        // This procedure is provider-agnostic, and can list  
        // instances of any provider's servers. Of course,  
        // not all providers can create a data source enumerator,  
        // so it's best to check the CanCreateDataSourceEnumerator  
        // property before attempting to list the data sources.  
        if (factory.CanCreateDataSourceEnumerator)  
        {  
            DbDataSourceEnumerator instance =  
                factory.CreateDataSourceEnumerator();  
            DataTable table = instance.GetDataSources();  
  
            foreach (DataRow row in table.Rows)  
            {  
                Console.WriteLine("{0}\\{1}",  
                    row["ServerName"], row["InstanceName"]);  
            }  
        }  
    }  
}
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreateParameter SqlClientFactory.CreateParameter

## In this Article

Returns a strongly typed [DbParameter](#) instance.

```
public override System.Data.Common.DbParameter CreateParameter ();  
override this.CreateParameter : unit -> System.Data.Common.DbParameter
```

Returns

[DbParameter](#) [DbParameter](#)

A new strongly typed instance of [DbParameter](#).

## Examples

The following code fragment returns a strongly typed [DbParameter](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;  
DbParameter cmd = newFactory.CreateParameter();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.CreatePermission SqlClientFactory.CreatePermission

## In this Article

Returns a new [CodeAccessPermission](#).

```
public override System.Security.CodeAccessPermission CreatePermission  
(System.Security.Permissions.PermissionState state);  
  
override this.CreatePermission : System.Security.Permissions.PermissionState ->  
System.Security.CodeAccessPermission
```

## Parameters

state [PermissionState](#) [PermissionState](#)

A member of the [PermissionState](#) enumeration.

## Returns

[CodeAccessPermission](#) [CodeAccessPermission](#)

A strongly typed instance of [CodeAccessPermission](#).

## See

[ADO.NET Overview](#)

## Also

# SqlClientFactory.Instance SqlClientFactory.Instance

## In this Article

Gets an instance of the [SqlClientFactory](#). This can be used to retrieve strongly typed data objects.

```
public static readonly System.Data.SqlClient.SqlClientFactory Instance;  
static val mutable Instance : System.Data.SqlClient.SqlClientFactory
```

Returns

[SqlClientFactory](#) [SqlClientFactory](#)

## Examples

The following code fragment uses the [Instance](#) property to retrieve a **SqlClientFactory** instance, and then return a strongly typed [DbCommand](#) instance:

```
SqlClientFactory newFactory = SqlClientFactory.Instance;  
DbCommand cmd = newFactory.CreateCommand();
```

See

[ADO.NET Overview](#)

Also

# SqlClientFactory.IServiceProvider.GetService

## In this Article

For a description of this member, see [GetService\(Type\)](#).

```
object IServiceProvider.GetService (Type serviceType);
```

### Parameters

|             |      |
|-------------|------|
| serviceType | Type |
|-------------|------|

An object that specifies the type of service object to get.

### Returns

[Object](#)

A service object.

### See

[ADO.NET Overview](#)

### Also

# SqlClientLogger SqlClientLogger Class

Represents a SQL client logger.

## Declaration

```
public class SqlClientLogger  
type SqlClientLogger = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

`SqlClientLogger()`

`SqlClientLogger()`

Initializes a new instance of the [SqlClientLogger](#) class.

## Properties

`IsLoggingEnabled`

`IsLoggingEnabled`

Gets a value that indicates whether bid tracing is enabled.

## Methods

`LogAssert(Boolean, String, String, String)`

`LogAssert(Boolean, String, String, String)`

Logs the specified message if `value` is `false`.

`.LogError(String, String, String)`

`.LogError(String, String, String)`

Logs an error through a specified method of the current instance type.

`LogInfo(String, String, String)`

`LogInfo(String, String, String)`

Logs information through a specified method of the current instance type.

# SqlClientLogger.IsEnabled SqlClientLogger.IsEnabled

## In this Article

Gets a value that indicates whether bid tracing is enabled.

```
public bool IsLoggingEnabled { get; }
```

```
member this.IsEnabled : bool
```

## Returns

**Boolean Boolean**

`true` if bid tracing is enabled; otherwise, `false`.

# SqlClientLogger.LogAssert

## In this Article

Logs the specified message if `value` is `false`.

```
public bool LogAssert (bool value, string type, string method, string message);  
member this.LogAssert : bool * string * string * string -> bool
```

### Parameters

|   |         |
|---|---------|
| value   | Boolean |
| <code>false</code> to log the message; otherwise, <code>true</code> . |         |
| type  | String  |
| The type to be logged.  |         |
| method  | String  |
| The logging method.   |         |
| message   | String  |
| The message to be logged.   |         |

### Returns

Boolean

`true` if the message is not logged; otherwise, `false`.

# SqlClientLogger.LogError SqlClientLogger.LogError

## In this Article

Logs an error through a specified method of the current instance type.

```
public void LogError (string type, string method, string message);  
member this.LogError : string * string * string -> unit
```

### Parameters

|                           |   |
|---------------------------|---|
| type                      | <a href="#">String</a> <a href="#">String</a> |
| The type to be logged.    |   |
| method                    | <a href="#">String</a> <a href="#">String</a> |
| The logging method.       |   |
| message                   | <a href="#">String</a> <a href="#">String</a> |
| The message to be logged. |   |

# SqlClientLogger.LogInfo SqlClientLogger.LogInfo

## In this Article

Logs information through a specified method of the current instance type.

```
public void LogInfo (string type, string method, string message);  
member this.LogInfo : string * string * string -> unit
```

## Parameters

|                           |   |
|---------------------------|---|
| type                      | <a href="#">String</a> <a href="#">String</a> |
| The type to be logged.    |   |
| method                    | <a href="#">String</a> <a href="#">String</a> |
| The logging method.       |   |
| message                   | <a href="#">String</a> <a href="#">String</a> |
| The message to be logged. |   |

# SqlClientLogger

## In this Article

Initializes a new instance of the [SqlClientLogger](#) class.

```
public SqlClientLogger ();
```

# SqlClientMetaDataCollectionNames SqlClientMetaData CollectionNames Class

Provides a list of constants for use with the **GetSchema** method to retrieve metadata collections.

## Declaration

```
public static class SqlClientMetaDataCollectionNames  
type SqlClientMetaDataCollectionNames = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Fields

### Columns

#### Columns

A constant for use with the **GetSchema** method that represents the **Columns** collection.

### Databases

#### Databases

A constant for use with the **GetSchema** method that represents the **Databases** collection.

### ForeignKeys

#### ForeignKeys

A constant for use with the **GetSchema** method that represents the **ForeignKeys** collection.

### IndexColumns

#### IndexColumns

A constant for use with the **GetSchema** method that represents the **IndexColumns** collection.

### Indexes

#### Indexes

A constant for use with the **GetSchema** method that represents the **Indexes** collection.

### Parameters

#### Parameters

A constant for use with the **GetSchema** method that represents the **Parameters** collection.

ProcedureColumns

ProcedureColumns

A constant for use with the **GetSchema** method that represents the **ProcedureColumns** collection.

Procedures

Procedures

A constant for use with the **GetSchema** method that represents the **Procedures** collection.

Tables

Tables

A constant for use with the **GetSchema** method that represents the **Tables** collection.

UserDefinedTypes

UserDefinedTypes

A constant for use with the **GetSchema** method that represents the **UserDefinedTypes** collection.

Users

Users

A constant for use with the **GetSchema** method that represents the **Users** collection.

ViewColumns

ViewColumns

A constant for use with the **GetSchema** method that represents the **ViewColumns** collection.

Views

Views

A constant for use with the **GetSchema** method that represents the **Views** collection.

## See Also

# SqlClientMetaDataCollectionNames.Columns SqlClient MetaDataCollectionNames.Columns

## In this Article

A constant for use with the **GetSchema** method that represents the **Columns** collection.

```
public static readonly string Columns;  
  
staticval mutable Columns : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Databases SqlClient MetaDataCollectionNames.Databases

## In this Article

A constant for use with the **GetSchema** method that represents the **Databases** collection.

```
public static readonly string Databases;  
staticval mutable Databases : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.ForeignKeys SqlClientMetaDataCollectionNames.ForeignKeys

## In this Article

A constant for use with the **GetSchema** method that represents the **ForeignKeys** collection.

```
public static readonly string ForeignKeys;  
staticval mutable ForeignKeys : string
```

Returns

[String](#) [String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.IndexColumns SqlClientMetaDataCollectionNames.IndexColumns

## In this Article

A constant for use with the **GetSchema** method that represents the **IndexColumns** collection.

```
public static readonly string IndexColumns;  
staticval mutable IndexColumns : string
```

Returns

[String String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Indexes SqlClient MetaDataCollectionNames.Indexes

## In this Article

A constant for use with the **GetSchema** method that represents the **Indexes** collection.

```
public static readonly string Indexes;  
staticval mutable Indexes : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Parameters SqlClient MetaDataCollectionNames.Parameters

## In this Article

A constant for use with the **GetSchema** method that represents the **Parameters** collection.

```
public static readonly string Parameters;  
staticval mutable Parameters : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.ProcedureColumns

## SqlClientMetaDataCollectionNames.ProcedureColumns

### In this Article

A constant for use with the **GetSchema** method that represents the **ProcedureColumns** collection.

```
public static readonly string ProcedureColumns;  
staticval mutable ProcedureColumns : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Procedures SqlClient MetaDataCollectionNames.Procedures

## In this Article

A constant for use with the **GetSchema** method that represents the **Procedures** collection.

```
public static readonly string Procedures;  
staticval mutable Procedures : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Tables SqlClientMetaDataCollectionNames.Tables

## In this Article

A constant for use with the **GetSchema** method that represents the **Tables** collection.

```
public static readonly string Tables;  
staticval mutable Tables : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.UserDefinedTypes

# SqlClientMetaDataCollectionNames.UserDefinedTypes

## In this Article

A constant for use with the **GetSchema** method that represents the **UserDefinedTypes** collection.

```
public static readonly string UserDefinedTypes;  
staticval mutable UserDefinedTypes : string
```

Returns

[String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Users SqlClientMetaDataCollectionNames.Users

## In this Article

A constant for use with the **GetSchema** method that represents the **Users** collection.

```
public static readonly string Users;  
staticval mutable Users : string
```

Returns

[String String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.ViewColumns SqlClientMetaDataCollectionNames.ViewColumns

## In this Article

A constant for use with the **GetSchema** method that represents the **ViewColumns** collection.

```
public static readonly string ViewColumns;  
staticval mutable ViewColumns : string
```

Returns

[String](#) [String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

# SqlClientMetaDataCollectionNames.Views SqlClientMetaDataCollectionNames.Views

## In this Article

A constant for use with the **GetSchema** method that represents the **Views** collection.

```
public static readonly string Views;  
staticval mutable Views : string
```

Returns

[String String](#)

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

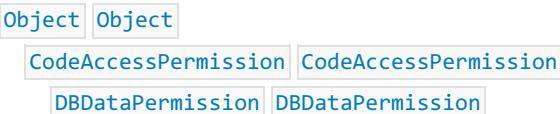
# SqlClientPermission SqlClientPermission Class

Enables the .NET Framework Data Provider for SQL Server to help make sure that a user has a security level sufficient to access a data source.

## Declaration

```
[Serializable]
public sealed class SqlClientPermission : System.Data.Common.DBDataPermission
type SqlClientPermission = class
    inherit DBDataPermission
```

## Inheritance Hierarchy



## Remarks

The [IsUnrestricted](#) property takes precedence over the [AllowBlankPassword](#) property. Therefore, if you set [AllowBlankPassword](#) to `false`, you must also set [IsUnrestricted](#) to `false` to prevent a user from making a connection using a blank password.

### Note

When using code access security permissions for ADO.NET, the correct pattern is to start with the most restrictive case (no permissions at all) and then add the specific permissions that are needed for the particular task that the code needs to perform. The opposite pattern, starting with all permissions and then denying a specific permission, is not secure, because there are many ways of expressing the same connection string. For example, if you start with all permissions and then attempt to deny the use of the connection string "server=someserver", the string "server=someserver.mycompany.com" would still be allowed. By always starting by granting no permissions at all, you reduce the chances that there are holes in the permission set.

## Constructors

[SqlClientPermission\(\)](#)

[SqlClientPermission\(\)](#)

Initializes a new instance of the [SqlClientPermission](#) class.

[SqlClientPermission\(PermissionState\)](#)

[SqlClientPermission\(PermissionState\)](#)

Initializes a new instance of the [SqlClientPermission](#) class.

[SqlClientPermission\(PermissionState, Boolean\)](#)

[SqlClientPermission\(PermissionState, Boolean\)](#)

Initializes a new instance of the [SqlClientPermission](#) class.

## Methods

Add(String, String, KeyRestrictionBehavior)

Add(String, String, KeyRestrictionBehavior)

Adds a new connection string and a set of restricted keywords to the [SqlClientPermission](#) object.

Copy()

Copy()

Returns the [SqlClientPermission](#) as an [IPermission](#).

## See Also

# SqlClientPermission.Add SqlClientPermission.Add

## In this Article

Adds a new connection string and a set of restricted keywords to the [SqlClientPermission](#) object.

```
public override void Add (string connectionString, string restrictions,
System.Data.KeyRestrictionBehavior behavior);

override this.Add : string * string * System.Data.KeyRestrictionBehavior -> unit
```

## Parameters

|   |   |
|---|---|
| connectionString  | <a href="#">String</a> <a href="#">String</a>                                 |
| The connection string.  |   |
| restrictions  | <a href="#">String</a> <a href="#">String</a>                                 |
| The key restrictions.   |   |
| behavior  | <a href="#">KeyRestrictionBehavior</a> <a href="#">KeyRestrictionBehavior</a> |
| One of the <a href="#">KeyRestrictionBehavior</a> enumerations. |   |

## Remarks

Use this method to configure which connection strings are allowed by a particular permission object. For example, use the following code fragment if you want to only allow a specific connection string and nothing else:

```
permission.Add("server=MyServer; database=MyDatabase; Integrated Security=true", "",  
KeyRestrictionBehavior.AllowOnly)
```

The following example allows connection strings that use any database, but only on the server named MyServer, with any user and password combination and containing no other connection string keywords:

```
permission.Add("server=MyServer;", "database=; user id=; password=;",  
KeyRestrictionBehavior.AllowOnly)
```

The following example uses the same scenario as above but allows for a failover partner that can be used when connecting to servers configured for mirroring:

```
permission.Add("server=MyServer; failover partner=MyMirrorServer", "database=; user id=; password=;",  
KeyRestrictionBehavior.AllowOnly)
```

### Note

When using code access security permissions for ADO.NET, the correct pattern is to start with the most restrictive case (no permissions at all) and then add the specific permissions that are needed for the particular task that the code needs to perform. The opposite pattern, starting with all permissions and then trying to deny a specific permission, is not secure, because there are many ways of expressing the same connection string. For example, if you start with all permissions and then attempt to deny the use of the connection string "server=someserver", the string "server=someserver.mycompany.com" would still be allowed. By always starting by granting no permissions at all, you reduce the chances that there are holes in the permission set.

See

[Code Access Security and ADO.NET](#)

Also

[ADO.NET Overview](#)

# SqlClientPermission.Copy SqlClientPermission.Copy

## In this Article

Returns the [SqlClientPermission](#) as an [IPermission](#).

```
public override System.Security.IPermission Copy ();
override this.Copy : unit -> System.Security.IPermission
```

Returns

[IPermission](#) [IPermission](#)

A copy of the current permission object.

[ADO.NET Overview](#)

See

Also

# SqlClientPermission SqlClientPermission

In this Article

## Overloads

|   |  |
|---|--|
| <a href="#">SqlClientPermission()</a>   | Initializes a new instance of the <a href="#">SqlClientPermission</a> class. |
| <a href="#">SqlClientPermission(PermissionState) SqlClientPermission(PermissionState)</a>                   | Initializes a new instance of the <a href="#">SqlClientPermission</a> class. |
| <a href="#">SqlClientPermission(PermissionState, Boolean) SqlClientPermission(PermissionState, Boolean)</a> | Initializes a new instance of the <a href="#">SqlClientPermission</a> class. |

## SqlClientPermission()

Initializes a new instance of the [SqlClientPermission](#) class.

```
[System.Obsolete("SqlClientPermission() has been deprecated. Use the  
SqlClientPermission(PermissionState.None) constructor. http://go.microsoft.com/fwlink/?  
linkid=14202", true)]  
[System.Obsolete("use SqlClientPermission(PermissionState.None)", true)]  
public SqlClientPermission();
```

Attributes

[ObsoleteAttribute](#) [ObsoleteAttribute](#)

See

[ADO.NET Overview](#)

Also

## SqlClientPermission(PermissionState)

## SqlClientPermission(PermissionState)

Initializes a new instance of the [SqlClientPermission](#) class.

```
public SqlClientPermission (System.Security.Permissions.PermissionState state);  
  
new System.Data.SqlClient.SqlClientPermission : System.Security.Permissions.PermissionState ->  
System.Data.SqlClient.SqlClientPermission
```

Parameters

state

[PermissionState](#) [PermissionState](#)

One of the [PermissionState](#) values.

See

[Code Access Security and ADO.NET](#)

Also

[ADO.NET Overview](#)

## SqlClientPermission(PermissionState, Boolean)

## SqlClientPermission(PermissionState, Boolean)

Initializes a new instance of the [SqlClientPermission](#) class.

```
[System.Obsolete("SqlClientPermission(PermissionState state, Boolean allowBlankPassword) has been  
deprecated. Use the SqlClientPermission(PermissionState.None) constructor.  
http://go.microsoft.com/fwlink/?LinkId=14202", true)]  
[System.Obsolete("use SqlClientPermission(PermissionState.None)", true)]  
public SqlClientPermission (System.Security.Permissions.PermissionState state, bool  
allowBlankPassword);  
  
new System.Data.SqlClient.SqlClientPermission : System.Security.Permissions.PermissionState * bool -> System.Data.SqlClient.SqlClientPermission
```

## Parameters

state PermissionState PermissionState

One of the [PermissionState](#) values.

allowBlankPassword Boolean Boolean

Indicates whether a blank password is allowed.

Attributes ObsoleteAttribute ObsoleteAttribute

## Remarks

The [PermissionState](#) enumeration takes precedence over the [AllowBlankPassword](#) property. Therefore, if you set [AllowBlankPassword](#) to [false](#), you must also set [PermissionState](#) to [None](#) to prevent a user from making a connection using a blank password. For an example demonstrating how to use security demands, see [Code Access Security and ADO.NET](#).

See [Code Access Security and ADO.NET](#)

Also [ADO.NET Overview](#)

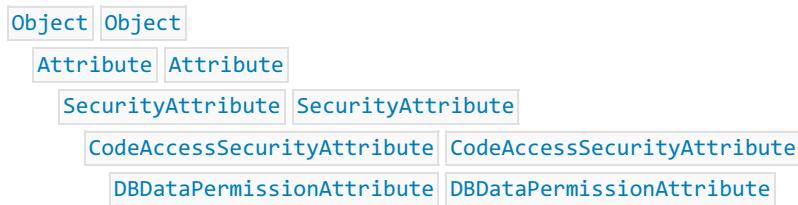
# SqlClientPermissionAttribute SqlClientPermission Attribute Class

Associates a security action with a custom security attribute.

## Declaration

```
[System.AttributeUsage(System.AttributeTargets.Assembly | System.AttributeTargets.Class |  
System.AttributeTargets.Constructor | System.AttributeTargets.Method |  
System.AttributeTargets.Struct, AllowMultiple=true, Inherited=false)]  
[System.AttributeUsage(System.AttributeTargets.Assembly | System.AttributeTargets.Class |  
System.AttributeTargets.Struct | System.AttributeTargets.Constructor |  
System.AttributeTargets.Method, AllowMultiple=true, Inherited=false)]  
[Serializable]  
public sealed class SqlClientPermissionAttribute : System.Data.Common.DBDataPermissionAttribute  
  
type SqlClientPermissionAttribute = class  
    inherit DBDataPermissionAttribute
```

## Inheritance Hierarchy



## Constructors

```
SqlClientPermissionAttribute(SecurityAction)  
SqlClientPermissionAttribute(SecurityAction)
```

Initializes a new instance of the [SqlClientPermissionAttribute](#) class.

## Methods

```
CreatePermission()  
CreatePermission()
```

Returns a [SqlClientPermission](#) object that is configured according to the attribute properties.

## See Also

# SqlClientPermissionAttribute.CreatePermission SqlClientPermissionAttribute.CreatePermission

## In this Article

Returns a [SqlClientPermission](#) object that is configured according to the attribute properties.

```
public override System.Security.IPermission CreatePermission ();  
  
abstract member CreatePermission : unit -> System.Security.IPermission  
override this.CreatePermission : unit -> System.Security.IPermission
```

Returns

[IPermission](#) [IPermission](#)

A [SqlClientPermission](#) object.

See

[ADO.NET Overview](#)

Also

# SqlClientPermissionAttribute SqlClientPermission Attribute

## In this Article

Initializes a new instance of the [SqlClientPermissionAttribute](#) class.

```
public SqlClientPermissionAttribute (System.Security.Permissions.SecurityAction action);  
new System.Data.SqlClient.SqlClientPermissionAttribute : System.Security.Permissions.SecurityAction  
-> System.Data.SqlClient.SqlClientPermissionAttribute
```

## Parameters

action [SecurityAction](#) [SecurityAction](#)

One of the [SecurityAction](#) values representing an action that can be performed by using declarative security.

See

[ADO.NET Overview](#)

Also

# SqlColumnEncryptionCertificateStoreProvider SqlColumnEncryptionCertificateStoreProvider Class

The implementation of the key store provider for Windows Certificate Store. This class enables using certificates stored in the Windows Certificate Store as column master keys. For details, see [Always Encrypted](#).

## Declaration

```
public class SqlColumnEncryptionCertificateStoreProvider :  
System.Data.SqlClient.SqlColumnEncryptionKeyStoreProvider  
  
type SqlColumnEncryptionCertificateStoreProvider = class  
inherit SqlColumnEncryptionKeyStoreProvider
```

## Inheritance Hierarchy

```
Object Object  
SqlColumnEncryptionKeyStoreProvider SqlColumnEncryptionKeyStoreProvider
```

## Constructors

```
SqlColumnEncryptionCertificateStoreProvider()  
SqlColumnEncryptionCertificateStoreProvider()
```

Key store provider for Windows Certificate Store.

## Fields

```
ProviderName
```

```
ProviderName
```

The provider name.

## Methods

```
DecryptColumnEncryptionKey(String, String, Byte[])  
DecryptColumnEncryptionKey(String, String, Byte[])
```

Decrypts the specified encrypted value of a column encryption key. The encrypted value is expected to be encrypted using the certificate with the specified key path and using the specified algorithm. The format of the key path should be "Local Machine/My/<certificate\_thumbprint>" or "Current User/My/<certificate\_thumbprint>".

```
EncryptColumnEncryptionKey(String, String, Byte[])  
EncryptColumnEncryptionKey(String, String, Byte[])
```

Encrypts a column encryption key using the certificate with the specified key path and using the specified algorithm. The format of the key path should be "Local Machine/My/<certificate\_thumbprint>" or "Current User/My/<certificate\_thumbprint>".

```
SignColumnMasterKeyMetadata(String, Boolean)
```

```
SignColumnMasterKeyMetadata(String, Boolean)
```

Digitally signs the column master key metadata with the column master key referenced by the `masterKeyPath` parameter.

```
VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])
```

```
VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])
```

This function must be implemented by the corresponding Key Store providers. This function should use an asymmetric key identified by a key path and verify the masterkey metadata consisting of (`masterKeyPath`, `allowEnclaveComputations`, `providerName`).

## See Also

# SqlColumnEncryptionCertificateStoreProvider.DecryptColumnEncryptionKey SqlColumnEncryptionCertificateStoreProvider.DecryptColumnEncryptionKey

## In this Article

Decrypts the specified encrypted value of a column encryption key. The encrypted value is expected to be encrypted using the certificate with the specified key path and using the specified algorithm. The format of the key path should be "Local Machine/My/<certificate\_thumbprint>" or "Current User/My/<certificate\_thumbprint>".

```
public override byte[] DecryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] encryptedColumnEncryptionKey);  
  
override this.DecryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

masterKeyPath String String

The master key path.

encryptionAlgorithm String String

The encryption algorithm. Currently, the only valid value is: RSA\_OAEP

encryptedColumnEncryptionKey Byte[]

The encrypted column encryption key.

## Returns

Byte[]

Returns Byte.

The decrypted column encryption key.

# SqlColumnEncryptionCertificateStoreProvider.EncryptColumnEncryptionKey SqlColumnEncryptionCertificateStoreProvider.EncryptColumnEncryptionKey

## In this Article

Encrypts a column encryption key using the certificate with the specified key path and using the specified algorithm. The format of the key path should be "Local Machine/My/<certificate\_thumbprint>" or "Current User/My/<certificate\_thumbprint>".

```
public override byte[] EncryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] columnEncryptionKey);  
  
override this.EncryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

masterKeyPath String String

The master key path.

encryptionAlgorithm String String

The encryption algorithm. Currently, the only valid value is: RSA\_OAEP

columnEncryptionKey Byte[]

The encrypted column encryption key.

## Returns

Byte[]

Returns Byte.

The encrypted column encryption key.

# SqlColumnEncryptionCertificateStoreProvider.ProviderName

## In this Article

The provider name.

```
public const string ProviderName;  
val mutable ProviderName : string
```

Returns

[String](#)

# SqlColumnEncryptionCertificateStoreProvider.SignColumnMasterKeyMetadata SqlColumnEncryptionCertificateStoreProvider.SignColumnMasterKeyMetadata

## In this Article

Digitally signs the column master key metadata with the column master key referenced by the `masterKeyPath` parameter.

```
public override byte[] SignColumnMasterKeyMetadata (string masterKeyPath, bool allowEnclaveComputations);  
override this.SignColumnMasterKeyMetadata : string * bool -> byte[]
```

## Parameters

`masterKeyPath` [String](#) [String](#)

The column master key path.

`allowEnclaveComputations` [Boolean](#) [Boolean](#)

`true` to indicate that the column master key supports enclave computations; otherwise, `false`.

## Returns

[Byte\[\]](#)

The signature of the column master key metadata.

# SqlColumnEncryptionCertificateStoreProvider

## In this Article

Key store provider for Windows Certificate Store.

```
public SqlColumnEncryptionCertificateStoreProvider ();
```

# SqlColumnEncryptionCertificateStoreProvider.VerifyColumnMasterKeyMetadata SqlColumnEncryptionCertificateStoreProvider.VerifyColumnMasterKeyMetadata

## In this Article

This function must be implemented by the corresponding Key Store providers. This function should use an asymmetric key identified by a key path and verify the masterkey metadata consisting of (masterKeyPath, allowEnclaveComputations, providerName).

```
public override bool VerifyColumnMasterKeyMetadata (string masterKeyPath, bool allowEnclaveComputations, byte[] signature);  
override this.VerifyColumnMasterKeyMetadata : string * bool * byte[] -> bool
```

## Parameters

masterKeyPath String String

The complete path of an asymmetric key. The path format is specific to a key store provider.

allowEnclaveComputations Boolean Boolean

A Boolean that indicates if this key can be sent to the trusted enclave.

signature Byte[]

The master key metadata siognature.

## Returns

Boolean Boolean

A Boolean value that indicates if the master key metadata can be verified based on the provided signature.

# SqlColumnEncryptionCngProvider SqlColumnEncryption CngProvider Class

The CMK Store provider implementation for using the Microsoft Cryptography API: Next Generation (CNG) with [Always Encrypted](#).

## Declaration

```
public class SqlColumnEncryptionCngProvider :  
System.Data.SqlClient.SqlColumnEncryptionKeyStoreProvider  
  
type SqlColumnEncryptionCngProvider = class  
    inherit SqlColumnEncryptionKeyStoreProvider
```

## Inheritance Hierarchy

```
Object Object  
SqlColumnEncryptionKeyStoreProvider SqlColumnEncryptionKeyStoreProvider
```

## Remarks

Enables storing Always Encrypted column master key keys in a store, such as a hardware security module (HSM), that supports the Microsoft Cryptography API: Next Generation (CNG).

## Constructors

```
SqlColumnEncryptionCngProvider()  
SqlColumnEncryptionCngProvider()
```

Initializes a new instance of the [SqlColumnEncryptionCngProvider](#) class.

## Fields

```
ProviderName  
ProviderName
```

A constant string for the provider name 'MSSQL\_CNG\_STORE'.

## Methods

```
DecryptColumnEncryptionKey(String, String, Byte[])
DecryptColumnEncryptionKey(String, String, Byte[])
```

Decrypts the given encrypted value using an asymmetric key specified by the key path and the specified algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CNG key store provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
EncryptColumnEncryptionKey(String, String, Byte[])
EncryptColumnEncryptionKey(String, String, Byte[])
```

Encrypts the given plain text column encryption key using an asymmetric key specified by the key path and the

specified algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CNG key store provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

`SignColumnMasterKeyMetadata(String, Boolean)`

`SignColumnMasterKeyMetadata(String, Boolean)`

Throws a [NotSupportedException](#) exception in all cases.

`VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])`

`VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])`

This function must be implemented by the corresponding Key Store providers. This function should use an asymmetric key identified by a key path and verify the masterkey metadata consisting of (masterKeyPath, allowEnclaveComputations, providerName).

## See Also

# SqlColumnEncryptionCngProvider.DecryptColumnEncryptionKey SqlColumnEncryptionCngProvider.DecryptColumnEncryptionKey

## In this Article

Decrypts the given encrypted value using an asymmetric key specified by the key path and the specified algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CNG key store provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
public override byte[] DecryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] encryptedColumnEncryptionKey);  
  
override this.DecryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

|                                      |   |
|--------------------------------------|---|
| masterKeyPath                        | <a href="#">String</a> <a href="#">String</a> |
| The master key path.                 |   |
| encryptionAlgorithm                  | <a href="#">String</a> <a href="#">String</a> |
| The encryption algorithm.            |   |
| encryptedColumnEncryptionKey         | <a href="#">Byte</a> []                       |
| The encrypted column encryption key. |   |

## Returns

[Byte](#)[]

The decrypted column encryption key.

# SqlColumnEncryptionCngProvider.EncryptColumnEncryptionKey SqlColumnEncryptionCngProvider.EncryptColumnEncryptionKey

## In this Article

Encrypts the given plain text column encryption key using an asymmetric key specified by the key path and the specified algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CNG key store provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
public override byte[] EncryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] columnEncryptionKey);  
  
override this.EncryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

|                                      |                        |
|--------------------------------------|------------------------|
| masterKeyPath                        | <a href="#">String</a> |
| The master key path.                 |                        |
| encryptionAlgorithm                  | <a href="#">String</a> |
| The encryption algorithm.            |                        |
| columnEncryptionKey                  | <a href="#">Byte[]</a> |
| The encrypted column encryption key. |                        |

## Returns

[Byte\[\]](#)

The encrypted column encryption key.

# SqlColumnEncryptionCngProvider.ProviderName SqlColumnEncryptionCngProvider.ProviderName

## In this Article

A constant string for the provider name 'MSSQL\_CNG\_STORE'.

```
public const string ProviderName;  
val mutable ProviderName : string
```

## Returns

[String](#) [String](#)

# SqlColumnEncryptionCngProvider.SignColumnMasterKeyMetadata SqlColumnEncryptionCngProvider.SignColumnMasterKeyMetadata

## In this Article

Throws a [NotSupportedException](#) exception in all cases.

```
public override byte[] SignColumnMasterKeyMetadata (string masterKeyPath, bool allowEnclaveComputations);  
override this.SignColumnMasterKeyMetadata : string * bool -> byte[]
```

## Parameters

masterKeyPath [String](#) [String](#)

The column master key path. The path format is specific to a key store provider.

allowEnclaveComputations [Boolean](#) [Boolean](#)

`true` to indicate that the column master key supports enclave computations; otherwise, `false`.

## Returns

[Byte\[\]](#)

The signature of the column master key metadata.

## Exceptions

[NotSupportedException](#) [NotSupportedException](#)

In all cases.

## Remarks

The [SignColumnMasterKeyMetadata](#) method must be implemented by the corresponding key store providers. [SignColumnMasterKeyMetadata](#) should use an asymmetric key identified by a key path and sign the masterkey metadata consisting of `masterKeyPath`, `allowEnclaveComputations`, and `providerName`.

# SqlColumnEncryptionCngProvider

## In this Article

Initializes a new instance of the [SqlColumnEncryptionCngProvider](#) class.

```
public SqlColumnEncryptionCngProvider ();
```

# SqlColumnEncryptionCngProvider.VerifyColumnMasterKeyMetadata SqlColumnEncryptionCngProvider.VerifyColumnMasterKeyMetadata

## In this Article

This function must be implemented by the corresponding Key Store providers. This function should use an asymmetric key identified by a key path and verify the masterkey metadata consisting of (masterKeyPath, allowEnclaveComputations, providerName).

```
public override bool VerifyColumnMasterKeyMetadata (string masterKeyPath, bool  
allowEnclaveComputations, byte[] signature);  
  
override this.VerifyColumnMasterKeyMetadata : string * bool * byte[] -> bool
```

## Parameters

masterKeyPath [String](#) [String](#)

The complete path of an asymmetric key. The path format is specific to a key store provider.

allowEnclaveComputations [Boolean](#) [Boolean](#)

A Boolean that indicates if this key can be sent to the trusted enclave.

signature [Byte](#)[]

The master key metadata signature.

## Returns

[Boolean](#) [Boolean](#)

A Boolean that indicates if the master key metadata can be verified based on the provided signature.

# SqlColumnEncryptionCspProvider SqlColumnEncryptionCspProvider Class

The CMK Store provider implementation for using Microsoft CAPI based Cryptographic Service Providers (CSP) with [Always Encrypted](#).

## Declaration

```
public class SqlColumnEncryptionCspProvider :  
System.Data.SqlClient.SqlColumnEncryptionKeyStoreProvider  
  
type SqlColumnEncryptionCspProvider = class  
inherit SqlColumnEncryptionKeyStoreProvider
```

## Inheritance Hierarchy

```
Object Object  
SqlColumnEncryptionKeyStoreProvider SqlColumnEncryptionKeyStoreProvider
```

## Remarks

Enables storing Always Encrypted column master key keys in a store, such as a hardware security module (HSM), that supports the Microsoft CAPI based Cryptographic Service Providers (CSP).

## Constructors

```
SqlColumnEncryptionCspProvider()  
SqlColumnEncryptionCspProvider()
```

Initializes a new instance of the [SqlColumnEncryptionCspProvider](#) class.

## Fields

```
ProviderName  
ProviderName
```

A constant string for the provider name 'MSSQL\_CSP\_PROVIDER'.

## Methods

```
DecryptColumnEncryptionKey(String, String, Byte[])
DecryptColumnEncryptionKey(String, String, Byte[])
```

Decrypts the given encrypted value using an asymmetric key specified by the key path and algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CSP provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
EncryptColumnEncryptionKey(String, String, Byte[])
EncryptColumnEncryptionKey(String, String, Byte[])
```

Encrypts the given plain text column encryption key using an asymmetric key specified by the key path and the

specified algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CSP provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
SignColumnMasterKeyMetadata(String, Boolean)
```

```
SignColumnMasterKeyMetadata(String, Boolean)
```

Throws a [NotSupportedException](#) exception in all cases.

```
VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])
```

```
VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])
```

This function must be implemented by the corresponding Key Store providers. This function should use an asymmetric key identified by a key path and sign the masterkey metadata consisting of (masterKeyPath, allowEnclaveComputations, providerName).

## See Also

# SqlColumnEncryptionCspProvider.DecryptColumnEncryptionKey SqlColumnEncryptionCspProvider.DecryptColumnEncryptionKey

## In this Article

Decrypts the given encrypted value using an asymmetric key specified by the key path and algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CSP provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
public override byte[] DecryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] encryptedColumnEncryptionKey);  
  
override this.DecryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

masterKeyPath String String

The master key path.

encryptionAlgorithm String String

The encryption algorithm.

encryptedColumnEncryptionKey Byte[]

The encrypted column encryption key.

## Returns

Byte[]

The decrypted column encryption key.

# SqlColumnEncryptionCspProvider.EncryptColumn EncryptionKey SqlColumnEncryptionCspProvider.Encrypt ColumnEncryptionKey

## In this Article

Encrypts the given plain text column encryption key using an asymmetric key specified by the key path and the specified algorithm. The key path will be in the format of [ProviderName]/KeyIdentifier and should be an asymmetric key stored in the specified CSP provider. The valid algorithm used to encrypt/decrypt the CEK is 'RSA\_OAEP'.

```
public override byte[] EncryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] columnEncryptionKey);  
  
override this.EncryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

|                                      |                         |
|--------------------------------------|-------------------------|
| masterKeyPath                        | <a href="#">String</a>  |
| The master key path.                 |                         |
| encryptionAlgorithm                  | <a href="#">String</a>  |
| The encryption algorithm.            |                         |
| columnEncryptionKey                  | <a href="#">Byte</a> [] |
| The encrypted column encryption key. |                         |

## Returns

[Byte](#)[]

The encrypted column encryption key.

# SqlColumnEncryptionCspProvider.ProviderName SqlColumnEncryptionCspProvider.ProviderName

## In this Article

A constant string for the provider name 'MSSQL\_CSP\_PROVIDER'.

```
public const string ProviderName;  
val mutable ProviderName : string
```

Returns

[String](#) [String](#)

# SqlColumnEncryptionCspProvider.SignColumnMasterKeyMetadata SqlColumnEncryptionCspProvider.SignColumnMasterKeyMetadata

## In this Article

Throws a [NotSupportedException](#) exception in all cases.

```
public override byte[] SignColumnMasterKeyMetadata (string masterKeyPath, bool allowEnclaveComputations);  
override this.SignColumnMasterKeyMetadata : string * bool -> byte[]
```

### Parameters

masterKeyPath [String](#) [String](#)

The column master key path. The path format is specific to a key store provider.

allowEnclaveComputations [Boolean](#) [Boolean](#)

`true` to indicate that the column master key supports enclave computations; otherwise, `false`.

### Returns

[Byte\[\]](#)

The signature of the column master key metadata.

### Exceptions

[NotSupportedException](#) [NotSupportedException](#)

In all cases.

## Remarks

The [SignColumnMasterKeyMetadata](#) method must be implemented by the corresponding key store providers. [SignColumnMasterKeyMetadata](#) should use an asymmetric key identified by a key path and sign the masterkey metadata consisting of `masterKeyPath`, `allowEnclaveComputations`, and `providerName`.

# SqlColumnEncryptionCspProvider

## In this Article

Initializes a new instance of the [SqlColumnEncryptionCspProvider](#) class.

```
public SqlColumnEncryptionCspProvider ();
```

# SqlColumnEncryptionCspProvider.VerifyColumnMasterKeyMetadata SqlColumnEncryptionCspProvider.VerifyColumnMasterKeyMetadata

## In this Article

This function must be implemented by the corresponding Key Store providers. This function should use an asymmetric key identified by a key path and sign the masterkey metadata consisting of (masterKeyPath, allowEnclaveComputations, providerName).

```
public override bool VerifyColumnMasterKeyMetadata (string masterKeyPath, bool  
allowEnclaveComputations, byte[] signature);  
  
override this.VerifyColumnMasterKeyMetadata : string * bool * byte[] -> bool
```

## Parameters

masterKeyPath [String](#) [String](#)

The complete path of an asymmetric key. The path format is specific to a key store provider.

allowEnclaveComputations [Boolean](#) [Boolean](#)

A boolean that indicates if this key can be sent to the trusted enclave.

signature [Byte](#)[]

Master key metadata signature.

## Returns

[Boolean](#) [Boolean](#)

A Boolean that indicates if the master key metadata can be verified based on the provided signature.

# SqlColumnEncryptionEnclaveProvider SqlColumnEncryptionEnclaveProvider Class

The base class that defines the interface for enclave providers for Always Encrypted.

## Declaration

```
public abstract class SqlColumnEncryptionEnclaveProvider  
type SqlColumnEncryptionEnclaveProvider = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

An enclave is a protected region of memory inside SQL Server, used for computations on encrypted columns. An enclave provider encapsulates the client-side implementation details of the enclave attestation protocol as well as the logic for creating and caching enclave sessions.

## Constructors

```
SqlColumnEncryptionEnclaveProvider()  
SqlColumnEncryptionEnclaveProvider()
```

Initializes a new instance of the [SqlColumnEncryptionEnclaveProvider](#) class

## Methods

```
CreateEnclaveSession(Byte[], ECDiffieHellmanCng, String, String, SqlEnclaveSession, Int64)  
CreateEnclaveSession(Byte[], ECDiffieHellmanCng, String, String, SqlEnclaveSession, Int64)
```

When overridden in a derived class, performs enclave attestation, generates a symmetric key for the session, creates a an enclave session and stores the session information in the cache.

```
GetAttestationParameters()  
GetAttestationParameters()
```

Gets the information that [SqlClient](#) subsequently uses to initiate the process of attesting the enclave and to establish a secure session with the enclave.

```
GetEnclaveSession(String, String, SqlEnclaveSession, Int64)  
GetEnclaveSession(String, String, SqlEnclaveSession, Int64)
```

When overridden in a derived class, looks up an existing enclave session information in the enclave session cache. If the enclave provider doesn't implement enclave session caching, this method is expected to return [null](#) in the [sqlEnclaveSession](#) parameter.

```
InvalidateEnclaveSession(String, String, SqlEnclaveSession)
```

```
InvalidateEnclaveSession(String, String, SqlEnclaveSession)
```

When overridden in a derived class, looks up and evicts an enclave session from the enclave session cache, if the provider implements session caching.

# SqlColumnEncryptionEnclaveProvider.CreateEnclaveSession SqlColumnEncryptionEnclaveProvider.CreateEnclaveSession

## In this Article

When overridden in a derived class, performs enclave attestation, generates a symmetric key for the session, creates a an enclave session and stores the session information in the cache.

```
public abstract void CreateEnclaveSession (byte[] enclaveAttestationInfo,
System.Security.Cryptography.ECDiffieHellmanCng clientDiffieHellmanKey, string attestationUrl,
string servername, out System.Data.SqlClient.SqlEnclaveSession sqlEnclaveSession, out long counter);

abstract member CreateEnclaveSession : byte[] * System.Security.Cryptography.ECDiffieHellmanCng * 
string * string *  * -> unit
```

## Parameters

enclaveAttestationInfo

[Byte\[\]](#)

The information the provider uses to attest the enclave and generate a symmetric key for the session. The format of this information is specific to the enclave attestation protocol.

clientDiffieHellmanKey

[ECDiffieHellmanCng](#) [ECDiffieHellmanCng](#)

A Diffie-Hellman algorithm object that encapsulates a client-side key pair.

attestationUrl

[String](#) [String](#)

The endpoint of an attestation service for attesting the enclave.

servername

[String](#) [String](#)

The name of the SQL Server instance containing the enclave.

sqlEnclaveSession

[SqlEnclaveSession](#) [SqlEnclaveSession](#)

The requested enclave session or `null` if the provider doesn't implement session caching.

counter

[Int64](#) [Int64](#)

A counter that the enclave provider is expected to increment each time `SqlClient` retrieves the session from the cache. The purpose of this field is to prevent replay attacks.

# SqlColumnEncryptionEnclaveProvider.GetAttestationParameters

## In this Article

Gets the information that SqlClient subsequently uses to initiate the process of attesting the enclave and to establish a secure session with the enclave.

```
public abstract System.Data.SqlClient.SqlEnclaveAttestationParameters GetAttestationParameters ();  
abstract member GetAttestationParameters : unit ->  
System.Data.SqlClient.SqlEnclaveAttestationParameters
```

## Returns

[SqlEnclaveAttestationParameters](#) [SqlEnclaveAttestationParameters](#)

The information SqlClient subsequently uses to initiate the process of attesting the enclave and to establish a secure session with the enclave.

# SqlColumnEncryptionEnclaveProvider.GetEnclaveSession

## SqlColumnEncryptionEnclaveProvider.GetEnclaveSession

### In this Article

When overridden in a derived class, looks up an existing enclave session information in the enclave session cache. If the enclave provider doesn't implement enclave session caching, this method is expected to return `null` in the `sqlEnclaveSession` parameter.

```
public abstract void GetEnclaveSession (string serverName, string attestationUrl, out System.Data.SqlClient.SqlEnclaveSession sqlEnclaveSession, out long counter);  
abstract member GetEnclaveSession : string * string *  * -> unit
```

### Parameters

|  |                                   |
|--|-----------------------------------|
| serverName   | <a href="#">String</a>            |
| The name of the SQL Server instance containing the enclave.  |                                   |
| attestationUrl   | <a href="#">String</a>            |
| The endpoint of an attestation service, SqlClient contacts to attest the enclave.  |                                   |
| sqlEnclaveSession  | <a href="#">SqlEnclaveSession</a> |
| When this method returns, the requested enclave session or <code>null</code> if the provider doesn't implement session caching. This parameter is treated as uninitialized.    |                                   |
| counter  | <a href="#">Int64</a>             |
| A counter that the enclave provider is expected to increment each time SqlClient retrieves the session from the cache. The purpose of this field is to prevent replay attacks. |                                   |

# SqlColumnEncryptionEnclaveProvider.InvalidateEnclaveSession SqlColumnEncryptionEnclaveProvider.InvalidateEnclaveSession

## In this Article

When overridden in a derived class, looks up and evicts an enclave session from the enclave session cache, if the provider implements session caching.

```
public abstract void InvalidateEnclaveSession (string serverName, string enclaveAttestationUrl,  
System.Data.SqlClient.SqlEnclaveSession enclaveSession);  
  
abstract member InvalidateEnclaveSession : string * string * System.Data.SqlClient.SqlEnclaveSession  
-> unit
```

## Parameters

|   |   |
|---|---|
| serverName  | <a href="#">String</a> <a href="#">String</a>                       |
| The name of the SQL Server instance containing the enclave.                       |   |
| enclaveAttestationUrl   | <a href="#">String</a> <a href="#">String</a>                       |
| The endpoint of an attestation service, SqlClient contacts to attest the enclave. |   |
| enclaveSession  | <a href="#">SqlEnclaveSession</a> <a href="#">SqlEnclaveSession</a> |
| The session to be invalidated.  |   |

# SqlColumnEncryptionEnclaveProvider

## In this Article

Initializes a new instance of the [SqlColumnEncryptionEnclaveProvider](#) class

```
protected SqlColumnEncryptionEnclaveProvider ();
```

# SqlColumnEncryptionKeyStoreProvider SqlColumn EncryptionKeyStoreProvider Class

Base class for all key store providers. A custom provider must derive from this class and override its member functions and then register it using `SqlConnection.RegisterColumnEncryptionKeyStoreProviders()`. For details see, [Always Encrypted](#).

## Declaration

```
public abstract class SqlColumnEncryptionKeyStoreProvider  
type SqlColumnEncryptionKeyStoreProvider = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

`SqlColumnEncryptionKeyStoreProvider()`

`SqlColumnEncryptionKeyStoreProvider()`

Initializes a new instance of the `SqlColumnEncryptionKeyStoreProvider` class.

## Methods

`DecryptColumnEncryptionKey(String, String, Byte[])`

`DecryptColumnEncryptionKey(String, String, Byte[])`

Decrypts the specified encrypted value of a column encryption key. The encrypted value is expected to be encrypted using the column master key with the specified key path and using the specified algorithm.

`EncryptColumnEncryptionKey(String, String, Byte[])`

`EncryptColumnEncryptionKey(String, String, Byte[])`

Encrypts a column encryption key using the column master key with the specified key path and using the specified algorithm.

`SignColumnMasterKeyMetadata(String, Boolean)`

`SignColumnMasterKeyMetadata(String, Boolean)`

When implemented in a derived class, digitally signs the column master key metadata with the column master key referenced by the `masterKeyPath` parameter. The input values used to generate the signature should be the specified values of the `masterKeyPath` and `allowEnclaveComputations` parameters.

`VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])`

`VerifyColumnMasterKeyMetadata(String, Boolean, Byte[])`

When implemented in a derived class, this method is expected to verify the specified signature is valid for the column master key with the specified key path and the specified enclave behavior. The default implementation

throws NotImplementedException.

## See Also

# SqlColumnEncryptionKeyStoreProvider.DecryptColumnEncryptionKey SqlColumnEncryptionKeyStoreProvider.DecryptColumnEncryptionKey

## In this Article

Decrypts the specified encrypted value of a column encryption key. The encrypted value is expected to be encrypted using the column master key with the specified key path and using the specified algorithm.

```
public abstract byte[] DecryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] encryptedColumnEncryptionKey);  
  
abstract member DecryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

## Parameters

masterKeyPath String String

The master key path.

encryptionAlgorithm String String

The encryption algorithm.

encryptedColumnEncryptionKey Byte[]

The encrypted column encryption key.

## Returns

[Byte\[\]](#)

Returns [Byte](#).

The decrypted column encryption key.

# SqlColumnEncryptionKeyStoreProvider.EncryptColumnEncryptionKey

## SqlColumnEncryptionKeyStoreProvider.EncryptColumnEncryptionKey

### In this Article

Encrypts a column encryption key using the column master key with the specified key path and using the specified algorithm.

```
public abstract byte[] EncryptColumnEncryptionKey (string masterKeyPath, string encryptionAlgorithm,  
byte[] columnEncryptionKey);  
  
abstract member EncryptColumnEncryptionKey : string * string * byte[] -> byte[]
```

### Parameters

|                                      |   |
|--------------------------------------|---|
| masterKeyPath                        | <a href="#">String</a> <a href="#">String</a> |
| The master key path.                 |   |
| encryptionAlgorithm                  | <a href="#">String</a> <a href="#">String</a> |
| The encryption algorithm.            |   |
| columnEncryptionKey                  | <a href="#">Byte</a> []                       |
| The encrypted column encryption key. |   |

### Returns

[Byte](#)[]

Returns [Byte](#).

The encrypted column encryption key.

# SqlColumnEncryptionKeyStoreProvider.SignColumnMasterKeyMetadata.SqlColumnEncryptionKeyStoreProvider.SignColumnMasterKeyMetadata

## In this Article

When implemented in a derived class, digitally signs the column master key metadata with the column master key referenced by the `masterKeyPath` parameter. The input values used to generate the signature should be the specified values of the `masterKeyPath` and `allowEnclaveComputations` parameters.

```
public virtual byte[] SignColumnMasterKeyMetadata (string masterKeyPath, bool allowEnclaveComputations);  
  
abstract member SignColumnMasterKeyMetadata : string * bool -> byte[]  
override this.SignColumnMasterKeyMetadata : string * bool -> byte[]
```

## Parameters

`masterKeyPath` [String](#) [String](#)

The column master key path.

`allowEnclaveComputations` [Boolean](#) [Boolean](#)

`true` to indicate that the column master key supports enclave computations; otherwise, `false`.

## Returns

[Byte\[\]](#)

The signature of the column master key metadata.

## Exceptions

[NotImplementedException](#) [NotImplementedException](#)

In all cases.

## Remarks

To ensure that the [SignColumnMasterKeyMetadata](#) method doesn't break applications that rely on an old API, it throws a [NotImplementedException](#) exception by default.

The [SignColumnMasterKeyMetadata](#) method will be used by client tools that generate Column Master Keys (CMK) for customers. [SignColumnMasterKeyMetadata](#) must be implemented by the corresponding key store providers that wish to use enclaves with [Always Encrypted](#).

# SqlColumnEncryptionKeyStoreProvider

## In this Article

Initializes a new instance of the SqlColumnEncryptionKeyStoreProviderClass.

```
protected SqlColumnEncryptionKeyStoreProvider ();
```

# SqlColumnEncryptionKeyStoreProvider.VerifyColumnMasterKeyMetadata SqlColumnEncryptionKeyStoreProvider.VerifyColumnMasterKeyMetadata

## In this Article

When implemented in a derived class, this method is expected to verify the specified signature is valid for the column master key with the specified key path and the specified enclave behavior. The default implementation throws NotImplementedException.

```
public virtual bool VerifyColumnMasterKeyMetadata (string masterKeyPath, bool allowEnclaveComputations, byte[] signature);

abstract member VerifyColumnMasterKeyMetadata : string * bool * byte[] -> bool
override this.VerifyColumnMasterKeyMetadata : string * bool * byte[] -> bool
```

## Parameters

|  |                         |
|--|-------------------------|
| masterKeyPath  | <a href="#">String</a>  |
| The column master key path.  |                         |
| allowEnclaveComputations   | <a href="#">Boolean</a> |
| Indicates whether the column master key supports enclave computations. |                         |
| signature  | <a href="#">Byte[]</a>  |
| The signature of the column master key metadata.                       |                         |

## Returns

[Boolean](#)

When implemented in a derived class, the method is expected to return true if the specified signature is valid, or false if the specified signature is not valid. The default implementation throws NotImplementedException.

# SqlCommand SqlCommand Class

Represents a Transact-SQL statement or stored procedure to execute against a SQL Server database. This class cannot be inherited.

## Declaration

```
public sealed class SqlCommand : System.Data.Common.DbCommand, ICloneable, IDisposable

type SqlCommand = class
    inherit DbCommand
    interface IDbCommand
    interface ICloneable
    interface IDisposable
```

## Inheritance Hierarchy



## Remarks

When an instance of [SqlCommand](#) is created, the read/write properties are set to their initial values. For a list of these values, see the [SqlCommand](#) constructor.

[SqlCommand](#) features the following methods for executing commands at a SQL Server database:

| ITEM                                  | DESCRIPTION   |
|---------------------------------------|---|
| <a href="#">BeginExecuteNonQuery</a>  | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> , generally executing commands such as INSERT, DELETE, UPDATE, and SET statements. Each call to <a href="#">BeginExecuteNonQuery</a> must be paired with a call to <a href="#">EndExecuteNonQuery</a> which finishes the operation, typically on a separate thread. |
| <a href="#">BeginExecuteReader</a>    | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> and retrieves one or more results sets from the server. Each call to <a href="#">BeginExecuteReader</a> must be paired with a call to <a href="#">EndExecuteReader</a> which finishes the operation, typically on a separate thread.                                |
| <a href="#">BeginExecuteXmlReader</a> | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> . Each call to <a href="#">BeginExecuteXmlReader</a> must be paired with a call to <a href="#">EndExecuteXmlReader</a> , which finishes the operation, typically on a separate thread, and returns an <a href="#">XmlReader</a> object.                             |
| <a href="#">ExecuteReader</a>         | Executes commands that return rows. For increased performance, <a href="#">ExecuteReader</a> invokes commands using the Transact-SQL <code>sp_executesql</code> system stored procedure. Therefore, <a href="#">ExecuteReader</a> might not have the effect that you want if used to execute commands such as Transact-SQL SET statements.  |

| ITEM                             | DESCRIPTION   |
|----------------------------------|---|
| <a href="#">ExecuteNonQuery</a>  | Executes commands such as Transact-SQL INSERT, DELETE, UPDATE, and SET statements.                                      |
| <a href="#">ExecuteScalar</a>    | Retrieves a single value (for example, an aggregate value) from a database.   |
| <a href="#">ExecuteXmlReader</a> | Sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> and builds an <a href="#">XmlReader</a> object. |

You can reset the [CommandText](#) property and reuse the [SqlCommand](#) object. However, you must close the [SqlDataReader](#) before you can execute a new or previous command.

If a [SqlException](#) is generated by the method executing a [SqlCommand](#), the [SqlConnection](#) remains open when the severity level is 19 or less. When the severity level is 20 or greater, the server ordinarily closes the [SqlConnection](#). However, the user can reopen the connection and continue.

**Note**

Nameless, also called ordinal, parameters are not supported by the .NET Framework Data Provider for SQL Server.

## Constructors

[SqlCommand\(\)](#)

[SqlCommand\(\)](#)

Initializes a new instance of the [SqlCommand](#) class.

[SqlCommand\(String\)](#)

[SqlCommand\(String\)](#)

Initializes a new instance of the [SqlCommand](#) class with the text of the query.

[SqlCommand\(String, SqlConnection\)](#)

[SqlCommand\(String, SqlConnection\)](#)

Initializes a new instance of the [SqlCommand](#) class with the text of the query and a [SqlConnection](#).

[SqlCommand\(String, SqlConnection, SqlTransaction\)](#)

[SqlCommand\(String, SqlConnection, SqlTransaction\)](#)

Initializes a new instance of the [SqlCommand](#) class with the text of the query, a [SqlConnection](#), and the [SqlTransaction](#).

[SqlCommand\(String, SqlConnection, SqlTransaction, SqlCommandColumnEncryptionSetting\)](#)

[SqlCommand\(String, SqlConnection, SqlTransaction, SqlCommandColumnEncryptionSetting\)](#)

Initializes a new instance of the [SqlCommand](#) class with specified command text, connection, transaction, and encryption setting.

## Properties

ColumnEncryptionSetting

ColumnEncryptionSetting

Gets or sets the column encryption setting for this command.

CommandText

CommandText

Gets or sets the Transact-SQL statement, table name or stored procedure to execute at the data source.

CommandTimeout

CommandTimeout

Gets or sets the wait time before terminating the attempt to execute a command and generating an error.

CommandType

CommandType

Gets or sets a value indicating how the [CommandText](#) property is to be interpreted.

Connection

Connection

Gets or sets the [SqlConnection](#) used by this instance of the [SqlCommand](#).

DesignTimeVisible

DesignTimeVisible

Gets or sets a value indicating whether the command object should be visible in a Windows Form Designer control.

Notification

Notification

Gets or sets a value that specifies the [SqlNotificationRequest](#) object bound to this command.

NotificationAutoEnlist

NotificationAutoEnlist

Gets or sets a value indicating whether the application should automatically receive query notifications from a common [SqlDependency](#) object.

Parameters

## Parameters

Gets the [SqlParameterCollection](#).

## Transaction

### Transaction

Gets or sets the [SqlTransaction](#) within which the [SqlCommand](#) executes.

## UpdatedRowSource

### UpdatedRowSource

Gets or sets how command results are applied to the [DataRow](#) when used by the **Update** method of the [DbDataAdapter](#).

## Methods

### BeginExecuteNonQuery()

### BeginExecuteNonQuery()

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#).

### BeginExecuteNonQuery(AsyncCallback, Object)

### BeginExecuteNonQuery(AsyncCallback, Object)

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#), given a callback procedure and state information.

### BeginExecuteReader()

### BeginExecuteReader()

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#), and retrieves one or more result sets from the server.

### BeginExecuteReader(CommandBehavior)

### BeginExecuteReader(CommandBehavior)

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) using one of the [CommandBehavior](#) values.

### BeginExecuteReader(AsyncCallback, Object)

### BeginExecuteReader(AsyncCallback, Object)

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) and retrieves one or more result sets from the server, given a callback procedure and state information.

```
BeginExecuteReader(AsyncCallback, Object, CommandBehavior)
BeginExecuteReader(AsyncCallback, Object, CommandBehavior)
```

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#), using one of the [CommandBehavior](#) values, and retrieving one or more result sets from the server, given a callback procedure and state information.

```
BeginExecuteXmlReader()
BeginExecuteXmlReader()
```

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) and returns results as an [XmlReader](#) object.

```
BeginExecuteXmlReader(AsyncCallback, Object)
BeginExecuteXmlReader(AsyncCallback, Object)
```

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) and returns results as an [XmlReader](#) object, using a callback procedure.

```
Cancel()
Cancel()
```

Tries to cancel the execution of a [SqlCommand](#).

```
Clone()
Clone()
```

Creates a new [SqlCommand](#) object that is a copy of the current instance.

```
CreateParameter()
CreateParameter()
```

Creates a new instance of a [SqlParameter](#) object.

```
EndExecuteNonQuery(IAsyncResult)
EndExecuteNonQuery(IAsyncResult)
```

Finishes asynchronous execution of a Transact-SQL statement.

```
EndExecuteReader(IAsyncResult)
EndExecuteReader(IAsyncResult)
```

Finishes asynchronous execution of a Transact-SQL statement, returning the requested [SqlDataReader](#).

```
EndExecuteXmlReader(IAsyncResult)
```

```
EndExecuteXmlReader(IAsyncResult)
```

Finishes asynchronous execution of a Transact-SQL statement, returning the requested data as XML.

```
ExecuteNonQuery()
```

```
ExecuteNonQuery()
```

Executes a Transact-SQL statement against the connection and returns the number of rows affected.

```
ExecuteNonQueryAsync(CancellationToken)
```

```
ExecuteNonQueryAsync(CancellationToken)
```

An asynchronous version of [ExecuteNonQuery\(\)](#), which executes a Transact-SQL statement against the connection and returns the number of rows affected. The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
ExecuteReader()
```

```
ExecuteReader()
```

Sends the [CommandText](#) to the [Connection](#) and builds a [SqlDataReader](#).

```
ExecuteReader(CommandBehavior)
```

```
ExecuteReader(CommandBehavior)
```

Sends the [CommandText](#) to the [Connection](#), and builds a [SqlDataReader](#) using one of the [CommandBehavior](#) values.

```
ExecuteReaderAsync()
```

```
ExecuteReaderAsync()
```

An asynchronous version of [ExecuteReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds a [SqlDataReader](#). Exceptions will be reported via the returned Task object.

```
ExecuteReaderAsync(CommandBehavior)
```

```
ExecuteReaderAsync(CommandBehavior)
```

An asynchronous version of [ExecuteReader\(CommandBehavior\)](#), which sends the [CommandText](#) to the [Connection](#), and builds a [SqlDataReader](#). Exceptions will be reported via the returned Task object.

```
ExecuteReaderAsync(CancellationToken)
```

```
ExecuteReaderAsync(CancellationToken)
```

An asynchronous version of [ExecuteReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds a [SqlDataReader](#).

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
ExecuteReaderAsync(CommandBehavior, CancellationToken)
ExecuteReaderAsync(CommandBehavior, CancellationToken)
```

An asynchronous version of [ExecuteReader\(CommandBehavior\)](#), which sends the [CommandText](#) to the [Connection](#), and builds a [SqlDataReader](#)

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
ExecuteScalar()
ExecuteScalar()
```

Executes the query, and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.

```
ExecuteScalarAsync(CancellationToken)
ExecuteScalarAsync(CancellationToken)
```

An asynchronous version of [ExecuteScalar\(\)](#), which executes the query asynchronously and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
ExecuteXmlReader()
ExecuteXmlReader()
```

Sends the [CommandText](#) to the [Connection](#) and builds an [XmlReader](#) object.

```
ExecuteXmlReaderAsync()
ExecuteXmlReaderAsync()
```

An asynchronous version of [ExecuteXmlReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds an [XmlReader](#) object.

Exceptions will be reported via the returned Task object.

```
ExecuteXmlReaderAsync(CancellationToken)
ExecuteXmlReaderAsync(CancellationToken)
```

An asynchronous version of [ExecuteXmlReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds an [XmlReader](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
Prepare()
Prepare()
```

Creates a prepared version of the command on an instance of SQL Server.

```
ResetCommandTimeout()  
ResetCommandTimeout()
```

Resets the [CommandTimeout](#) property to its default value.

## Events

```
StatementCompleted
```

```
StatementCompleted
```

Occurs when the execution of a Transact-SQL statement completes.

```
IDbCommand.CreateParameter()  
IDbCommand.CreateParameter()
```

```
IDbCommand.ExecuteReader()  
IDbCommand.ExecuteReader()
```

```
IDbCommand.ExecuteReader(CommandBehavior)  
IDbCommand.ExecuteReader(CommandBehavior)
```

```
ICloneable.Clone()  
ICloneable.Clone()
```

Creates a new [SqlCommand](#) object that is a copy of the current instance.

## See Also

# SqlCommand.BeginExecuteNonQuery SqlCommand.BeginExecuteNonQuery

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">BeginExecuteNonQuery()</a> <a href="#">BeginExecuteNonQuery()</a>   | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> .   |
| <a href="#">BeginExecuteNonQuery(AsyncCallback, Object)</a> <a href="#">BeginExecuteNonQuery(AsyncCallback, Object)</a> | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> , given a callback procedure and state information. |

## BeginExecuteNonQuery() BeginExecuteNonQuery()

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#).

```
public IAsyncResult BeginExecuteNonQuery ();
member this.BeginExecuteNonQuery : unit -> IAsyncResult
```

Returns

[IAsyncResult](#) [IAsyncResult](#)

An [IAsyncResult](#) that can be used to poll or wait for results, or both; this value is also needed when invoking [EndExecuteNonQuery\(IAsyncResult\)](#), which returns the number of affected rows.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException](#) [SQLException](#)

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this [SqlCommand](#).

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see

## [SqlClient Streaming Support](#)

### [IOException IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### [ObjectDisposedException ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### Examples

The following console application creates updates data within the **AdventureWorks** sample database, doing its work asynchronously. In order to emulate a long-running process, this example inserts a WAITFOR statement in the command text. Normally, you would not take efforts to make your commands run slower, but doing this in this case makes it easier to demonstrate the asynchronous behavior.

```
using System.Data.SqlClient;

class Class1
{
    static void Main()
    {
        // This is a simple example that demonstrates the usage of the
        // BeginExecuteNonQuery functionality.
        // The WAITFOR statement simply adds enough time to prove the
        // asynchronous nature of the command.

        string commandText =
            "UPDATE Production.Product SET ReorderPoint = ReorderPoint + 1 " +
            "WHERE ReorderPoint Is Not Null;" +
            "WAITFOR DELAY '0:0:3';" +
            "UPDATE Production.Product SET ReorderPoint = ReorderPoint - 1 " +
            "WHERE ReorderPoint Is Not Null";

        RunCommandAsynchronously(commandText, GetConnectionString());

        Console.WriteLine("Press ENTER to continue.");
        Console.ReadLine();
    }

    private static void RunCommandAsynchronously(
        string commandText, string connectionString)
    {
        // Given command text and connection string, asynchronously execute
        // the specified command against the connection. For this example,
        // the code displays an indicator as it is working, verifying the
        // asynchronous behavior.
        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            try
            {
                int count = 0;
                SqlCommand command = new SqlCommand(commandText, connection);
                connection.Open();

                IAsyncResult result = command.BeginExecuteNonQuery();
                while (!result.IsCompleted)
                {
                    Console.WriteLine("Waiting ({0})", count++);
                    // Wait for 1/10 second, so the counter
                }
            }
        }
    }
}
```

```

        // does not consume all available resources
        // on the main thread.
        System.Threading.Thread.Sleep(100);
    }
    Console.WriteLine("Command complete. Affected {0} rows.",
        command.EndExecuteNonQuery(result));
}
catch (SqlException ex)
{
    Console.WriteLine("Error ({0}): {1}", ex.Number, ex.Message);
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Error: {0}", ex.Message);
}
catch (Exception ex)
{
    // You might want to pass these errors
    // back out to the caller.
    Console.WriteLine("Error: {0}", ex.Message);
}
}

private static string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.

    // If you have not included "Asynchronous Processing=true" in the
    // connection string, the command is not able
    // to execute asynchronously.
    return "Data Source=(local);Integrated Security=SSPI;" +
        "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
}
}

```

## Remarks

The [BeginExecuteNonQuery](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that does not return rows, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteNonQuery](#) method to finish the operation. The [BeginExecuteNonQuery](#) method returns immediately ([CommandTimeout](#) has no effect on [BeginExecuteNonQuery](#)), but until the code executes the corresponding [EndExecuteNonQuery](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteNonQuery](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous.

Because this overload does not support a callback procedure, developers must either poll to determine whether the command has completed, using the [IsCompleted](#) property of the [IAsyncResult](#) returned by the [BeginExecuteNonQuery](#) method; or wait for the completion of one or more commands using the [AsyncWaitHandle](#) property of the returned [IAsyncResult](#).

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## BeginExecuteNonQuery(AsyncCallback, Object)

## BeginExecuteNonQuery(AsyncCallback, Object)

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#), given a callback procedure and state information.

```
public IAsyncResult BeginExecuteNonQuery (AsyncCallback callback, object stateObject);  
member this.BeginExecuteNonQuery : AsyncCallback * obj -> IAsyncResult
```

### Parameters

callback [AsyncCallback AsyncCallback](#)

An  [AsyncCallback](#) delegate that is invoked when the command's execution has completed. Pass `null` (`Nothing`) in Microsoft Visual Basic) to indicate that no callback is required.

stateObject [Object Object](#)

A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the  [AsyncState](#) property.

### Returns

[IAsyncResult IAsyncResult](#)

An [IAsyncResult](#) that can be used to poll or wait for results, or both; this value is also needed when invoking [EndExecuteNonQuery\(IAsyncResult\)](#), which returns the number of affected rows.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

A  [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A  [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A  [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException SQLException](#)

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException InvalidOperationException](#)

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this [SqlCommand](#).

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException IOException](#)

An error occurred in a  [Stream](#),  [XmlReader](#) or  [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following Windows application demonstrates the use of the [BeginExecuteNonQuery](#) method, executing a Transact-SQL statement that includes a delay of several seconds (emulating a long-running command).

This example demonstrates many important techniques. This includes calling a method that interacts with the form from a separate thread. In addition, this example demonstrates how you must block users from executing a command multiple times concurrently, and how you must make sure that the form does not close before the callback procedure is called.

To set up this example, create a new Windows application. Put a [Button](#) control and a [Label](#) control on the form (accepting the default name for each control). Add the following code to the form's class, modifying the connection string as needed for your environment.

```
using System.Data.SqlClient;

namespace Microsoft.AdodotNet.CodeSamples
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // Hook up the form's Load event handler (you can double-click on
        // the form's design surface in Visual Studio), and then add
        // this code to the form's class:
        private void Form1_Load(object sender, EventArgs e)
        {
            this.button1.Click += new System.EventHandler(this.button1_Click);
            this.FormClosing += new System.Windows.Forms.
                FormClosingEventHandler(this.Form1_FormClosing);

        }

        // You need this delegate in order to display text from a thread
        // other than the form's thread. See the HandleCallback
        // procedure for more information.
        // This same delegate matches both the DisplayStatus
        // and DisplayResults methods.
        private delegate void DisplayInfoDelegate(string Text);

        // This flag ensures that the user does not attempt
        // to restart the command or close the form while the
        // asynchronous command is executing.
        private bool isExecuting;

        // This example maintains the connection object
        // externally, so that it is available for closing.
        private SqlConnection connection;

        private static string GetConnectionString()
        {
            // To avoid storing the connection string in your code,
            // you can retrieve it from a configuration file.

            // If you have not included "Asynchronous Processing=true" in the
            // connection string, the command is not able
            // to execute asynchronously.
            //string "Data Source=(local)\Integrated Security=True";
        }
    }
}
```

```

        return "Data Source=(local);Integrated Security=true;" +
               "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
    }

    private void DisplayStatus(string Text)
    {
        this.label1.Text = Text;
    }

    private void DisplayResults(string Text)
    {
        this.label1.Text = Text;
        DisplayStatus("Ready");
    }

    private void Form1_FormClosing(object sender,
                                  System.Windows.Forms.FormClosingEventArgs e)
    {
        if (isExecuting)
        {
            MessageBox.Show(this, "Cannot close the form until " +
                            "the pending asynchronous command has completed. Please wait..."); 
            e.Cancel = true;
        }
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        if (isExecuting)
        {
            MessageBox.Show(this,
                           "Already executing. Please wait until the current query " +
                           "has completed.");
        }
        else
        {
            SqlCommand command = null;
            try
            {
                DisplayResults("");
                DisplayStatus("Connecting...");
                connection = new SqlConnection(GetConnectionString());
                // To emulate a long-running query, wait for
                // a few seconds before working with the data.
                // This command does not do much, but that's the point--
                // it does not change your data, in the long run.
                string commandText =
                    "WAITFOR DELAY '0:0:05';" +
                    "UPDATE Production.Product SET ReorderPoint = ReorderPoint + 1 " +
                    "WHERE ReorderPoint Is Not Null;" +
                    "UPDATE Production.Product SET ReorderPoint = ReorderPoint - 1 " +
                    "WHERE ReorderPoint Is Not Null";

                command = new SqlCommand(commandText, connection);
                connection.Open();

                DisplayStatus("Executing...");
                isExecuting = true;
                // Although it is not required that you pass the
                // SqlCommand object as the second parameter in the
                // BeginExecuteNonQuery call, doing so makes it easier
                // to call EndExecuteNonQuery in the callback procedure.
                AsyncCallback callback = new AsyncCallback(HandleCallback);
                command.BeginExecuteNonQuery(callback, command);
            }
        }
    }
}

```

```

        catch (Exception ex)
    {
        isExecuting = false;
        DisplayStatus(string.Format("Ready (last error: {0})", ex.Message));
        if (connection != null)
        {
            connection.Close();
        }
    }
}

private void HandleCallback(IAsyncResult result)
{
    try
    {
        // Retrieve the original command object, passed
        // to this procedure in the AsyncState property
        // of the IAsyncResult parameter.
        SqlCommand command = (SqlCommand)result.AsyncState;
        int rowCount = command.EndExecuteNonQuery(result);
        string rowText = " rows affected.";
        if (rowCount == 1)
        {
            rowText = " row affected.";
        }
        rowText = rowCount + rowText;

        // You may not interact with the form and its contents
        // from a different thread, and this callback procedure
        // is all but guaranteed to be running from a different thread
        // than the form. Therefore you cannot simply call code that
        // displays the results, like this:
        // DisplayResults(rowText)

        // Instead, you must call the procedure from the form's thread.
        // One simple way to accomplish this is to call the Invoke
        // method of the form, which calls the delegate you supply
        // from the form's thread.
        DisplayInfoDelegate del = new DisplayInfoDelegate(DisplayResults);
        this.Invoke(del, rowText);

    }
    catch (Exception ex)
    {
        // Because you are now running code in a separate thread,
        // if you do not handle the exception here, none of your other
        // code catches the exception. Because none of
        // your code is on the call stack in this thread, there is nothing
        // higher up the stack to catch the exception if you do not
        // handle it here. You can either log the exception or
        // invoke a delegate (as in the non-error case in this
        // example) to display the error on the form. In no case
        // can you simply display the error without executing a delegate
        // as in the try block here.

        // You can create the delegate instance as you
        // invoke it, like this:
        this.Invoke(new DisplayInfoDelegate(DisplayStatus),
            String.Format("Ready(last error: {0})", ex.Message));
    }
    finally
    {
        isExecuting = false;
        if (connection != null)

```

```
        {
            connection.Close();
        }
    }
}
```

## Remarks

The [BeginExecuteNonQuery](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that does not return rows, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteNonQuery](#) method to finish the operation. The [BeginExecuteNonQuery](#) method returns immediately ([CommandTimeout](#) has no effect on [BeginExecuteNonQuery](#)), but until the code executes the corresponding [EndExecuteNonQuery](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteNonQuery](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

The `callback` parameter lets you specify an [AsyncCallback](#) delegate that is called when the statement has completed. You can call the [EndExecuteNonQuery](#) method from within this delegate procedure, or from any other location within your application. In addition, you can pass any object in the `asyncStateObject` parameter, and your callback procedure can retrieve this information using the [AsyncResult](#) property.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous.

Because the callback procedure executes from within a background thread supplied by the Microsoft .NET common language runtime, it is very important that you take a rigorous approach to handling cross-thread interactions from within your applications. For example, you must not interact with a form's contents from within your callback procedure; should you have to update the form, you must switch back to the form's thread in order to do your work. The example in this topic demonstrates this behavior.

All errors that occur during the execution of the operation are thrown as exceptions in the callback procedure. You must handle the exception in the callback procedure, not in the main application. See the example in this topic for additional information on handling exceptions in the callback procedure.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.BeginExecuteReader SqlCommand.BeginExecuteReader

In this Article

## Overloads

|   |  |
|---|--|
| <a href="#">BeginExecuteReader()</a> <a href="#">BeginExecuteReader()</a>   | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> , and retrieves one or more result sets from the server.   |
| <a href="#">BeginExecuteReader(CommandBehavior)</a> <a href="#">BeginExecuteReader(CommandBehavior)</a>   | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> using one of the <a href="#">CommandBehavior</a> values.   |
| <a href="#">BeginExecuteReader(AsyncCallback, Object)</a> <a href="#">BeginExecuteReader(AsyncCallback, Object)</a>                                   | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> and retrieves one or more result sets from the server, given a callback procedure and state information.   |
| <a href="#">BeginExecuteReader(AsyncCallback, Object, CommandBehavior)</a> <a href="#">BeginExecuteReader(AsyncCallback, Object, CommandBehavior)</a> | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> , using one of the <a href="#">CommandBehavior</a> values, and retrieving one or more result sets from the server, given a callback procedure and state information. |

## BeginExecuteReader() BeginExecuteReader()

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#), and retrieves one or more result sets from the server.

```
public IAsyncResult BeginExecuteReader ();
member this.BeginExecuteReader : unit -> IAsyncResult
```

Returns

[IAsyncResult](#) [IAsyncResult](#)

An [IAsyncResult](#) that can be used to poll or wait for results, or both; this value is also needed when invoking [EndExecuteReader\(IAsyncResult\)](#), which returns a [SqlDataReader](#) instance that can be used to retrieve the returned rows.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A `SqlDbType` other than **Xml** was used when `Value` was set to `XmlReader`.

### SQLException SQLException

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### InvalidOperationException InvalidOperationException

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this `SqlCommand`.

The `SqlConnection` closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### IOException IOException

An error occurred in a `Stream`, `XmlReader` or `TextReader` object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### ObjectDisposedException ObjectDisposedException

The `Stream`, `XmlReader` or `TextReader` object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following console application starts the process of retrieving a data reader asynchronously. While waiting for the results, this simple application sits in a loop, investigating the `IsCompleted` property value. As soon as the process has completed, the code retrieves the `SqlDataReader` and displays its contents.

```
using System.Data.SqlClient;

class Class1
{
    static void Main()
    {
        // This is a simple example that demonstrates the usage of the
        // BeginExecuteReader functionality
        // The WAITFOR statement simply adds enough time to prove the
        // asynchronous nature of the command.
        string commandText =
            "WAITFOR DELAY '00:00:03';" +
            "SELECT LastName, FirstName FROM Person.Contact " +
            "WHERE LastName LIKE 'M%'"';

        RunCommandAsynchronously(commandText, GetConnectionString());

        Console.WriteLine("Press ENTER to continue.");
        Console.ReadLine();
    }

    private static void RunCommandAsynchronously(
        string commandText, string connectionString)
    {
        // Given command text and connection string, asynchronously execute
        // the specified command against the connection. For this example,
        // the code displays an indicator as it is working, verifying the
        // asynchronous behavior.
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
```

```

try
{
    SqlCommand command = new SqlCommand(commandText, connection);

    connection.Open();
    IAsyncResult result = command.BeginExecuteReader();

    // Although it is not necessary, the following code
    // displays a counter in the console window, indicating that
    // the main thread is not blocked while awaiting the command
    // results.
    int count = 0;
    while (!result.IsCompleted)
    {
        count += 1;
        Console.WriteLine("Waiting ({0})", count);
        // Wait for 1/10 second, so the counter
        // does not consume all available resources
        // on the main thread.
        System.Threading.Thread.Sleep(100);
    }

    using (SqlDataReader reader = command.EndExecuteReader(result))
    {
        DisplayResults(reader);
    }
}
catch (SqlException ex)
{
    Console.WriteLine("Error ({0}): {1}", ex.Number, ex.Message);
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Error: {0}", ex.Message);
}
catch (Exception ex)
{
    // You might want to pass these errors
    // back out to the caller.
    Console.WriteLine("Error: {0}", ex.Message);
}
}

private static void DisplayResults(SqlDataReader reader)
{
    // Display the data within the reader.
    while (reader.Read())
    {
        // Display all the columns.
        for (int i = 0; i < reader.FieldCount; i++)
            Console.Write("{0} ", reader.GetValue(i));
        Console.WriteLine();
    }
}

private static string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.

    // If you have not included "Asynchronous Processing=true" in the
    // connection string, the command is not able
    // to execute asynchronously.
    return "Data Source=(local);Integrated Security=true;" +
        "MultipleActiveResultSets=True; Asynchronous Processing=True";
}

```

```
    Initial Catalog=Adventureworks; Asyncnronous Processing=true";
}
```

## Remarks

The [BeginExecuteReader](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that returns rows, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteReader](#) method to finish the operation and retrieve the [SqlDataReader](#) returned by the command. The [BeginExecuteReader](#) method returns immediately, but until the code executes the corresponding [EndExecuteReader](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteReader](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous. Although command execution is asynchronous, value fetching is still synchronous. This means that calls to [Read](#) may block if more data is required and the underlying network's read operation blocks.

Because this overload does not support a callback procedure, developers must either poll to determine whether the command has completed, using the [IsCompleted](#) property of the [IAsyncResult](#) returned by the [BeginExecuteReader](#) method; or wait for the completion of one or more commands using the [AsyncWaitHandle](#) property of the returned [IAsyncResult](#).

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server will return any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## BeginExecuteReader(CommandBehavior)

## BeginExecuteReader(CommandBehavior)

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) using one of the [CommandBehavior](#) values.

```
public IAsyncResult BeginExecuteReader (System.Data.CommandBehavior behavior);
member this.BeginExecuteReader : System.Data.CommandBehavior -> IAsyncResult
```

Parameters

behavior

[CommandBehavior](#) [CommandBehavior](#)

One of the [CommandBehavior](#) values, indicating options for statement execution and data retrieval.

Returns

[IAsyncResult](#) [IAsyncResult](#)

An [IAsyncResult](#) that can be used to poll, wait for results, or both; this value is also needed when invoking [EndExecuteReader\(IAsyncResult\)](#), which returns a [SqlDataReader](#) instance that can be used to retrieve the returned rows.

Exceptions

## InvalidOperationException InvalidOperationException

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

## SqlException SqlException

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## InvalidOperationException InvalidOperationException

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this [SqlCommand](#).

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## IOException IOException

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## ObjectDisposedException ObjectDisposedException

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following console application starts the process of retrieving a data reader asynchronously. While waiting for the results, this simple application sits in a loop, investigating the [IsCompleted](#) property value. Once the process has completed, the code retrieves the [SqlDataReader](#) and displays its contents.

This example also passes the [CommandBehavior.CloseConnection](#) and [CommandBehavior.SingleRow](#) values in the behavior parameter, causing the connection to be closed with the returned [SqlDataReader](#) is closed, and to optimize for a single row result.

```
using System.Data.SqlClient;
class Class1
{
    static void Main()
    {
        // This example is not terribly useful, but it proves a point.
        // The WAITFOR statement simply adds enough time to prove the
        // asynchronous nature of the command.
        string commandText = "WAITFOR DELAY '00:00:03';" +
            "SELECT ProductID, Name FROM Production.Product WHERE ListPrice < 100";

        RunCommandAsynchronously(commandText, GetConnectionString());

        Console.WriteLine("Press ENTER to continue.");
        Console.ReadLine();
    }

    private static void RunCommandAsynchronously(
```

```
string commandText, string connectionString)
{
    // Given command text and connection string, asynchronously execute
    // the specified command against the connection. For this example,
    // the code displays an indicator as it is working, verifying the
    // asynchronous behavior.

    try
    {
        // The code does not need to handle closing the connection explicitly--
        // the use of the CommandBehavior.CloseConnection option takes care
        // of that for you.
        SqlConnection connection = new SqlConnection(connectionString);
        SqlCommand command = new SqlCommand(commandText, connection);

        connection.Open();
        IAsyncResult result = command.BeginExecuteReader(
            CommandBehavior.CloseConnection);

        // Although it is not necessary, the following code
        // displays a counter in the console window, indicating that
        // the main thread is not blocked while awaiting the command
        // results.
        int count = 0;
        while (!result.IsCompleted)
        {
            Console.WriteLine("Waiting ({0})", count++);
            // Wait for 1/10 second, so the counter
            // does not consume all available resources
            // on the main thread.
            System.Threading.Thread.Sleep(100);
        }

        using (SqlDataReader reader = command.EndExecuteReader(result))
        {
            DisplayResults(reader);
        }
    }
    catch (SqlException ex)
    {
        Console.WriteLine("Error ({0}): {1}", ex.Number, ex.Message);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("Error: {0}", ex.Message);
    }
    catch (Exception ex)
    {
        // You might want to pass these errors
        // back out to the caller.
        Console.WriteLine("Error: {0}", ex.Message);
    }
}

private static void DisplayResults(SqlDataReader reader)
{
    // Display the data within the reader.
    while (reader.Read())
    {
        // Display all the columns.
        for (int i = 0; i < reader.FieldCount; i++)
        {
            Console.Write("{0}    ", reader.GetValue(i));
        }
        Console.WriteLine();
    }
}
```

```

        }

        private static string GetConnectionString()
        {
            // To avoid storing the connection string in your code,
            // you can retrieve it from a configuration file.

            // If you have not included "Asynchronous Processing=true" in the
            // connection string, the command is not able
            // to execute asynchronously.
            return "Data Source=(local);Integrated Security=true;" +
                "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
        }
    }
}

```

## Remarks

The [BeginExecuteReader](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that returns rows, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteReader](#) method to finish the operation and retrieve the [SqlDataReader](#) returned by the command. The [BeginExecuteReader](#) method returns immediately, but until the code executes the corresponding [EndExecuteReader](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteReader](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

The `behavior` parameter lets you specify options that control the behavior of the command and its connection. These values can be combined together (using the programming language's `OR` operator); generally, developers use the `CommandBehavior.CloseConnection` value to make sure that the connection is closed by the runtime when the [SqlDataReader](#) is closed.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous. Although command execution is asynchronous, value fetching is still synchronous. This means that calls to [Read](#) may block if more data is required and the underlying network's read operation blocks.

Because this overload does not support a callback procedure, developers must either poll to determine whether the command has completed, using the [IsCompleted](#) property of the [IAsyncResult](#) returned by the [BeginExecuteNonQuery](#) method; or wait for the completion of one or more commands using the [AsyncWaitHandle](#) property of the returned [IAsyncResult](#).

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server returns any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## **BeginExecuteReader(AsyncCallback, Object)** **BeginExecuteReader(AsyncCallback, Object)**

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) and retrieves one or more result sets from the server, given a callback procedure and state information.

```
public IAsyncResult BeginExecuteReader (AsyncCallback callback, object stateObject);  
member this.BeginExecuteReader : AsyncCallback * obj -> IAsyncResult
```

## Parameters

callback [AsyncCallback AsyncCallback](#)

An  [AsyncCallback](#) delegate that is invoked when the command's execution has completed. Pass `null` (`Nothing`) in Microsoft Visual Basic) to indicate that no callback is required.

stateObject [Object Object](#)

A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the  [AsyncState](#) property.

## Returns

[IAsyncResult IAsyncResult](#)

An  [IAsyncResult](#) that can be used to poll, wait for results, or both; this value is also needed when invoking [EndExecuteReader\(IAsyncResult\)](#), which returns a  [SqlDataReader](#) instance which can be used to retrieve the returned rows.

## Exceptions

[InvalidOperationException InvalidOperationException](#)

A  [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A  [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A  [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SqlException SqlException](#)

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException InvalidOperationException](#)

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this  [SqlCommand](#).

The  [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException IOException](#)

An error occurred in a  [Stream](#),  [XmlReader](#) or  [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException ObjectDisposedException](#)

The  [Stream](#),  [XmlReader](#) or  [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following Windows application demonstrates the use of the  [BeginExecuteReader](#) method, executing a Transact-

SQL statement that includes a delay of a few seconds (emulating a long-running command). Because the sample executes the command asynchronously, the form remains responsive while awaiting the results. This example passes the executing `SqlCommand` object as the `stateObject` parameter; doing so makes it simple to retrieve the `SqlCommand` object from within the callback procedure, so that the code can call the `EndExecuteReader` method corresponding to the initial call to `BeginExecuteReader`.

This example demonstrates many important techniques. This includes calling a method that interacts with the form from a separate thread. In addition, this example demonstrates how you must block users from executing a command multiple times concurrently, and how you must make sure that the form does not close before the callback procedure is called.

To set up this example, create a new Windows application. Put a `Button` control, a `DataGridView` control, and a `Label` control on the form (accepting the default name for each control). Add the following code to the form's class, modifying the connection string as needed for your environment.

```
using System.Data.SqlClient;

namespace Microsoft.AdodotNet.CodeSamples
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        // Hook up the form's Load event handler (you can double-click on
        // the form's design surface in Visual Studio), and then add
        // this code to the form's class:

        // You need this delegate in order to fill the grid from
        // a thread other than the form's thread. See the HandleCallback
        // procedure for more information.
        private delegate void FillGridDelegate(SqlDataReader reader);

        // You need this delegate to update the status bar.
        private delegate void DisplayStatusDelegate(string Text);

        // This flag ensures that the user does not attempt
        // to restart the command or close the form while the
        // asynchronous command is executing.
        private bool isExecuting = false;

        // Because the overloaded version of BeginExecuteReader
        // demonstrated here does not allow you to have the connection
        // closed automatically, this example maintains the
        // connection object externally, so that it is available for closing.
        private SqlConnection connection = null;

        private void DisplayStatus(string Text)
        {
            this.label1.Text = Text;
        }

        private void FillGrid(SqlDataReader reader)
        {
            try
            {
                DataTable table = new DataTable();
                table.Load(reader);
                this.dataGridView1.DataSource = table;
                DisplayStatus("Ready");
            }
        }
    }
}
```

```

        catch (Exception ex)
    {
        // Because you are guaranteed this procedure
        // is running from within the form's thread,
        // it can directly interact with members of the form.
        DisplayStatus(string.Format("Ready (last attempt failed: {0})",
            ex.Message));
    }
    finally
    {
        // Do not forget to close the connection, as well.
        if (reader != null)
        {
            reader.Close();
        }
        if (connection != null)
        {
            connection.Close();
        }
    }
}

private void HandleCallback(IAsyncResult result)
{
    try
    {
        // Retrieve the original command object, passed
        // to this procedure in the AsyncState property
        // of the IAsyncResult parameter.
        SqlCommand command = (SqlCommand)result.AsyncState;
        SqlDataReader reader = command.EndExecuteReader(result);
        // You may not interact with the form and its contents
        // from a different thread, and this callback procedure
        // is all but guaranteed to be running from a different thread
        // than the form. Therefore you cannot simply call code that
        // fills the grid, like this:
        // FillGrid(reader);
        // Instead, you must call the procedure from the form's thread.
        // One simple way to accomplish this is to call the Invoke
        // method of the form, which calls the delegate you supply
        // from the form's thread.
        FillGridDelegate del = new FillGridDelegate(FillGrid);
        this.Invoke(del, reader);
        // Do not close the reader here, because it is being used in
        // a separate thread. Instead, have the procedure you have
        // called close the reader once it is done with it.
    }
    catch (Exception ex)
    {
        // Because you are now running code in a separate thread,
        // if you do not handle the exception here, none of your other
        // code catches the exception. Because there is none of
        // your code on the call stack in this thread, there is nothing
        // higher up the stack to catch the exception if you do not
        // handle it here. You can either log the exception or
        // invoke a delegate (as in the non-error case in this
        // example) to display the error on the form. In no case
        // can you simply display the error without executing a delegate
        // as in the try block here.
        // You can create the delegate instance as you
        // invoke it, like this:
        this.Invoke(new DisplayStatusDelegate(DisplayStatus),
            "Error: " + ex.Message);
    }
    finally

```

```

        {
            isExecuting = false;
        }
    }

    private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.

        // If you do not include the Asynchronous Processing=true name/value pair,
        // you will not be able to execute the command asynchronously.
        return "Data Source=(local);Integrated Security=true;" +
            "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        if (isExecuting)
        {
            MessageBox.Show(this,
                "Already executing. Please wait until the current query " +
                "has completed.");
        }
        else
        {
            SqlCommand command = null;
            try
            {
                DisplayStatus("Connecting...");
                connection = new SqlConnection(GetConnectionString());
                // To emulate a long-running query, wait for
                // a few seconds before retrieving the real data.
                command = new SqlCommand("WAITFOR DELAY '0:0:5';" +
                    "SELECT ProductID, Name, ListPrice, Weight FROM Production.Product",
                    connection);
                connection.Open();

                DisplayStatus("Executing...");
                isExecuting = true;
                // Although it is not required that you pass the
                // SqlCommand object as the second parameter in the
                // BeginExecuteReader call, doing so makes it easier
                // to call EndExecuteReader in the callback procedure.
                AsyncCallback callback = new AsyncCallback(HandleCallback);
                command.BeginExecuteReader(callback, command);
            }
            catch (Exception ex)
            {
                DisplayStatus("Error: " + ex.Message);
                if (connection != null)
                {
                    connection.Close();
                }
            }
        }
    }

    private void Form1_Load(object sender, System.EventArgs e)
    {
        this.button1.Click += new System.EventHandler(this.button1_Click);
        this.FormClosing += new FormClosingEventHandler(Form1_FormClosing);
    }

    void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {

```

```
        if (isExecuting)
    {
        MessageBox.Show(this, "Cannot close the form until " +
            "the pending asynchronous command has completed. Please wait...");  
        e.Cancel = true;
    }
}
}
```

## Remarks

The [BeginExecuteReader](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that returns rows, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteReader](#) method to finish the operation and retrieve the [SqlDataReader](#) returned by the command. The [BeginExecuteReader](#) method returns immediately, but until the code executes the corresponding [EndExecuteReader](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteReader](#) before the command's execution is completed cause the [SqlCommand](#) object to block until the execution is finished.

The `callback` parameter lets you specify an [AsyncCallback](#) delegate that is called when the statement has completed. You can call the [EndExecuteReader](#) method from within this delegate procedure, or from any other location within your application. In addition, you can pass any object in the `stateObject` parameter, and your callback procedure can retrieve this information using the [AsyncResult](#) property.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous. Although command execution is asynchronous, value fetching is still synchronous. This means that calls to [Read](#) may block if more data is required and the underlying network's read operation blocks.

Because the callback procedure executes from within a background thread supplied by the Microsoft .NET runtime, it is very important that you take a rigorous approach to handling cross-thread interactions from within your applications. For example, you must not interact with a form's contents from within your callback procedure; should you have to update the form, you must switch back to the form's thread in order to do your work. The example in this topic demonstrates this behavior.

All errors that occur during the execution of the operation are thrown as exceptions in the callback procedure. You must handle the exception in the callback procedure, not in the main application. See the example in this topic for additional information on handling exceptions in the callback procedure.

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server returns any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **BeginExecuteReader(AsyncCallback, Object, CommandBehavior)**

## **BeginExecuteReader(AsyncCallback, Object, CommandBehavior)**

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#), using one of the `CommandBehavior` values, and retrieving one or more result sets from the server, given a callback procedure and state information.

```
public IAsyncResult BeginExecuteReader (AsyncCallback callback, object stateObject,
System.Data.CommandBehavior behavior);

member this.BeginExecuteReader : AsyncCallback * obj * System.Data.CommandBehavior -> IAsyncResult
```

## Parameters

callback [AsyncCallback](#)  [AsyncCallback](#)

An  [AsyncCallback](#) delegate that is invoked when the command's execution has completed. Pass `null` (`Nothing` in Microsoft Visual Basic) to indicate that no callback is required.

stateObject [Object](#)  [Object](#)

A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the  [AsyncState](#) property.

behavior [CommandBehavior](#)  [CommandBehavior](#)

One of the  [CommandBehavior](#) values, indicating options for statement execution and data retrieval.

## Returns

[IAsyncResult](#)  [IAsyncResult](#)

An  [IAsyncResult](#) that can be used to poll or wait for results, or both; this value is also needed when invoking  [EndExecuteReader\(IAsyncResult\)](#), which returns a  [SqlDataReader](#) instance which can be used to retrieve the returned rows.

## Exceptions

[InvalidOperationException](#)  [InvalidOperationException](#)

A  [SqlDbType](#) other than **Binary** or **VarBinary** was used when  [Value](#) was set to  [Stream](#). For more information about streaming, see  [SqlCommand Streaming Support](#).

A  [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when  [Value](#) was set to  [TextReader](#).

A  [SqlDbType](#) other than **Xml** was used when  [Value](#) was set to  [XmlReader](#).

[SqlException](#)  [SqlException](#)

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see  [SqlCommand Streaming Support](#).

[InvalidOperationException](#)  [InvalidOperationException](#)

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this  [SqlCommand](#).

The  [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see  [SqlCommand Streaming Support](#).

[IOException](#)  [IOException](#)

An error occurred in a  [Stream](#),  [XmlReader](#) or  [TextReader](#) object during a streaming operation. For more information about streaming, see  [SqlCommand Streaming Support](#).

[ObjectDisposedException](#)  [ObjectDisposedException](#)

The  [Stream](#),  [XmlReader](#) or  [TextReader](#) object was closed during a streaming operation. For more information about

streaming, see [SqlClient Streaming Support](#).

## Examples

The following Windows application demonstrates the use of the `BeginExecuteReader` method, executing a Transact-SQL statement that includes a delay of a few seconds (emulating a long-running command). Because the sample executes the command asynchronously, the form remains responsive while awaiting the results. This example passes the executing `SqlCommand` object as the `stateObject` parameter; doing so makes it simple to retrieve the `SqlCommand` object from within the callback procedure, so that the code can call the `EndExecuteReader` method corresponding to the initial call to `BeginExecuteReader`.

This example demonstrates many important techniques. This includes calling a method that interacts with the form from a separate thread. In addition, this example demonstrates how you must block users from executing a command multiple times concurrently, and how you must make sure that the form does not close before the callback procedure is called.

To set up this example, create a new Windows application. Put a `Button` control, a `DataGridView` control, and a `Label` control on the form (accepting the default name for each control). Add the following code to the form's class, modifying the connection string as needed for your environment.

This example passes the `CommandBehavior.CloseConnection` value in the `behavior` parameter, causing the returned `SqlDataReader` to automatically close its connection when it is closed.

```
using System.Data.SqlClient;

namespace Microsoft.AdodotNet.CodeSamples
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        // Hook up the form's Load event handler (you can double-click on
        // the form's design surface in Visual Studio), and then add
        // this code to the form's class:
        // You need this delegate in order to fill the grid from
        // a thread other than the form's thread. See the HandleCallback
        // procedure for more information.
        private delegate void FillGridDelegate(SqlDataReader reader);

        // You need this delegate to update the status bar.
        private delegate void DisplayStatusDelegate(string Text);

        // This flag ensures that the user does not attempt
        // to restart the command or close the form while the
        // asynchronous command is executing.
        private bool isExecuting;

        private void DisplayStatus(string Text)
        {
            this.label1.Text = Text;
        }

        private void FillGrid(SqlDataReader reader)
        {
            try
            {
                DataTable table = new DataTable();
                table.Load(reader);
                this.dataGridView1.DataSource = table;
                DisplayStatus("Ready");
            }
        }
    }
}
```

```

        }

        catch (Exception ex)
        {
            // Because you are guaranteed this procedure
            // is running from within the form's thread,
            // it can directly interact with members of the form.
            DisplayStatus(string.Format("Ready (last attempt failed: {0})",
                ex.Message));
        }

        finally
        {
            // Closing the reader also closes the connection,
            // because this reader was created using the
            // CommandBehavior.CloseConnection value.
            if (reader != null)
            {
                reader.Close();
            }
        }
    }

    private void HandleCallback(IAsyncResult result)
    {
        try
        {
            // Retrieve the original command object, passed
            // to this procedure in the AsyncState property
            // of the IAsyncResult parameter.
            SqlCommand command = (SqlCommand)result.AsyncState;
            SqlDataReader reader = command.EndExecuteReader(result);
            // You may not interact with the form and its contents
            // from a different thread, and this callback procedure
            // is all but guaranteed to be running from a different thread
            // than the form. Therefore you cannot simply call code that
            // fills the grid, like this:
            // FillGrid(reader);
            // Instead, you must call the procedure from the form's thread.
            // One simple way to accomplish this is to call the Invoke
            // method of the form, which calls the delegate you supply
            // from the form's thread.
            FillGridDelegate del = new FillGridDelegate(FillGrid);
            this.Invoke(del, reader);
            // Do not close the reader here, because it is being used in
            // a separate thread. Instead, have the procedure you have
            // called close the reader once it is done with it.
        }
        catch (Exception ex)
        {
            // Because you are now running code in a separate thread,
            // if you do not handle the exception here, none of your other
            // code catches the exception. Because there is none of
            // your code on the call stack in this thread, there is nothing
            // higher up the stack to catch the exception if you do not
            // handle it here. You can either log the exception or
            // invoke a delegate (as in the non-error case in this
            // example) to display the error on the form. In no case
            // can you simply display the error without executing a delegate
            // as in the try block here.
            // You can create the delegate instance as you
            // invoke it, like this:
            this.Invoke(new DisplayStatusDelegate(DisplayStatus), "Error: " +
                ex.Message);
        }

        finally
        {
            isExecuting = false;
        }
    }
}

```

```

        }

    }

    private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.

        // If you do not include the Asynchronous Processing=true name/value pair,
        // you will not be able to execute the command asynchronously.
        return "Data Source=(local);Integrated Security=true;" +
            "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        if (isExecuting)
        {
            MessageBox.Show(this,
                "Already executing. Please wait until the current query " +
                "has completed.");
        }
        else
        {
            SqlCommand command = null;
            SqlConnection connection = null;
            try
            {
                DisplayStatus("Connecting...");
                connection = new SqlConnection(GetConnectionString());
                // To emulate a long-running query, wait for
                // a few seconds before retrieving the real data.
                command = new SqlCommand("WAITFOR DELAY '0:0:5';" +
                    "SELECT ProductID, Name, ListPrice, Weight FROM Production.Product",
                    connection);
                connection.Open();

                DisplayStatus("Executing...");
                isExecuting = true;
                // Although it is not required that you pass the
                // SqlCommand object as the second parameter in the
                // BeginExecuteReader call, doing so makes it easier
                // to call EndExecuteReader in the callback procedure.
                AsyncCallback callback = new AsyncCallback(HandleCallback);
                command.BeginExecuteReader(callback, command,
                    CommandBehavior.CloseConnection);
            }
            catch (Exception ex)
            {
                DisplayStatus("Error: " + ex.Message);
                if (connection != null)
                {
                    connection.Close();
                }
            }
        }
    }

    private void Form1_Load(object sender, System.EventArgs e)
    {
        this.button1.Click += new System.EventHandler(this.button1_Click);
        this.FormClosing += new FormClosingEventHandler(Form1_FormClosing);
    }

    void Form1_FormClosing(object sender, FormClosingEventArgs e)

```

```

    {
        if (isExecuting)
        {
            MessageBox.Show(this, "Cannot close the form until " +
                "the pending asynchronous command has completed. Please wait..."); 
            e.Cancel = true;
        }
    }
}

```

## Remarks

The [BeginExecuteReader](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that returns rows, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteReader](#) method to finish the operation and retrieve the [SqlDataReader](#) returned by the command. The [BeginExecuteReader](#) method returns immediately, but until the code executes the corresponding [EndExecuteReader](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteReader](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

The `callback` parameter lets you specify an [AsyncCallback](#) delegate that is called when the statement has completed. You can call the [EndExecuteReader](#) method from within this delegate procedure, or from any other location within your application. In addition, you can pass any object in the `stateObject` parameter, and your callback procedure can retrieve this information using the [AsyncResult](#) property.

The `behavior` parameter lets you specify options that control the behavior of the command and its connection. These values can be combined together (using the programming language's `Or` operator); generally, developers use the `CloseConnection` value to make sure that the connection is closed by the runtime when the [SqlDataReader](#) is closed. Developers can also optimize the behavior of the [SqlDataReader](#) by specifying the `SingleRow` value when it is known in advance that the Transact-SQL statement or stored procedure only returns a single row.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous. Although command execution is asynchronous, value fetching is still synchronous. This means that calls to [Read](#) may block if more data is required and the underlying network's read operation blocks.

Because the callback procedure executes from within a background thread supplied by the Microsoft .NET common language runtime, it is very important that you take a rigorous approach to handling cross-thread interactions from within your applications. For example, you must not interact with a form's contents from within your callback procedure--should you have to update the form, you must switch back to the form's thread in order to do your work. The example in this topic demonstrates this behavior.

All errors that occur during the execution of the operation are thrown as exceptions in the callback procedure. You must handle the exception in the callback procedure, not in the main application. See the example in this topic for additional information on handling exceptions in the callback procedure.

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server will return any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

Also

[Connecting and Retrieving Data in ADO.NET](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.BeginExecuteXmlReader SqlCommand.BeginExecuteXmlReader

In this Article

## Overloads

|   |   |
|---|---|
| <code>BeginExecuteXmlReader()</code> <code>BeginExecuteXmlReader()</code>   | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> and returns results as an <a href="#">XmlReader</a> object.                             |
| <code>BeginExecuteXmlReader(AsyncCallback, Object)</code> <code>BeginExecuteXmlReader(AsyncCallback, Object)</code> | Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this <a href="#">SqlCommand</a> and returns results as an <a href="#">XmlReader</a> object, using a callback procedure. |

## BeginExecuteXmlReader() BeginExecuteXmlReader()

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) and returns results as an [XmlReader](#) object.

```
public IAsyncResult BeginExecuteXmlReader ();
member this.BeginExecuteXmlReader : unit -> IAsyncResult
```

Returns

[IAsyncResult](#) [IAsyncResult](#)

An [IAsyncResult](#) that can be used to poll or wait for results, or both; this value is also needed when invoking [EndExecuteXmlReader](#), which returns a single XML value.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to **Stream**. For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException](#) [SQLException](#)

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this [SqlCommand](#).

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see

## [SqlClient Streaming Support](#)

### [IOException IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### [ObjectDisposedException ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### Examples

The following console application starts the process of retrieving XML data asynchronously. While waiting for the results, this simple application sits in a loop, investigating the [IsCompleted](#) property value. Once the process has completed, the code retrieves the XML and displays its contents.

```
using System.Data.SqlClient;
using System.Xml;

class Class1
{
    static void Main()
    {
        // This example is not terribly effective, but it proves a point.
        // The WAITFOR statement simply adds enough time to prove the
        // asynchronous nature of the command.
        string commandText =
            "WAITFOR DELAY '00:00:03';" +
            "SELECT Name, ListPrice FROM Production.Product " +
            "WHERE ListPrice < 100 " +
            "FOR XML AUTO, XMLDATA";

        RunCommandAsynchronously(commandText, GetConnectionString());

        Console.WriteLine("Press ENTER to continue.");
        Console.ReadLine();
    }

    private static void RunCommandAsynchronously(
        string commandText, string connectionString)
    {
        // Given command text and connection string, asynchronously execute
        // the specified command against the connection. For this example,
        // the code displays an indicator as it is working, verifying the
        // asynchronous behavior.
        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(commandText, connection);

            connection.Open();
            IAsyncResult result = command.BeginExecuteXmlReader();

            // Although it is not necessary, the following procedure
            // displays a counter in the console window, indicating that
            // the main thread is not blocked while awaiting the command
            // results.
            int count = 0;
            while (!result.IsCompleted)
            {
                Console.WriteLine("Waiting ({0})", count++);
                // Wait for 1/10 second, so the counter
                // does not consume all available resources
            }
        }
    }
}
```

```

        // on the main thread.
        System.Threading.Thread.Sleep(100);
    }

    XmlReader reader = command.EndExecuteXmlReader(result);
    DisplayProductInfo(reader);
}
}

private static void DisplayProductInfo(XmlReader reader)
{
    // Display the data within the reader.
    while (reader.Read())
    {
        // Skip past items that are not from the correct table.
        if (reader.LocalName.ToString() == "Production.Product")
        {
            Console.WriteLine("{0}: {1:C}",
                reader["Name"], Convert.ToSingle(reader["ListPrice"]));
        }
    }
}

private static string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.

    // If you have not included "Asynchronous Processing=true" in the
    // connection string, the command is not able
    // to execute asynchronously.
    return "Data Source=(local);Integrated Security=true;" +
        "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
}
}

```

## Remarks

The [BeginExecuteXmlReader](#) method starts the process of asynchronously executing a Transact-SQL statement that returns rows as XML, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteXmlReader](#) method to finish the operation and retrieve the XML returned by the command. The [BeginExecuteXmlReader](#) method returns immediately, but until the code executes the corresponding [EndExecuteXmlReader](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteXmlReader](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

The [CommandText](#) property ordinarily specifies a Transact-SQL statement with a valid FOR XML clause. However, [CommandText](#) can also specify a statement that returns [ntext](#) data that contains valid XML.

A typical [BeginExecuteXmlReader](#) query can be formatted as in the following C# example:

```
SqlCommand command = new SqlCommand("SELECT ContactID, FirstName, LastName FROM dbo.Contact FOR XML AUTO, XMLDATA", SqlConn);
```

This method can also be used to retrieve a single-row, single-column result set. In this case, if more than one row is returned, the [EndExecuteXmlReader](#) method attaches the [XmlReader](#) to the value on the first row, and discards the rest of the result set.

The multiple active result set (MARS) feature lets multiple actions use the same connection.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters are sent, this method may block during writes. After the command is sent, the method returns immediately

without waiting for an answer from the server--that is, reads are asynchronous. Although command execution is asynchronous, value fetching is still synchronous.

Because this overload does not support a callback procedure, developers need to either poll to determine whether the command has completed, using the [IsCompleted](#) property of the [IAsyncResult](#) returned by the [BeginExecuteXmlReader](#) method; or wait for the completion of one or more commands using the [AsyncWaitHandle](#) property of the returned [IAsyncResult](#).

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server returns any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **BeginExecuteXmlReader(AsyncCallback, Object)**

## **BeginExecuteXmlReader(AsyncCallback, Object)**

Initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this [SqlCommand](#) and returns results as an [XmlReader](#) object, using a callback procedure.

```
public IAsyncResult BeginExecuteXmlReader (AsyncCallback callback, object stateObject);  
member this.BeginExecuteXmlReader : AsyncCallback * obj -> IAsyncResult
```

Parameters

callback

[AsyncCallback](#)  [AsyncCallback](#)

An  [AsyncCallback](#) delegate that is invoked when the command's execution has completed. Pass `null` (`Nothing` in Microsoft Visual Basic) to indicate that no callback is required.

stateObject

[Object](#)  [Object](#)

A user-defined state object that is passed to the callback procedure. Retrieve this object from within the callback procedure using the  [AsyncState](#) property.

Returns

[IAsyncResult](#) [IAsyncResult](#)

An [IAsyncResult](#) that can be used to poll, wait for results, or both; this value is also needed when the [EndExecuteXmlReader\(IAsyncResult\)](#) is called, which returns the results of the command as XML.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A  [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A  [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A  [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SqlException](#) [SqlException](#)

Any error that occurred while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [InvalidOperationException](#) InvalidOperationException

The name/value pair "Asynchronous Processing=true" was not included within the connection string defining the connection for this [SqlCommand](#).

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [IOException](#) IOException

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [ObjectDisposedException](#) ObjectDisposedException

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### Examples

The following Windows application demonstrates the use of the [BeginExecuteXmlReader](#) method, executing a Transact-SQL statement that includes a delay of a few seconds (emulating a long-running command). This example passes the executing [SqlCommand](#) object as the `stateObject` parameter--doing so makes it simple to retrieve the [SqlCommand](#) object from within the callback procedure, so that the code can call the [EndExecuteXmlReader](#) method corresponding to the initial call to [BeginExecuteXmlReader](#).

This example demonstrates many important techniques. This includes calling a method that interacts with the form from a separate thread. In addition, this example demonstrates how you must block users from executing a command multiple times concurrently, and how you must make sure that the form does not close before the callback procedure is called.

To set up this example, create a new Windows application. Put a [Button](#) control, a [ListBox](#) control, and a [Label](#) control on the form (accepting the default name for each control). Add the following code to the form's class, modifying the connection string as needed for your environment.

```
using System.Data.SqlClient;
using System.Xml;

namespace Microsoft.AdodotNet.CodeSamples
{
    public partial class Form1 : Form
    {
        // Hook up the form's Load event handler and then add
        // this code to the form's class:
        // You need these delegates in order to display text from a thread
        // other than the form's thread. See the HandleCallback
        // procedure for more information.
        private delegate void DisplayInfoDelegate(string Text);
        private delegate void DisplayReaderDelegate(XmlReader reader);

        private bool isExecuting;

        // This example maintains the connection object
        // externally, so that it is available for closing.
        private SqlConnection connection;

        public Form1()
        {
```

```

        InitializeComponent();
    }

private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.

    // If you do not include the Asynchronous Processing=true name/value pair,
    // you will not be able to execute the command asynchronously.
    return "Data Source=(local);Integrated Security=true;" +
    "Initial Catalog=AdventureWorks; Asynchronous Processing=true";
}

private void DisplayStatus(string Text)
{
    this.label1.Text = Text;
}

private void ClearProductInfo()
{
    // Clear the list box.
    this.listBox1.Items.Clear();
}

private void DisplayProductInfo(XmlReader reader)
{
    // Display the data within the reader.
    while (reader.Read())
    {
        // Skip past items that are not from the correct table.
        if (reader.LocalName.ToString() == "Production.Product")
        {
            this.listBox1.Items.Add(String.Format("{0}: {1:C}",
                reader["Name"], Convert.ToDecimal(reader["ListPrice"])));
        }
    }
    DisplayStatus("Ready");
}

private void Form1_FormClosing(object sender,
    System.Windows.Forms.FormClosingEventArgs e)
{
    if (isExecuting)
    {
        MessageBox.Show(this, "Cannot close the form until " +
            "the pending asynchronous command has completed. Please wait...");
        e.Cancel = true;
    }
}

private void button1_Click(object sender, System.EventArgs e)
{
    if (isExecuting)
    {
        MessageBox.Show(this,
            "Already executing. Please wait until the current query " +
            "has completed.");
    }
    else
    {
        SqlCommand command = null;
        try
        {
            ClearProductInfo();
            DisnlavStatus("Connecting..."):

```

```

    DisplayStatus("Connecting... ");
    connection = new SqlConnection(GetConnectionString());

    // To emulate a long-running query, wait for
    // a few seconds before working with the data.
    string commandText =
        "WAITFOR DELAY '00:00:03';" +
        "SELECT Name, ListPrice FROM Production.Product " +
        "WHERE ListPrice < 100 " +
        "FOR XML AUTO, XMLDATA";

    command = new SqlCommand(commandText, connection);
    connection.Open();

    DisplayStatus("Executing...");
    isExecuting = true;
    // Although it is not required that you pass the
    // SqlCommand object as the second parameter in the
    // BeginExecuteXmlReader call, doing so makes it easier
    // to call EndExecuteXmlReader in the callback procedure.
    AsyncCallback callback = new AsyncCallback(HandleCallback);
    command.BeginExecuteXmlReader(callback, command);

}

catch (Exception ex)
{
    isExecuting = false;
    DisplayStatus(string.Format("Ready (last error: {0})", ex.Message));
    if (connection != null)
    {
        connection.Close();
    }
}
}

private void HandleCallback(IAsyncResult result)
{
    try
    {
        // Retrieve the original command object, passed
        // to this procedure in the AsyncState property
        // of the IAsyncResult parameter.
        SqlCommand command = (SqlCommand)result.AsyncState;
        XmlReader reader = command.EndExecuteXmlReader(result);

        // You may not interact with the form and its contents
        // from a different thread, and this callback procedure
        // is all but guaranteed to be running from a different thread
        // than the form.

        // Instead, you must call the procedure from the form's thread.
        // One simple way to accomplish this is to call the Invoke
        // method of the form, which calls the delegate you supply
        // from the form's thread.
        DisplayReaderDelegate del = new DisplayReaderDelegate(DisplayProductInfo);
        this.Invoke(del, reader);

    }
    catch (Exception ex)
    {
        // Because you are now running code in a separate thread,
        // if you do not handle the exception here, none of your other
        // code catches the exception. Because none of
        // your code is on the call stack in this thread, there is nothing
        // higher up the stack to catch the exception if you do not
    }
}

```

```

        // handle it here. You can either log the exception or
        // invoke a delegate (as in the non-error case in this
        // example) to display the error on the form. In no case
        // can you simply display the error without executing a delegate
        // as in the try block here.

        // You can create the delegate instance as you
        // invoke it, like this:
        this.Invoke(new DisplayInfoDelegate(DisplayStatus),
        String.Format("Ready(last error: {0})", ex.Message));
    }
    finally
    {
        isExecuting = false;
        if (connection != null)
        {
            connection.Close();
        }
    }
}

private void Form1_Load(object sender, System.EventArgs e)
{
    this.button1.Click += new System.EventHandler(this.button1_Click);
    this.FormClosing += new System.Windows.Forms.
        FormClosingEventHandler(this.Form1_FormClosing);
}
}
}

```

## Remarks

The [BeginExecuteXmlReader](#) method starts the process of asynchronously executing a Transact-SQL statement or stored procedure that returns rows as XML, so that other tasks can run concurrently while the statement is executing. When the statement has completed, developers must call the [EndExecuteXmlReader](#) method to finish the operation and retrieve the requested XML data. The [BeginExecuteXmlReader](#) method returns immediately, but until the code executes the corresponding [EndExecuteXmlReader](#) method call, it must not execute any other calls that start a synchronous or asynchronous execution against the same [SqlCommand](#) object. Calling the [EndExecuteXmlReader](#) before the command's execution is completed causes the [SqlCommand](#) object to block until the execution is finished.

The [CommandText](#) property ordinarily specifies a Transact-SQL statement with a valid FOR XML clause. However, [CommandText](#) can also specify a statement that returns data that contains valid XML. This method can also be used to retrieve a single-row, single-column result set. In this case, if more than one row is returned, the [EndExecuteXmlReader](#) method attaches the [XmlReader](#) to the value on the first row, and discards the rest of the result set.

A typical [BeginExecuteXmlReader](#) query can be formatted as in the following C# example:

```
SqlCommand command = new SqlCommand("SELECT ContactID, FirstName, LastName FROM Contact FOR XML
AUTO, XMLDATA", SqlConn);
```

This method can also be used to retrieve a single-row, single-column result set. In this case, if more than one row is returned, the [EndExecuteXmlReader](#) method attaches the [XmlReader](#) to the value on the first row, and discards the rest of the result set.

The multiple active result set (MARS) feature lets multiple actions use the same connection.

The [callback](#) parameter lets you specify an [AsyncCallback](#) delegate that is called when the statement has completed. You can call the [EndExecuteXmlReader](#) method from within this delegate procedure, or from any other location within your application. In addition, you can pass any object in the [stateObject](#) parameter, and your callback procedure can retrieve this information using the [AsyncResult](#) property.

Note that the command text and parameters are sent to the server synchronously. If a large command or many parameters is sent, this method may block during writes. After the command is sent, the method returns immediately without waiting for an answer from the server--that is, reads are asynchronous.

All errors that occur during the execution of the operation are thrown as exceptions in the callback procedure. You must handle the exception in the callback procedure, not in the main application. See the example in this topic for additional information on handling exceptions in the callback procedure.

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server will return any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[EndExecuteXmlReader\(IAsyncResult\)](#)

Also

[ExecuteXmlReader\(\)](#)

[ExecuteXmlReader\(IAsyncResult\)](#)

[ADO.NET Overview](#)

# SqlCommand.Cancel SqlCommand.Cancel

## In this Article

Tries to cancel the execution of a [SqlCommand](#).

```
public override void Cancel ();
override this.Cancel : unit -> unit
```

## Examples

The following example demonstrates the use of the [Cancel](#) method.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Threading;

class Program
{
    private static SqlCommand m_rCommand;

    public static SqlCommand Command
    {
        get { return m_rCommand; }
        set { m_rCommand = value; }
    }

    public static void Thread_Cancel()
    {
        Command.Cancel();
    }

    static void Main()
    {
        string connectionString = GetConnectionString();
        try
        {
            using (SqlConnection connection = new SqlConnection(connectionString))
            {
                connection.Open();

                Command = connection.CreateCommand();
                Command.CommandText = "DROP TABLE TestCancel";
                try
                {
                    Command.ExecuteNonQuery();
                }
                catch { }

                Command.CommandText = "CREATE TABLE TestCancel(co1 int, co2 char(10))";
                Command.ExecuteNonQuery();
                Command.CommandText = "INSERT INTO TestCancel VALUES (1, '1')";
                Command.ExecuteNonQuery();

                Command.CommandText = "SELECT * FROM TestCancel";
                SqlDataReader reader = Command.ExecuteReader();

                Thread rThread2 = new Thread(new ThreadStart(Thread_Cancel));
                rThread2.Start();
                rThread2.Join();

                reader.Read();
            }
        }
    }
}
```

```
    reader.Read(),
    System.Console.WriteLine(reader.FieldCount);
    reader.Close();
}
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
}
static private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
    return "Data Source=(local);Initial Catalog=AdventureWorks;" +
        "Integrated Security=SSPI";
}
}
```

## Remarks

If there is nothing to cancel, nothing occurs. However, if there is a command in process, and the attempt to cancel fails, no exception is generated.

In some, rare, cases, if you call [ExecuteReader](#) then call [Close](#) (implicitly or explicitly) before calling [Cancel](#), and then call [Cancel](#), the cancel command will not be sent to SQL Server and the result set can continue to stream after you call [Close](#). To avoid this, make sure that you call [Cancel](#) before closing the reader or connection.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.Clone SqlCommand.Clone

## In this Article

Creates a new [SqlCommand](#) object that is a copy of the current instance.

```
public System.Data.SqlClient.SqlCommand Clone ();  
member this.Clone : unit -> System.Data.SqlClient.SqlCommand
```

Returns

[SqlCommand](#) [SqlCommand](#)

A new [SqlCommand](#) object that is a copy of this instance.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.ColumnEncryptionSetting SqlCommand.ColumnEncryptionSetting

## In this Article

Gets or sets the column encryption setting for this command.

```
[System.ComponentModel.Browsable(false)]
public System.Data.SqlClient.SqlCommandColumnEncryptionSetting ColumnEncryptionSetting { get; }

member this.ColumnEncryptionSetting : System.Data.SqlClient.SqlCommandColumnEncryptionSetting
```

## Returns

[SqlCommandColumnEncryptionSetting](#) [SqlCommandColumnEncryptionSetting](#)

The column encryption setting for this command.

## Attributes

[BrowsableAttribute](#)

# SqlCommand.CommandText SqlCommand.CommandText

## In this Article

Gets or sets the Transact-SQL statement, table name or stored procedure to execute at the data source.

```
[System.Data.DataSysDescription("DbCommand_CommandText")]
public override string CommandText { get; set; }

member this.CommandText : string with get, set
```

Returns

[String String](#)

The Transact-SQL statement or stored procedure to execute. The default is an empty string.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlCommand](#) and sets some of its properties.

```
public void CreateCommand()
{
    SqlCommand command = new SqlCommand();
    command.CommandText = "SELECT * FROM Categories ORDER BY CategoryID";
    command.CommandTimeout = 15;
    command.CommandType = CommandType.Text;
}
```

## Remarks

When the [CommandType](#) property is set to [StoredProcedure](#), the [CommandText](#) property should be set to the name of the stored procedure. The user may be required to use escape character syntax if the stored procedure name contains any special characters. The command executes this stored procedure when you call one of the [Execute](#) methods.

The Microsoft .NET Framework Data Provider for SQL Server does not support the question mark (?) placeholder for passing parameters to a Transact-SQL statement or a stored procedure called by a command of [CommandType.Text](#). In this case, named parameters must be used. For example:

```
SELECT * FROM dbo.Customers WHERE CustomerID = @CustomerID
```

For more information, see [Configuring Parameters and Parameter Data Types](#).

See

[Retrieving and Modifying Data in ADO.NET](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlCommand.CommandTimeout SqlCommand.CommandTimeout

## In this Article

Gets or sets the wait time before terminating the attempt to execute a command and generating an error.

```
[System.Data.DataSysDescription("DbCommand_CommandTimeout")]
public override int CommandTimeout { get; set; }

member this.CommandTimeout : int with get, set
```

Returns

[Int32](#) [Int32](#)

The time in seconds to wait for the command to execute. The default is 30 seconds.

Attributes

[DataSysDescriptionAttribute](#)

## Remarks

A value of 0 indicates no limit (an attempt to execute a command will wait indefinitely).

**Note**

The [CommandTimeout](#) property will be ignored during asynchronous method calls such as [BeginExecuteReader](#).

[CommandTimeout](#) has no effect when the command is executed against a context connection (a [SqlConnection](#) opened with "context connection=true" in the connection string).

**Note**

This property is the cumulative time-out (for all network packets that are read during the invocation of a method) for all network reads during command execution or processing of the results. A time-out can still occur after the first row is returned, and does not include user processing time, only network read time.

For example, with a 30 second time out, if [Read](#) requires two network packets, then it has 30 seconds to read both network packets. If you call [Read](#) again, it will have another 30 seconds to read any data that it requires.

```
using System;
using System.Data.SqlClient;
///
public class A {
    ///
    public static void Main() {
        string connectionString = "";
        // Wait for 5 second delay in the command
        string queryString = "waitfor delay '00:00:05'";
        using (SqlConnection connection = new SqlConnection(connectionString)) {
            connection.Open();
            SqlCommand command = new SqlCommand(queryString, connection);
            // Setting command timeout to 1 second
            command.CommandTimeout = 1;
            try {
                command.ExecuteNonQuery();
            }
            catch (SqlException e) {
                Console.WriteLine("Got expected SqlException due to command timeout ");
                Console.WriteLine(e);
            }
        }
    }
}
```

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.CommandType SqlCommand.CommandType

## In this Article

Gets or sets a value indicating how the [CommandText](#) property is to be interpreted.

```
[System.Data.DataSysDescription("DbCommand_CommandType")]
public override System.Data.CommandType CommandType { get; set; }

member this.CommandType : System.Data.CommandType with get, set
```

Returns

[CommandType](#) [CommandType](#)

One of the [CommandType](#) values. The default is [Text](#).

Attributes

[DataSysDescriptionAttribute](#)

Exceptions

[ArgumentException](#) [ArgumentException](#)

The value was not a valid [CommandType](#).

## Examples

The following example creates a [SqlCommand](#) and sets some of its properties.

```
public void CreateSqlCommand()
{
    SqlCommand command = new SqlCommand();
    command.CommandTimeout = 15;
    command.CommandType = CommandType.Text;
}
```

## Remarks

When you set the [CommandType](#) property to [StoredProcedure](#), you should set the [CommandText](#) property to the name of the stored procedure. The command executes this stored procedure when you call one of the Execute methods.

The Microsoft .NET Framework Data Provider for SQL Server does not support the question mark (?) placeholder for passing parameters to a SQL Statement or a stored procedure called with a [CommandType](#) of [Text](#). In this case, named parameters must be used. For example:

```
SELECT * FROM Customers WHERE CustomerID = @CustomerID
```

For more information, see [Configuring Parameters and Parameter Data Types](#).

See

[Retrieving and Modifying Data in ADO.NET](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlCommand.Connection SqlCommand.Connection

## In this Article

Gets or sets the [SqlConnection](#) used by this instance of the [SqlCommand](#).

```
[System.Data.DataSysDescription("DbCommand_Connection")]
public System.Data.SqlClient.SqlConnection Connection { get; set; }

member this.Connection : System.Data.SqlClient.SqlConnection with get, set
```

Returns

[SqlConnection](#) [SqlConnection](#)

The connection to a data source. The default value is `null`.

Attributes

[DataSysDescriptionAttribute](#)

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The [Connection](#) property was changed while the command was enlisted in a transaction..

## Examples

The following example creates a [SqlCommand](#) and sets some of its properties.

```
private static void CreateCommand(string queryString,
    string connectionString)
{
    using (SqlConnection connection = new SqlConnection(
        connectionString))
    {
        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandTimeout = 15;
        command.CommandType = CommandType.Text;
        command.CommandText = queryString;

        connection.Open();
        SqlDataReader reader = command.ExecuteReader();
        while (reader.Read())
        {
            Console.WriteLine(String.Format("{0}, {1}",
                reader[0], reader[1]));
        }
    }
}
```

## Remarks

If the command is enlisted in an existing transaction, and the connection is changed, trying to execute the command will throw an [InvalidOperationException](#).

If the [Transaction](#) property is not null and the transaction has already been committed or rolled back, [Transaction](#) is set to null.

See

[CommandText](#)  
[CommandText](#)

Also

[CommandTimeout](#)  
[CommandTimeout](#)  
[CommandType](#)  
[CommandType](#)



# SqlCommand.CreateParameter SqlCommand.CreateCommand

## In this Article

Creates a new instance of a [SqlParameter](#) object.

```
public System.Data.SqlClient.SqlParameter CreateParameter ();  
override this.CreateParameter : unit -> System.Data.SqlClient.SqlParameter
```

Returns

[SqlParameter](#) [SqlParameter](#)

A [SqlParameter](#) object.

## Remarks

The [CreateParameter](#) method is a strongly-typed version of [CreateParameter](#).

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.DesignTimeVisible SqlCommand.DesignTimeVisible

## In this Article

Gets or sets a value indicating whether the command object should be visible in a Windows Form Designer control.

```
[System.ComponentModel.Browsable(false)]
public override bool DesignTimeVisible { get; set; }

member this.DesignTimeVisible : bool with get, set
```

## Returns

[Boolean Boolean](#)

A value indicating whether the command object should be visible in a control. The default is **true**.

## Attributes

[BrowsableAttribute](#)

## See

[ADO.NET Overview](#)

## Also

# SqlCommand.EndExecuteNonQuery SqlCommand.EndExecuteNonQuery

## In this Article

Finishes asynchronous execution of a Transact-SQL statement.

```
public int EndExecuteNonQuery (IAsyncResult asyncResult);  
member this.EndExecuteNonQuery : IAsyncResult -> int
```

### Parameters

asyncResult [IAsyncResult](#) [IAsyncResult](#)

The [IAsyncResult](#) returned by the call to [BeginExecuteNonQuery\(\)](#).

### Returns

[Int32](#) [Int32](#)

The number of rows affected (the same behavior as [ExecuteNonQuery\(\)](#)).

### Exceptions

[ArgumentException](#) [ArgumentException](#)

`asyncResult` parameter is null (`Nothing` in Microsoft Visual Basic)

[InvalidOperationException](#) [InvalidOperationException](#)

[EndExecuteNonQuery\(IAsyncResult\)](#) was called more than once for a single command execution, or the method was mismatched against its execution method (for example, the code called [EndExecuteNonQuery\(IAsyncResult\)](#) to complete execution of a call to [BeginExecuteXmlReader\(\)](#)).

[SqlException](#) [SqlException](#)

The amount of time specified in [CommandTimeout](#) elapsed and the asynchronous operation specified with [BeginExecuteNonQuery](#) is not complete.

In some situations, [IAsyncResult](#) can be set to `IsCompleted` incorrectly. If this occurs and [EndExecuteNonQuery\(IAsyncResult\)](#) is called, [EndExecuteNonQuery](#) could raise a [SqlException](#) error if the amount of time specified in [CommandTimeout](#) elapsed and the asynchronous operation specified with [BeginExecuteNonQuery](#) is not complete. To correct this situation, you should either increase the value of [CommandTimeout](#) or reduce the work being done by the asynchronous operation.

## Examples

For examples demonstrating the use of the [EndExecuteNonQuery](#) method, see [BeginExecuteNonQuery](#).

## Remarks

When you call [BeginExecuteNonQuery](#) to execute a Transact-SQL statement, you must call [EndExecuteNonQuery](#) in order to complete the operation. If the process of executing the command has not yet finished, this method blocks until the operation is complete. Users can verify that the command has completed its operation by using the [IAsyncResult](#) instance returned by the [BeginExecuteNonQuery](#) method. If a callback procedure was specified in the call to [BeginExecuteNonQuery](#), this method must be called.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommand.EndExecuteReader SqlCommand.EndExecuteReader

## In this Article

Finishes asynchronous execution of a Transact-SQL statement, returning the requested [SqlDataReader](#).

```
public System.Data.SqlClient.SqlDataReader EndExecuteReader (IAsyncResult asyncResult);  
member this.EndExecuteReader : IAsyncResult -> System.Data.SqlClient.SqlDataReader
```

### Parameters

asyncResult [IAsyncResult](#) [IAsyncResult](#)

The [IAsyncResult](#) returned by the call to [BeginExecuteReader\(\)](#).

### Returns

[SqlDataReader](#) [SqlDataReader](#)

A [SqlDataReader](#) object that can be used to retrieve the requested rows.

### Exceptions

[ArgumentException](#) [ArgumentException](#)

`asyncResult` parameter is null (`Nothing` in Microsoft Visual Basic)

[InvalidOperationException](#) [InvalidOperationException](#)

[EndExecuteReader\(IAsyncResult\)](#) was called more than once for a single command execution, or the method was mismatched against its execution method (for example, the code called [EndExecuteReader\(IAsyncResult\)](#) to complete execution of a call to [BeginExecuteXmlReader\(\)](#)).

## Examples

For examples demonstrating the use of the [EndExecuteReader](#) method, see [BeginExecuteReader](#).

## Remarks

When you call [BeginExecuteReader](#) to execute a Transact-SQL statement, you must call [EndExecuteReader](#) in order to complete the operation. If the process of executing the command has not yet finished, this method blocks until the operation is complete. Users can verify that the command has completed its operation by using the [IAsyncResult](#) instance returned by the [BeginExecuteReader](#) method. If a callback procedure was specified in the call to [BeginExecuteReader](#), this method must be called.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommand.EndExecuteXmlReader SqlCommand.EndExecuteXmlReader

## In this Article

Finishes asynchronous execution of a Transact-SQL statement, returning the requested data as XML.

```
public System.Xml.XmlReader EndExecuteXmlReader (IAsyncResult asyncResult);  
member this.EndExecuteXmlReader : IAsyncResult -> System.Xml.XmlReader
```

### Parameters

asyncResult [IAsyncResult](#)

The [IAsyncResult](#) returned by the call to [BeginExecuteXmlReader\(\)](#).

### Returns

[XmlReader](#) [XmlReader](#)

An [XmlReader](#) object that can be used to fetch the resulting XML data.

### Exceptions

[ArgumentException](#) [ArgumentException](#)

`asyncResult` parameter is null (`Nothing` in Microsoft Visual Basic)

[InvalidOperationException](#) [InvalidOperationException](#)

[EndExecuteXmlReader\(IAsyncResult\)](#) was called more than once for a single command execution, or the method was mismatched against its execution method (for example, the code called [EndExecuteXmlReader\(IAsyncResult\)](#) to complete execution of a call to [BeginExecuteNonQuery\(\)](#)).

## Examples

For examples demonstrating the use of the [EndExecuteXmlReader](#) method, see [BeginExecuteXmlReader](#).

## Remarks

When you call [BeginExecuteXmlReader](#) to execute a Transact-SQL statement, you must call [EndExecuteXmlReader](#) in order to complete the operation. If the process of executing the command has not yet finished, this method blocks until the operation is complete. Users can verify that the command has completed its operation by using the [IAsyncResult](#) instance returned by the [BeginExecuteXmlReader](#) method. If a callback procedure was specified in the call to [BeginExecuteXmlReader](#), this method must be called.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommand.ExecuteNonQuery SqlCommand.ExecuteNonQuery

## In this Article

Executes a Transact-SQL statement against the connection and returns the number of rows affected.

```
public override int ExecuteNonQuery ();  
override this.ExecuteNonQuery : unit -> int
```

## Returns

[Int32](#) [Int32](#)

The number of rows affected.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException](#) [SQLException](#)

An exception occurred while executing the command against a locked row. This exception is not generated when you are using Microsoft .NET Framework version 1.0.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException](#) [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following example creates a [SqlCommand](#) and then executes it using [ExecuteNonQuery](#). The example is passed a string that is a Transact-SQL statement (such as UPDATE, INSERT, or DELETE) and a string to use to connect to the data source.

```
private static void CreateCommand(string queryString,
    string connectionString)
{
    using (SqlConnection connection = new SqlConnection(
        connectionString))
    {
        SqlCommand command = new SqlCommand(queryString, connection);
        command.Connection.Open();
        command.ExecuteNonQuery();
    }
}
```

## Remarks

You can use the [ExecuteNonQuery](#) to perform catalog operations (for example, querying the structure of a database or creating database objects such as tables), or to change the data in a database without using a [DataSet](#) by executing UPDATE, INSERT, or DELETE statements.

Although the [ExecuteNonQuery](#) returns no rows, any output parameters or return values mapped to parameters are populated with data.

For UPDATE, INSERT, and DELETE statements, the return value is the number of rows affected by the command. When a trigger exists on a table being inserted or updated, the return value includes the number of rows affected by both the insert or update operation and the number of rows affected by the trigger or triggers. For all other types of statements, the return value is -1. If a rollback occurs, the return value is also -1.

See

[Retrieving and Modifying Data in ADO.NET](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlCommand.ExecuteNonQueryAsync SqlCommand.ExecuteNonQueryAsync

## In this Article

An asynchronous version of [ExecuteNonQuery\(\)](#), which executes a Transact-SQL statement against the connection and returns the number of rows affected. The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public override System.Threading.Tasks.Task<int> ExecuteNonQueryAsync  
(System.Threading.CancellationToken cancellationToken);  
  
override this.ExecuteNonQueryAsync : System.Threading.CancellationToken ->  
System.Threading.Tasks.Task<int>
```

## Parameters

cancellationToken [CancellationToken CancellationToken](#)

The cancellation instruction.

## Returns

[Task<Int32>](#)

A task representing the asynchronous operation.

## Exceptions

[InvalidOperationException InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[InvalidOperationException InvalidOperationException](#)

Calling [ExecuteNonQueryAsync\(CancellationToken\)](#) more than once for the same instance before task completion.

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

[SQLException SQLException](#)

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about

streaming, see [SqlClient Streaming Support](#).

## Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

# SqlCommand.ExecuteReader SqlCommand.ExecuteReader

In this Article

## Overloads

|  |   |
|--|---|
| <code>ExecuteReader() ExecuteReader()</code>                               | Sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> and builds a <a href="#">SqlDataReader</a> .  |
| <code>ExecuteReader(CommandBehavior) ExecuteReader(CommandBehavior)</code> | Sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> , and builds a <a href="#">SqlDataReader</a> using one of the <a href="#">CommandBehavior</a> values. |

### ExecuteReader() ExecuteReader()

Sends the [CommandText](#) to the [Connection](#) and builds a [SqlDataReader](#).

```
public System.Data.SqlClient.SqlDataReader ExecuteReader ();
override this.ExecuteReader : unit -> System.Data.SqlClient.SqlDataReader
```

Returns

[SqlDataReader](#) [SqlDataReader](#)

A [SqlDataReader](#) object.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SqlException](#) [SqlException](#)

An exception occurred while executing the command against a locked row. This exception is not generated when you are using Microsoft .NET Framework version 1.0.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The current state of the connection is closed. [ExecuteReader\(\)](#) requires an open [SqlConnection](#).

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information

about streaming, see [SqlClient Streaming Support](#).

## ObjectDisposedException ObjectDisposedException

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following example creates a [SqlCommand](#), and then executes it by passing a string that is a Transact-SQL SELECT statement, and a string to use to connect to the data source.

```
private static void CreateCommand(string queryString,
    string connectionString)
{
    using (SqlConnection connection = new SqlConnection(
        connectionString))
    {
        connection.Open();

        SqlCommand command = new SqlCommand(queryString, connection);
        SqlDataReader reader = command.ExecuteReader();
        while (reader.Read())
        {
            Console.WriteLine(String.Format("{0}", reader[0]));
        }
    }
}
```

## Remarks

When the  [CommandType](#) property is set to [StoredProcedure](#), the [CommandText](#) property should be set to the name of the stored procedure. The command executes this stored procedure when you call [ExecuteReader](#).

### Note

If a transaction is deadlocked, an exception may not be thrown until [Read](#) is called.

The multiple active result set (MARS) feature allows for multiple actions using the same connection.

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server will return any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

### See

### Also

[Retrieving and Modifying Data in ADO.NET](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

## ExecuteReader(CommandBehavior) ExecuteReader(CommandBehavior)

Sends the [CommandText](#) to the [Connection](#), and builds a [SqlDataReader](#) using one of the [CommandBehavior](#) values.

```
public System.Data.SqlClient.SqlDataReader ExecuteReader (System.Data.CommandBehavior behavior);
override this.ExecuteReader : System.Data.CommandBehavior -> System.Data.SqlClient.SqlDataReader
```

### Parameters

behavior

[CommandBehavior](#) [CommandBehavior](#)

One of the [CommandBehavior](#) values.

Returns

[SqlDataReader](#) [SqlDataReader](#)

A [SqlDataReader](#) object.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException](#) [SQLException](#)

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException](#) [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following example creates a [SqlCommand](#), and then executes it by passing a string that is a Transact-SQL SELECT statement, and a string to use to connect to the data source. [CommandBehavior](#) is set to [CloseConnection](#).

```
private static void CreateCommand(string queryString,
    string connectionString)
{
    using (SqlConnection connection = new SqlConnection(
        connectionString))
    {
        SqlCommand command = new SqlCommand(queryString, connection);
        connection.Open();
        SqlDataReader reader =
            command.ExecuteReader(CommandBehavior.CloseConnection);
        while (reader.Read())
        {
            Console.WriteLine(String.Format("{0}", reader[0]));
        }
    }
}
```

## Remarks

When the  [CommandType](#) property is set to [StoredProcedure](#), the [CommandText](#) property should be set to the name

of the stored procedure. The command executes this stored procedure when you call [ExecuteReader](#).

**Note**

Use [SequentialAccess](#) to retrieve large values and binary data. Otherwise, an [OutOfMemoryException](#) might occur and the connection will be closed.

The multiple active result set (MARS) feature allows for multiple actions using the same connection.

If you use [ExecuteReader](#) or [BeginExecuteReader](#) to access XML data, SQL Server will return any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use [ExecuteXmlReader](#) or [BeginExecuteXmlReader](#) to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.ExecuteReaderAsync SqlCommand.ExecuteReaderAsync

In this Article

## Overloads

|  |  |
|--|--|
| <a href="#">ExecuteReaderAsync()</a> <a href="#">ExecuteReaderAsync()</a>  | An asynchronous version of <a href="#">ExecuteReader()</a> , which sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> and builds a <a href="#">SqlDataReader</a> . Exceptions will be reported via the returned Task object.  |
| <a href="#">ExecuteReaderAsync(CommandBehavior)</a> <a href="#">ExecuteReaderAsync(CommandBehavior)</a>  | An asynchronous version of <a href="#">ExecuteReader(CommandBehavior)</a> , which sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> , and builds a <a href="#">SqlDataReader</a> . Exceptions will be reported via the returned Task object.   |
| <a href="#">ExecuteReaderAsync(CancellationToken)</a> <a href="#">ExecuteReaderAsync(CancellationToken)</a>                                      | An asynchronous version of <a href="#">ExecuteReader()</a> , which sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> and builds a <a href="#">SqlDataReader</a> . The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.                  |
| <a href="#">ExecuteReaderAsync(CommandBehavior, CancellationToken)</a><br><a href="#">ExecuteReaderAsync(CommandBehavior, CancellationToken)</a> | An asynchronous version of <a href="#">ExecuteReader(CommandBehavior)</a> , which sends the <a href="#">CommandText</a> to the <a href="#">Connection</a> , and builds a <a href="#">SqlDataReader</a> . The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object. |

## Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

## ExecuteReaderAsync() ExecuteReaderAsync()

An asynchronous version of [ExecuteReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds a [SqlDataReader](#). Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader> ExecuteReaderAsync ();  
override this.ExecuteReaderAsync : unit ->  
System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader>
```

Returns

[Task<SqlDataReader>](#)

A task representing the asynchronous operation.

Exceptions

#### [InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

#### [ArgumentException](#) [ArgumentException](#)

An invalid [CommandBehavior](#) value.

#### [InvalidOperationException](#) [InvalidOperationException](#)

Calling [ExecuteReaderAsync\(\)](#) more than once for the same instance before task completion.

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

#### [SqlException](#) [SqlException](#)

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [ObjectDisposedException](#) [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

## **ExecuteReaderAsync(CommandBehavior)**

## **ExecuteReaderAsync(CommandBehavior)**

An asynchronous version of [ExecuteReader\(CommandBehavior\)](#), which sends the [CommandText](#) to the [Connection](#), and builds a [SqlDataReader](#). Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader> ExecuteReaderAsync  
(System.Data.CommandBehavior behavior);  
  
override this.ExecuteReaderAsync : System.Data.CommandBehavior ->  
System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader>
```

## Parameters

### behavior

CommandBehavior CommandBehavior

Options for statement execution and data retrieval. When is set to `Default`, `ReadAsync(CancellationToken)` reads the entire row before returning a complete Task.

### Returns

`Task<SqlDataReader>`

A task representing the asynchronous operation.

### Exceptions

`InvalidOperationException` `InvalidCastException`

A `SqlDbType` other than `Binary` or `VarBinary` was used when `Value` was set to `Stream`. For more information about streaming, see [SqlClient Streaming Support](#).

A `SqlDbType` other than `Char`, `NChar`, `NVarChar`, `VarChar`, or `Xml` was used when `Value` was set to `TextReader`.

A `SqlDbType` other than `Xml` was used when `Value` was set to `XmlReader`.

`ArgumentException` `ArgumentNullException`

An invalid `CommandBehavior` value.

`InvalidOperationException` `InvalidOperationException`

Calling `ExecuteReaderAsync(CommandBehavior)` more than once for the same instance before task completion.

The `SqlConnection` closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

`SqlException` `SqlException`

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`IOException` `IOException`

An error occurred in a `Stream`, `XmlReader` or `TextReader` object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`ObjectDisposedException` `ObjectDisposedException`

The `Stream`, `XmlReader` or `TextReader` object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

### See

[ADO.NET Overview](#)

### Also

## ExecuteReaderAsvnc(CancellationToken)

## ExecuteReaderAsync(CancellationToken)

An asynchronous version of [ExecuteReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds a [SqlDataReader](#).

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader> ExecuteReaderAsync  
(System.Threading.CancellationToken cancellationToken);  
  
override this.ExecuteReaderAsync : System.Threading.CancellationToken ->  
System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader>
```

Parameters

cancellationToken

[CancellationToken](#) [CancellationToken](#)

The cancellation instruction.

Returns

[Task<SqlDataReader>](#)

A task representing the asynchronous operation.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[ArgumentException](#) [ArgumentException](#)

An invalid [CommandBehavior](#) value.

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [ExecuteReaderAsync\(CommandBehavior, CancellationToken\)](#) more than once for the same instance before task completion.

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

[SQLException](#) [SQLException](#)

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## ObjectDisposedException ObjectDisposedException

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

### See

[ADO.NET Overview](#)

### Also

## ExecuteReaderAsync(CommandBehavior, CancellationToken) ExecuteReaderAsync(CommandBehavior, CancellationToken)

An asynchronous version of [ExecuteReader\(CommandBehavior\)](#), which sends the [CommandText](#) to the [Connection](#), and builds a [SqlDataReader](#)

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader> ExecuteReaderAsync  
(System.Data.CommandBehavior behavior, System.Threading.CancellationToken cancellationToken);  
  
override this.ExecuteReaderAsync : System.Data.CommandBehavior * System.Threading.CancellationToken  
-> System.Threading.Tasks.Task<System.Data.SqlClient.SqlDataReader>
```

### Parameters

behavior [CommandBehavior CommandBehavior](#)

Options for statement execution and data retrieval. When is set to [Default](#), [ReadAsync\(CancellationToken\)](#) reads the entire row before returning a complete Task.

cancellationToken [CancellationToken CancellationToken](#)

The cancellation instruction.

### Returns

[Task<SqlDataReader>](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[ArgumentException ArgumentException](#)

An invalid [CommandBehavior](#) value.

[InvalidOperationException InvalidOperationException](#)

Calling `ExecuteReaderAsync(CommandBehavior, CancellationToken)` more than once for the same instance before task completion.

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

#### [SQLException](#) [SQLException](#)

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [ObjectDisposedException](#) [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

# SqlCommand.ExecuteScalar SqlCommand.ExecuteScalar

## In this Article

Executes the query, and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.

```
public override object ExecuteScalar ();
override this.ExecuteScalar : unit -> obj
```

## Returns

[Object](#) [Object](#)

The first column of the first row in the result set, or a null reference (`Nothing` in Visual Basic) if the result set is empty.  
Returns a maximum of 2033 characters.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException](#) [SQLException](#)

An exception occurred while executing the command against a locked row. This exception is not generated when you are using Microsoft .NET Framework version 1.0.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException](#) [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following example creates a [SqlCommand](#) and then executes it using [ExecuteScalar](#). The example is passed a string representing a new value to be inserted into a table, and a string to use to connect to the data source. The function returns the new **Identity** column value if a new row was inserted, 0 on failure.

```

static public int AddProductCategory(string newName, string connString)
{
    Int32 newProdID = 0;
    string sql =
        "INSERT INTO Production.ProductCategory (Name) VALUES (@Name); "
        + "SELECT CAST(scope_identity() AS int)";
    using (SqlConnection conn = new SqlConnection(connString))
    {
        SqlCommand cmd = new SqlCommand(sql, conn);
        cmd.Parameters.Add("@Name", SqlDbType.VarChar);
        cmd.Parameters["@name"].Value = newName;
        try
        {
            conn.Open();
            newProdID = (Int32)cmd.ExecuteScalar();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
    return (int)newProdID;
}

```

## Remarks

Use the [ExecuteScalar](#) method to retrieve a single value (for example, an aggregate value) from a database. This requires less code than using the [ExecuteReader](#) method, and then performing the operations that you need to generate the single value using the data returned by a [SqlDataReader](#).

A typical [ExecuteScalar](#) query can be formatted as in the following C# example:

```

cmd.CommandText = "SELECT COUNT(*) FROM dbo.region";
Int32 count = (Int32) cmd.ExecuteScalar();

```

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommand.ExecuteScalarAsync SqlCommand.ExecuteNonQueryScalarAsync

## In this Article

An asynchronous version of [ExecuteScalar\(\)](#), which executes the query asynchronously and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public override System.Threading.Tasks.Task<object> ExecuteScalarAsync  
(System.Threading.CancellationToken cancellationToken);  
  
override this.ExecuteScalarAsync : System.Threading.CancellationToken ->  
System.Threading.Tasks.Task<obj>
```

## Parameters

cancellationToken

[CancellationToken](#)   [CancellationToken](#)

The cancellation instruction.

## Returns

[Task<Object>](#)

A task representing the asynchronous operation.

## Exceptions

[InvalidCastException](#)   [InvalidCastException](#)

A  [SqlDbType](#)  other than **Binary** or **VarBinary** was used when  [Value](#)  was set to  [Stream](#) . For more information about streaming, see  [SqlConnection Streaming Support](#) .

A  [SqlDbType](#)  other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when  [Value](#)  was set to  [TextReader](#) .

A  [SqlDbType](#)  other than **Xml** was used when  [Value](#)  was set to  [XmlReader](#) .

[InvalidOperationException](#)   [InvalidOperationException](#)

Calling  [ExecuteScalarAsync\(CancellationToken\)](#)  more than once for the same instance before task completion.

The  [SqlConnection](#)  closed or dropped during a streaming operation. For more information about streaming, see  [SqlConnection Streaming Support](#) .

`Context Connection=true` is specified in the connection string.

[SQLException](#)   [SQLException](#)

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see  [SqlConnection Streaming Support](#) .

[IOException](#)   [IOException](#)

An error occurred in a  [Stream](#) ,  [XmlReader](#)  or  [TextReader](#)  object during a streaming operation. For more information about streaming, see  [SqlConnection Streaming Support](#) .

[ObjectDisposedException](#)   [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

# SqlCommand.ExecuteNonQuery SqlCommand.ExecuteNonQuery

## In this Article

Sends the [CommandText](#) to the [Connection](#) and builds an [XmlReader](#) object.

```
public System.Xml.XmlReader ExecuteXmlReader ();  
member this.ExecuteXmlReader : unit -> System.Xml.XmlReader
```

## Returns

[XmlReader](#) [XmlReader](#)

An [XmlReader](#) object.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A [SqlDbType](#) other than **Xml** was used when [Value](#) was set to [XmlReader](#).

[SQLException](#) [SQLException](#)

An exception occurred while executing the command against a locked row. This exception is not generated when you are using Microsoft .NET Framework version 1.0.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlConnection](#) closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[IOException](#) [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

[ObjectDisposedException](#) [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## Examples

The following example creates a [SqlCommand](#) and then executes it using [ExecuteXmlReader](#). The example is passed a string that is a Transact-SQL FOR XML SELECT statement, and a string to use to connect to the data source.

```
private static void CreateXMLReader(string queryString,
    string connectionString)
{
    using (SqlConnection connection = new SqlConnection(
        connectionString))
    {
        connection.Open();
        SqlCommand command = new SqlCommand(queryString, connection);
        System.Xml.XmlReader reader = command.ExecuteXmlReader();
    }
}
```

## Remarks

The **XmlReader** returned by this method does not support asynchronous operations.

The **CommandText** property ordinarily specifies a Transact-SQL statement with a valid FOR XML clause. However, **CommandText** can also specify a statement that returns **ntext** or **nvarchar** data that contains valid XML, or the contents of a column defined with the **xml** data type.

A typical **ExecuteXmlReader** query can be formatted as in the following Microsoft Visual C# example:

```
SqlCommand command = new SqlCommand("SELECT * FROM dbo.Customers FOR XML AUTO, XMLDATA", SqlConn);
```

This method can also be used to retrieve a single-row, single-column result set that contains XML data. In this case, if more than one row is returned, the **ExecuteXmlReader** method attaches the **XmlReader** to the value on the first row, and discards the rest of the result set.

The multiple active result set (MARS) feature allows for multiple actions using the same connection.

If you use **ExecuteReader** or **BeginExecuteReader** to access XML data, SQL Server will return any XML results greater than 2,033 characters in length in multiple rows of 2,033 characters each. To avoid this behavior, use **ExecuteXmlReader** or **BeginExecuteXmlReader** to read FOR XML queries. For more information, see article Q310378, "PRB: XML Data Is Truncated When You Use SqlDataReader," in the Microsoft Knowledge Base at <http://support.microsoft.com>.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.ExecuteReaderAsync SqlCommand.ExecuteReaderAsync

In this Article

## Overloads

|   |  |
|---|--|
| <code>ExecuteXmlReaderAsync()</code> <code>ExecuteXmlReaderAsync()</code>                                   | An asynchronous version of <code>ExecuteXmlReader()</code> , which sends the <code>CommandText</code> to the <code>Connection</code> and builds an <code>XmlReader</code> object.<br><br>Exceptions will be reported via the returned Task object.   |
| <code>ExecuteXmlReaderAsync(CancellationToken)</code> <code>ExecuteXmlReaderAsync(CancellationToken)</code> | An asynchronous version of <code>ExecuteXmlReader()</code> , which sends the <code>CommandText</code> to the <code>Connection</code> and builds an <code>XmlReader</code> object.<br><br>The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object. |

## ExecuteXmlReaderAsync() ExecuteXmlReaderAsync()

An asynchronous version of `ExecuteXmlReader()`, which sends the `CommandText` to the `Connection` and builds an `XmlReader` object.

Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task<System.Xml.XmlReader> ExecuteXmlReaderAsync ();  
member this.ExecuteXmlReaderAsync : unit -> System.Threading.Tasks.Task<System.Xml.XmlReader>
```

Returns

`Task<XmlReader>`

A task representing the asynchronous operation.

Exceptions

`InvalidOperationException` `InvalidOperationException`

A `SqlDbType` other than **Binary** or **VarBinary** was used when `Value` was set to `Stream`. For more information about streaming, see [SqlClient Streaming Support](#).

A `SqlDbType` other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when `Value` was set to `TextReader`.

A `SqlDbType` other than **Xml** was used when `Value` was set to `XmlReader`.

`InvalidOperationException` `InvalidOperationException`

Calling `ExecuteScalarAsync(CancellationToken)` more than once for the same instance before task completion.

The `SqlConnection` closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

## [SqlException](#)

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## [IOException](#)

An error occurred in a [Stream](#), [XmlReader](#) or [TextReader](#) object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

## [ObjectDisposedException](#)

The [Stream](#), [XmlReader](#) or [TextReader](#) object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

### Remarks

The **XmIReader** returned by this method does not support asynchronous operations.

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

### See

[ADO.NET Overview](#)

### Also

## **ExecuteXmlReaderAsync(CancellationToken)** **ExecuteXmlReaderAsync(CancellationToken)**

An asynchronous version of [ExecuteXmlReader\(\)](#), which sends the [CommandText](#) to the [Connection](#) and builds an [XmlReader](#) object.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public System.Threading.Tasks.Task<System.Xml.XmlReader> ExecuteXmlReaderAsync  
(System.Threading.CancellationToken cancellationToken);  
  
member this.ExecuteXmlReaderAsync : System.Threading.CancellationToken ->  
System.Threading.Tasks.Task<System.Xml.XmlReader>
```

### Parameters

#### cancellationToken

[CancellationToken](#) [CancellationToken](#)

The cancellation instruction.

### Returns

#### [Task<XmlReader>](#)

A task representing the asynchronous operation.

### Exceptions

#### [InvalidOperationException](#)

A [SqlDbType](#) other than **Binary** or **VarBinary** was used when [Value](#) was set to [Stream](#). For more information about streaming, see [SqlClient Streaming Support](#).

A [SqlDbType](#) other than **Char**, **NChar**, **NVarChar**, **VarChar**, or **Xml** was used when [Value](#) was set to [TextReader](#).

A `SqlDbType` other than **Xml** was used when `Value` was set to `XmlReader`.

#### [InvalidOperationException](#) `InvalidOperationException`

Calling `ExecuteScalarAsync(CancellationToken)` more than once for the same instance before task completion.

The `SqlConnection` closed or dropped during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

`Context Connection=true` is specified in the connection string.

#### [SqlException](#) `SqlException`

SQL Server returned an error while executing the command text.

A timeout occurred during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [IOException](#) `IOException`

An error occurred in a `Stream`, `XmlReader` or `TextReader` object during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### [ObjectDisposedException](#) `ObjectDisposedException`

The `Stream`, `XmlReader` or `TextReader` object was closed during a streaming operation. For more information about streaming, see [SqlClient Streaming Support](#).

#### Remarks

The **XmlReader** returned by this method does not support asynchronous operations.

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

#### See

[ADO.NET Overview](#)

#### Also

# SqlCommand.ICollectionable.Clone

## In this Article

Creates a new [SqlCommand](#) object that is a copy of the current instance.

```
object ICloneable.Clone();
```

### Returns

[Object](#)

A new [SqlCommand](#) object that is a copy of this instance.

See

[ADO.NET Overview](#)

Also

# SqlCommand.IDbCommand.CreateParameter

## In this Article

```
System.Data.IDbDataParameter IDbCommand.CreateParameter();
```

## Returns

[IDbDataParameter](#)

# SqlCommand.IDbCommand.ExecuteReader

In this Article

## Overloads

[IDbCommand.ExecuteReader\(\)](#)

[IDbCommand.ExecuteReader\(CommandBehavior\)](#)

### IDbCommand.ExecuteReader()

```
System.Data.IDataReader IDbCommand.ExecuteReader ();
```

Returns

[IDataReader](#)

### IDbCommand.ExecuteReader(CommandBehavior)

```
System.Data.IDataReader IDbCommand.ExecuteReader (System.Data.CommandBehavior behavior);
```

Parameters

behavior

[CommandBehavior](#)

Returns

[IDataReader](#)

# SqlCommand.Notification SqlCommand.Notification

## In this Article

Gets or sets a value that specifies the [SqlNotificationRequest](#) object bound to this command.

```
[System.ComponentModel.Browsable(false)]
public System.Data.SqlClient.SqlNotificationRequest Notification { get; set; }

member this.Notification : System.Data.SqlClient.SqlNotificationRequest with get, set
```

Returns

[SqlNotificationRequest](#) [SqlNotificationRequest](#)

When set to null (default), no notification should be requested.

Attributes

[BrowsableAttribute](#)

## Remarks

You must set the value for this property before the command is executed for it to take effect.

See

[Using Query Notifications](#)

Also

[Connecting and Retrieving Data in ADO.NET](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.NotificationAutoEnlist SqlCommand.NotificationAutoEnlist

## In this Article

Gets or sets a value indicating whether the application should automatically receive query notifications from a common [SqlDependency](#) object.

```
public bool NotificationAutoEnlist { get; set; }  
member this.NotificationAutoEnlist : bool with get, set
```

## Returns

[Boolean](#)

**true** if the application should automatically receive query notifications; otherwise **false**. The default value is **true**.

## Remarks

This feature is used in ASP.NET applications to receive notifications for all commands executed in an ASP page against SQL Server. This enables ASP.NET to cache the page until the queries used to render the page would produce a different result. Automatic enlistment.

This property applies only to versions of SQL Server that support query notifications. For earlier versions, setting this property to **true** has no effect on the application.

### See

[Using Query Notifications](#)

### Also

[Connecting and Retrieving Data in ADO.NET](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.Parameters

## In this Article

Gets the [SqlParameterCollection](#).

```
[System.Data.DataSysDescription("DbCommand_Parameters")]
public System.Data.SqlClient.SqlParameterCollection Parameters { get; }

member this.Parameters : System.Data.SqlClient.SqlParameterCollection
```

Returns

[SqlParameterCollection](#) [SqlParameterCollection](#)

The parameters of the Transact-SQL statement or stored procedure. The default is an empty collection.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example demonstrates how to create a [SqlCommand](#) and add parameters to the [SqlParameterCollection](#).

```
private static void UpdateDemographics(Int32 customerID,
    string demoXml, string connectionString)
{
    // Update the demographics for a store, which is stored
    // in an xml column.
    string commandText = "UPDATE Sales.Store SET Demographics = @demographics "
        + "WHERE CustomerID = @ID;";

    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        SqlCommand command = new SqlCommand(commandText, connection);
        command.Parameters.Add("@ID", SqlDbType.Int);
        command.Parameters["@ID"].Value = customerID;

        // Use AddWithValue to assign Demographics.
        // SQL Server will implicitly convert strings into XML.
        command.Parameters.AddWithValue("@demographics", demoXml);

        try
        {
            connection.Open();
            Int32 rowsAffected = command.ExecuteNonQuery();
            Console.WriteLine("RowsAffected: {0}", rowsAffected);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

## Remarks

The Microsoft .NET Framework Data Provider for SQL Server does not support the question mark (?) placeholder for passing parameters to a SQL Statement or a stored procedure called by a command of  [CommandType.Text](#). In this case, named parameters must be used. For example:

```
SELECT * FROM Customers WHERE CustomerID = @CustomerID
```

**Note**

If the parameters in the collection do not match the requirements of the query to be executed, an error may result.

For more information, see [Configuring Parameters and Parameter Data Types](#).

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.Prepare SqlCommand.Prepare

## In this Article

Creates a prepared version of the command on an instance of SQL Server.

```
public override void Prepare ();  
override this.Prepare : unit -> unit
```

## Examples

The following example demonstrates the use of the [Prepare](#) method.

```
private static void SqlCommandPrepareEx(string connectionString)  
{  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        connection.Open();  
        SqlCommand command = new SqlCommand(null, connection);  
  
        // Create and prepare an SQL statement.  
        command.CommandText =  
            "INSERT INTO Region (RegionID, RegionDescription) " +  
            "VALUES (@id, @desc)";  
        SqlParameter idParam = new SqlParameter("@id", SqlDbType.Int, 0);  
        SqlParameter descParam =  
            new SqlParameter("@desc", SqlDbType.Text, 100);  
        idParam.Value = 20;  
        descParam.Value = "First Region";  
        command.Parameters.Add(idParam);  
        command.Parameters.Add(descParam);  
  
        // Call Prepare after setting the Commandtext and Parameters.  
        command.Prepare();  
        command.ExecuteNonQuery();  
  
        // Change parameter values and call ExecuteNonQuery.  
        command.Parameters[0].Value = 21;  
        command.Parameters[1].Value = "Second Region";  
        command.ExecuteNonQuery();  
    }  
}
```

## Remarks

If  [CommandType](#) is set to [StoredProcedure](#), the call to [Prepare](#) should succeed, although it may cause a no-op.

Before you call [Prepare](#), specify the data type of each parameter in the statement to be prepared. For each parameter that has a variable length data type, you must set the [Size](#) property to the maximum size needed. [Prepare](#) returns an error if these conditions are not met.

### Note

If the database context is changed by executing the Transact-SQL [use <database>](#) statement, or by calling the [ChangeDatabase](#) method, then [Prepare](#) must be called a second time.

If you call an [Execute](#) method after calling [Prepare](#), any parameter value that is larger than the value specified by the [Size](#) property is automatically truncated to the original specified size of the parameter, and no truncation errors are returned.

Output parameters (whether prepared or not) must have a user-specified data type. If you specify a variable length data type, you must also specify the maximum [Size](#).

Prior to Visual Studio 2010, [Prepare](#) threw an exception. Beginning in Visual Studio 2010, this method does not throw an exception.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand.ResetCommandTimeout SqlCommand.ResetCommandTimeout

## In this Article

Resets the [CommandTimeout](#) property to its default value.

```
public void ResetCommandTimeout ();  
member this.ResetCommandTimeout : unit -> unit
```

## Remarks

The default value of the [CommandTimeout](#) is 30 seconds.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommand SqlCommand

In this Article

## Overloads

|  |   |
|--|---|
| <code>SqlCommand()</code>  | Initializes a new instance of the <a href="#">SqlCommand</a> class.   |
| <code>SqlCommand(String) SqlCommand(String)</code>   | Initializes a new instance of the <a href="#">SqlCommand</a> class with the text of the query.  |
| <code>SqlCommand(String, SqlConnection) SqlCommand(String, SqlConnection)</code>   | Initializes a new instance of the <a href="#">SqlCommand</a> class with the text of the query and a <a href="#">SqlConnection</a> .                                       |
| <code>SqlCommand(String, SqlConnection, SqlTransaction) SqlCommand(String, SqlConnection, SqlTransaction)</code>   | Initializes a new instance of the <a href="#">SqlCommand</a> class with the text of the query, a <a href="#">SqlConnection</a> , and the <a href="#">SqlTransaction</a> . |
| <code>SqlCommand(String, SqlConnection, SqlTransaction, SqlCommandColumnEncryptionSetting) SqlCommand(String, SqlConnection, SqlTransaction, SqlCommandColumnEncryptionSetting)</code> | Initializes a new instance of the <a href="#">SqlCommand</a> class with specified command text, connection, transaction, and encryption setting.                          |

## SqlCommand()

Initializes a new instance of the [SqlCommand](#) class.

```
public SqlCommand ();
```

### Examples

The following example creates a [SqlCommand](#) and sets the `CommandTimeout` property.

```
public void CreateSqlCommand()
{
    SqlCommand command = new SqlCommand();
    command.CommandTimeout = 15;
    command.CommandType = CommandType.Text;
}
```

### Remarks

The base constructor initializes all fields to their default values. The following table shows initial property values for an instance of [SqlCommand](#).

| PROPERTIES                  | INITIAL VALUE     |
|-----------------------------|-------------------|
| <code>CommandText</code>    | empty string ("") |
| <code>CommandTimeout</code> | 30                |

| PROPERTIES  | INITIAL VALUE    |
|-------------|------------------|
| CommandType | CommandType.Text |
| Connection  | Null             |

You can change the value for any of these properties through a separate call to the property.

See

Also

[Retrieving and Modifying Data in ADO.NET](#)  
[SQL Server and ADO.NET](#)  
[ADO.NET Overview](#)

## SqlCommand(String) SqlCommand(String)

Initializes a new instance of the [SqlCommand](#) class with the text of the query.

```
public SqlCommand (string cmdText);
new System.Data.SqlClient.SqlCommand : string -> System.Data.SqlClient.SqlCommand
```

Parameters

|         |                        |
|---------|------------------------|
| cmdText | <a href="#">String</a> |
|---------|------------------------|

The text of the query.

Examples

The following example creates a [SqlCommand](#), passing in the connection string and command text.

```
public void CreateCommand()
{
    string queryString = "SELECT * FROM Categories ORDER BY CategoryID";
    SqlCommand command = new SqlCommand(queryString);
    command.CommandTimeout = 15;
    command.CommandType = CommandType.Text;
}
```

Remarks

When an instance of [SqlCommand](#) is created, the following read/write properties are set to initial values.

| PROPERTIES     | INITIAL VALUE    |
|----------------|------------------|
| CommandText    | cmdText          |
| CommandTimeout | 30               |
| CommandType    | CommandType.Text |
| Connection     | null             |

You can change the value for any of these properties through a separate call to the property.

See

Also

[Connecting and Retrieving Data in ADO.NET](#)  
[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommand(String, SqlConnection) SqlCommand(String, SqlConnection)

Initializes a new instance of the [SqlCommand](#) class with the text of the query and a [SqlConnection](#).

```
public SqlCommand (string cmdText, System.Data.SqlClient.SqlConnection connection);  
new System.Data.SqlClient.SqlCommand : string * System.Data.SqlClient.SqlConnection ->  
System.Data.SqlClient.SqlCommand
```

## Parameters

cmdText [String](#) String

The text of the query.

connection [SqlConnection](#) SqlConnection

A [SqlConnection](#) that represents the connection to an instance of SQL Server.

## Examples

The following example creates a [SqlCommand](#) and sets some of its properties.

```
private static void CreateCommand(string queryString,  
    string connectionString)  
{  
    using (SqlConnection connection = new SqlConnection(  
        connectionString))  
    {  
        SqlCommand command = new SqlCommand(  
            queryString, connection);  
        connection.Open();  
        SqlDataReader reader = command.ExecuteReader();  
        while (reader.Read())  
        {  
            Console.WriteLine(String.Format("{0}, {1}",  
                reader[0], reader[1]));  
        }  
    }  
}
```

## Remarks

The following table shows initial property values for an instance of [SqlCommand](#).

| PROPERTIES     | INITIAL VALUE   |
|----------------|---|
| CommandText    | cmdText   |
| CommandTimeout | 30  |
| CommandType    | CommandType.Text  |
| Connection     | A new <a href="#">SqlConnection</a> that is the value for the connection parameter. |

You can change the value for any of these parameters by setting the related property.

## See

[Connecting and Retrieving Data in ADO.NET](#)

Also

Using the .NET Framework Data Provider for SQL Server  
ADO.NET Overview

## SqlCommand(String, SqlConnection, SqlTransaction) SqlCommand(String, SqlConnection, SqlTransaction)

Initializes a new instance of the [SqlCommand](#) class with the text of the query, a [SqlConnection](#), and the [SqlTransaction](#).

```
public SqlCommand (string cmdText, System.Data.SqlClient.SqlConnection connection,
System.Data.SqlClient.SqlTransaction transaction);

new System.Data.SqlClient.SqlCommand : string * System.Data.SqlClient.SqlConnection *
System.Data.SqlClient.SqlTransaction -> System.Data.SqlClient.SqlCommand
```

Parameters

cmdText [String](#) [String](#)

The text of the query.

connection [SqlConnection](#) [SqlConnection](#)

A [SqlConnection](#) that represents the connection to an instance of SQL Server.

transaction [SqlTransaction](#) [SqlTransaction](#)

The [SqlTransaction](#) in which the [SqlCommand](#) executes.

Remarks

The following table shows initial property values for an instance of [SqlCommand](#).

| PROPERTIES     | INITIAL VALUE   |
|----------------|---|
| CommandText    | cmdText   |
| CommandTimeout | 30  |
| CommandType    | CommandType.Text  |
| Connection     | A new <a href="#">SqlConnection</a> that is the value for the <a href="#">connection</a> parameter. |

You can change the value for any of these parameters by setting the related property.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## SqlCommand(String, SqlConnection, SqlTransaction, SqlCommandColumnEncryptionSetting) SqlCommand(String, SqlConnection, SqlTransaction, SqlCommandColumnEncryptionSetting)

Initializes a new instance of the [SqlCommand](#) class with specified command text, connection, transaction, and encryption setting.

```
public SqlCommand (string cmdText, System.Data.SqlClient.SqlConnection connection,
System.Data.SqlClient.SqlTransaction transaction,
System.Data.SqlClient.SqlCommandColumnEncryptionSetting columnEncryptionSetting);

new System.Data.SqlClient.SqlCommand : string * System.Data.SqlClient.SqlConnection *
System.Data.SqlClient.SqlTransaction * System.Data.SqlClient.SqlCommandColumnEncryptionSetting ->
System.Data.SqlClient.SqlCommand
```

## Parameters

|  |   |
|--|---|
| cmdText  | <a href="#">String</a> <a href="#">String</a>   |
| The text of the query.   |   |
| connection   | <a href="#">SqlConnection</a> <a href="#">SqlConnection</a>   |
| A <a href="#">SqlConnection</a> that represents the connection to an instance of SQL Server. |   |
| transaction  | <a href="#">SqlTransaction</a> <a href="#">SqlTransaction</a>                                       |
| The <a href="#">SqlTransaction</a> in which the <a href="#">SqlCommand</a> executes.         |   |
| columnEncryptionSetting  | <a href="#">SqlCommandColumnEncryptionSetting</a> <a href="#">SqlCommandColumnEncryptionSetting</a> |
| The encryption setting. For more information, see <a href="#">Always Encrypted</a> .         |   |

# SqlCommand.StatementCompleted SqlCommand.StatementCompleted

## In this Article

Occurs when the execution of a Transact-SQL statement completes.

```
public event System.Data.StatementCompletedEventHandler StatementCompleted;  
member this.StatementCompleted : System.Data.StatementCompletedEventHandler
```

See

[ADO.NET Overview](#)

Also

# SqlCommand.Transaction

## In this Article

Gets or sets the [SqlTransaction](#) within which the [SqlCommand](#) executes.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("DbCommand_Transaction")]
public System.Data.SqlClient.SqlTransaction Transaction { get; set; }

member this.Transaction : System.Data.SqlClient.SqlTransaction with get, set
```

Returns

[SqlTransaction](#) [SqlTransaction](#)

The [SqlTransaction](#). The default value is `null`.

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Remarks

You cannot set the [Transaction](#) property if it is already set to a specific value, and the command is in the process of executing. If you set the transaction property to a [SqlTransaction](#) object that is not connected to the same [SqlConnection](#) as the [SqlCommand](#) object, an exception is thrown the next time that you attempt to execute a statement.

See

[Performing a Transaction](#)

Also

[Connecting and Retrieving Data in ADO.NET](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommand.UpdatedRowSource SqlCommand.UpdatedRowSource

## In this Article

Gets or sets how command results are applied to the [DataRow](#) when used by the **Update** method of the [DbDataAdapter](#).

```
[System.Data.DataSysDescription("DbCommand_UpdatedRowSource")]
public override System.Data.UpdateRowSource UpdatedRowSource { get; set; }

member this.UpdatedRowSource : System.Data.UpdateRowSource with get, set
```

## Returns

[UpdateRowSource](#) [UpdateRowSource](#)

One of the [UpdateRowSource](#) values.

## Attributes

[DataSysDescriptionAttribute](#)

## Remarks

The default [UpdateRowSource](#) value is **Both** unless the command is automatically generated (as in the case of the [SqlCommandBuilder](#)), in which case the default is **None**.

For more information about using the **UpdatedRowSource** property, see [DataAdapter Parameters](#).

### See

[Connecting and Retrieving Data in ADO.NET](#)

### Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder SqlCommandBuilder Class

Automatically generates single-table commands that are used to reconcile changes made to a [DataSet](#) with the associated SQL Server database. This class cannot be inherited.

## Declaration

```
public sealed class SqlCommandBuilder : System.Data.Common.DbCommandBuilder  
type SqlCommandBuilder = class  
    inherit DbCommandBuilder
```

## Inheritance Hierarchy



## Remarks

The [SqlDataAdapter](#) does not automatically generate the Transact-SQL statements required to reconcile changes made to a [DataSet](#) with the associated instance of SQL Server. However, you can create a [SqlCommandBuilder](#) object to automatically generate Transact-SQL statements for single-table updates if you set the [SelectCommand](#) property of the [SqlDataAdapter](#). Then, any additional Transact-SQL statements that you do not set are generated by the [SqlCommandBuilder](#).

The [SqlCommandBuilder](#) registers itself as a listener for [RowUpdating](#) events whenever you set the [DataAdapter](#) property. You can only associate one [SqlDataAdapter](#) or [SqlCommandBuilder](#) object with each other at one time.

To generate INSERT, UPDATE, or DELETE statements, the [SqlCommandBuilder](#) uses the [SelectCommand](#) property to retrieve a required set of metadata automatically. If you change the [SelectCommand](#) after the metadata has been retrieved, such as after the first update, you should call the [RefreshSchema](#) method to update the metadata.

The [SelectCommand](#) must also return at least one primary key or unique column. If none are present, an **InvalidOperationException** exception is generated, and the commands are not generated.

The [SqlCommandBuilder](#) also uses the [Connection](#), [CommandTimeout](#), and [Transaction](#) properties referenced by the [SelectCommand](#). The user should call [RefreshSchema](#) if one or more of these properties are modified, or if the [SelectCommand](#) itself is replaced. Otherwise the [InsertCommand](#), [UpdateCommand](#), and [DeleteCommand](#) properties retain their previous values.

If you call [Dispose](#), the [SqlCommandBuilder](#) is disassociated from the [SqlDataAdapter](#), and the generated commands are no longer used.

## Constructors

```
SqlCommandBuilder()  
SqlCommandBuilder()
```

Initializes a new instance of the [SqlCommandBuilder](#) class.

```
SqlCommandBuilder(SqlDataAdapter)  
SqlCommandBuilder(SqlDataAdapter)
```

Initializes a new instance of the [SqlCommandBuilder](#) class with the associated [SqlDataAdapter](#) object.

## Properties

CatalogLocation

CatalogLocation

Sets or gets the [CatalogLocation](#) for an instance of the [SqlCommandBuilder](#) class.

CatalogSeparator

CatalogSeparator

Sets or gets a string used as the catalog separator for an instance of the [SqlCommandBuilder](#) class.

DataAdapter

DataAdapter

Gets or sets a [SqlDataAdapter](#) object for which Transact-SQL statements are automatically generated.

QuotePrefix

QuotePrefix

Gets or sets the starting character or characters to use when specifying SQL Server database objects, such as tables or columns, whose names contain characters such as spaces or reserved tokens.

QuoteSuffix

QuoteSuffix

Gets or sets the ending character or characters to use when specifying SQL Server database objects, such as tables or columns, whose names contain characters such as spaces or reserved tokens.

SchemaSeparator

SchemaSeparator

Gets or sets the character to be used for the separator between the schema identifier and any other identifiers.

## Methods

DeriveParameters(SqlCommand)

DeriveParameters(SqlCommand)

Retrieves parameter information from the stored procedure specified in the [SqlCommand](#) and populates the [Parameters](#) collection of the specified [SqlCommand](#) object.

GetDeleteCommand()

GetDeleteCommand()

Gets the automatically generated [SqlCommand](#) object required to perform deletions on the database.

```
GetDeleteCommand(Boolean)  
GetDeleteCommand(Boolean)
```

Gets the automatically generated [SqlCommand](#) object that is required to perform deletions on the database.

```
GetInsertCommand()  
GetInsertCommand()
```

Gets the automatically generated [SqlCommand](#) object required to perform insertions on the database.

```
GetInsertCommand(Boolean)  
GetInsertCommand(Boolean)
```

Gets the automatically generated [SqlCommand](#) object that is required to perform insertions on the database.

```
GetUpdateCommand()  
GetUpdateCommand()
```

Gets the automatically generated [SqlCommand](#) object required to perform updates on the database.

```
GetUpdateCommand(Boolean)  
GetUpdateCommand(Boolean)
```

Gets the automatically generated [SqlCommand](#) object required to perform updates on the database.

```
QuoteIdentifier(String)  
QuoteIdentifier(String)
```

Given an unquoted identifier in the correct catalog case, returns the correct quoted form of that identifier. This includes correctly escaping any embedded quotes in the identifier.

```
RefreshSchema()  
RefreshSchema()
```

```
UnquoteIdentifier(String)  
UnquoteIdentifier(String)
```

Given a quoted identifier, returns the correct unquoted form of that identifier. This includes correctly unescaping any embedded quotes in the identifier.

## See Also

# SqlCommandBuilder.CatalogLocation SqlCommand Builder.CatalogLocation

## In this Article

Sets or gets the [CatalogLocation](#) for an instance of the [SqlCommandBuilder](#) class.

```
[System.ComponentModel.Browsable(false)]
public override System.Data.Common.CatalogLocation CatalogLocation { get; set; }

member this.CatalogLocation : System.Data.Common.CatalogLocation with get, set
```

## Returns

[CatalogLocation CatalogLocation](#)

A [CatalogLocation](#) object.

## Attributes

[BrowsableAttribute](#)

## See

[Connecting and Retrieving Data in ADO.NET](#)

## Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.CatalogSeparator SqlCommand Builder.CatalogSeparator

## In this Article

Sets or gets a string used as the catalog separator for an instance of the [SqlCommandBuilder](#) class.

```
[System.ComponentModel.Browsable(false)]  
public override string CatalogSeparator { get; set; }  
  
member this.CatalogSeparator : string with get, set
```

## Returns

[String](#) [String](#)

A string that indicates the catalog separator for use with an instance of the [SqlCommandBuilder](#) class.

## Attributes

[BrowsableAttribute](#)

## See

[Connecting and Retrieving Data in ADO.NET](#)

## Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.DataAdapter SqlCommandBuilder. DataAdapter

## In this Article

Gets or sets a [SqlDataAdapter](#) object for which Transact-SQL statements are automatically generated.

```
[System.Data.DataSysDescription("SqlCommandBuilder_DataAdapter")]
public System.Data.SqlClient.SqlDataAdapter DataAdapter { get; set; }

member this.DataAdapter : System.Data.SqlClient.SqlDataAdapter with get, set
```

## Returns

[SqlDataAdapter](#) [SqlDataAdapter](#)

A [SqlDataAdapter](#) object.

## Attributes

[DataSysDescriptionAttribute](#)

## Remarks

The [SqlCommandBuilder](#) registers itself as a listener for [RowUpdating](#) events that are generated by the [SqlDataAdapter](#) specified in this property.

When you create a new instance of [SqlCommandBuilder](#), any existing [SqlCommandBuilder](#) associated with this [SqlDataAdapter](#) is released.

## See

[Connecting and Retrieving Data in ADO.NET](#)

## Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlCommandBuilder.DeriveParameters SqlCommand Builder.DeriveParameters

## In this Article

Retrieves parameter information from the stored procedure specified in the [SqlCommand](#) and populates the [Parameters](#) collection of the specified [SqlCommand](#) object.

```
public static void DeriveParameters (System.Data.SqlClient.SqlCommand command);  
static member DeriveParameters : System.Data.SqlClient.SqlCommand -> unit
```

### Parameters

command [SqlCommand](#) [SqlCommand](#)

The [SqlCommand](#) referencing the stored procedure from which the parameter information is to be derived. The derived parameters are added to the [Parameters](#) collection of the [SqlCommand](#).

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The command text is not a valid stored procedure name.

## Remarks

[DeriveParameters](#) overwrites any existing parameter information for the [SqlDbCommand](#).

[DeriveParameters](#) requires an additional call to the database to obtain the information. If the parameter information is known in advance, it is more efficient to populate the parameters collection by setting the information explicitly.

You can only use [DeriveParameters](#) with stored procedures. You cannot use [DeriveParameters](#) with extended stored procedures. You cannot use [DeriveParameters](#) to populate the [SqlParameterCollection](#) with arbitrary Transact-SQL statements, such as a parameterized SELECT statement.

For more information, see [Configuring Parameters and Parameter Data Types](#).

### See

[Connecting and Retrieving Data in ADO.NET](#)

### Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.GetDeleteCommand SqlCommand Builder.GetDeleteCommand

In this Article

## Overloads

|  |   |
|--|---|
| <code>GetDeleteCommand() GetDeleteCommand()</code>               | Gets the automatically generated <a href="#">SqlCommand</a> object required to perform deletions on the database.         |
| <code>GetDeleteCommand(Boolean) GetDeleteCommand(Boolean)</code> | Gets the automatically generated <a href="#">SqlCommand</a> object that is required to perform deletions on the database. |

## Remarks

An application can use the [GetDeleteCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetDeleteCommand](#) as the basis of a modified command. For example, you might call [GetDeleteCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetDeleteCommand](#) will still be using information from the previous statement, which might not be correct. The SQL statements are first generated when the application calls either [Update](#) or [GetDeleteCommand](#).

For more information, see [Generating Commands with CommandBuilders](#).

## GetDeleteCommand() GetDeleteCommand()

Gets the automatically generated [SqlCommand](#) object required to perform deletions on the database.

```
public System.Data.SqlClient.SqlCommand GetDeleteCommand ();  
override this.GetDeleteCommand : unit -> System.Data.SqlClient.SqlCommand
```

Returns

[SqlCommand](#) [SqlCommand](#)

The automatically generated [SqlCommand](#) object required to perform deletions.

### Remarks

An application can use the [GetDeleteCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetDeleteCommand](#) as the basis of a modified command. For example, you might call [GetDeleteCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetDeleteCommand](#) will still be using information from the previous statement, which might not be correct. The SQL statements are first generated when the application calls either [Update](#) or [GetDeleteCommand](#).

For more information, see [Generating Commands with CommandBuilders](#).

See

## Connecting and Retrieving Data in ADO.NET

Also

# Using the .NET Framework Data Provider for SQL Server

## ADO.NET Overview

**GetDeleteCommand(Boolean) GetDeleteCommand(Boolean)**

Gets the automatically generated `SqlCommand` object that is required to perform deletions on the database.

```
public System.Data.SqlClient.SqlCommand GetDeleteCommand (bool useColumnsForParameterNames);  
override this.GetDeleteCommand : bool -> System.Data.SqlClient.SqlCommand
```

## Parameters

`useColumnsForParameterNames` Boolean Boolean

If true, generate parameter names matching column names if possible. If false, generate @p1, @p2, and so on.

## Returns

## SqlCommand SqlCommand

## Boolean Boolean

The automatically generated `SqlCommand` object that is required to perform deletions.

## Remarks

An application can use the [GetDeleteCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetDeleteCommand](#) as the basis of a modified command. For example, you might call [GetDeleteCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetDeleteCommand](#) will still be using information from the previous statement, which might not be correct. The SQL statements are first generated when the application calls either [Update](#) or [GetDeleteCommand](#).

The default behavior, when generating parameter names, is to use @p1, @p2, and so on for the various parameters. Passing `true` for the `useColumnsForParameterNames` parameter lets you force the `OleDbCommandBuilder` to generate parameters based on the column names instead. This succeeds only if the following conditions are met:

- The [ParameterNameMaxLength](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection has been specified and its length is equal to or greater than the generated parameter name.
  - The generated parameter name meets the criteria specified in the [ParameterNamePattern](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection regular expression.
  - A [ParameterMarkerFormat](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection is specified.

For more information, see [Generating Commands with CommandBuilders](#).

See

## Connecting and Retrieving Data in ADO.NET

Also

# Using the .NET Framework Data Provider for SQL Server

## ADO.NET Overview

# SqlCommandBuilder.GetInsertCommand SqlCommand Builder.GetInsertCommand

In this Article

## Overloads

|  |  |
|--|--|
| <code>GetInsertCommand()</code>        | Gets the automatically generated <a href="#">SqlCommand</a> object required to perform insertions on the database.         |
| <code>GetInsertCommand(Boolean)</code> | Gets the automatically generated <a href="#">SqlCommand</a> object that is required to perform insertions on the database. |

## Remarks

An application can use the [GetInsertCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetInsertCommand](#) as the basis of a modified command. For example, you might call [GetInsertCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the Transact-SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetInsertCommand](#) will still be using information from the previous statement, which might not be correct. The Transact-SQL statements are first generated when the application calls either [Update](#) or [GetInsertCommand](#).

For more information, see [Generating Commands with CommandBuilders](#).

## GetInsertCommand() GetInsertCommand()

Gets the automatically generated [SqlCommand](#) object required to perform insertions on the database.

```
public System.Data.SqlClient.SqlCommand GetInsertCommand ();  
override this.GetInsertCommand : unit -> System.Data.SqlClient.SqlCommand
```

Returns

[SqlCommand](#) [SqlCommand](#)

The automatically generated [SqlCommand](#) object required to perform insertions.

### Remarks

An application can use the [GetInsertCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetInsertCommand](#) as the basis of a modified command. For example, you might call [GetInsertCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the Transact-SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetInsertCommand](#) will still be using information from the previous statement, which might not be correct. The Transact-SQL statements are first generated when the application calls either [Update](#) or [GetInsertCommand](#).

For more information, see [Generating Commands with CommandBuilders](#).

See

## Connecting and Retrieving Data in ADO.NET

Also

# Using the .NET Framework Data Provider for SQL Server

## ADO.NET Overview

## **GetInsertCommand(Boolean) GetInsertCommand(Boolean)**

Gets the automatically generated `SqlCommand` object that is required to perform insertions on the database.

```
public System.Data.SqlClient.SqlCommand GetInsertCommand (bool useColumnsForParameterNames);  
override this.GetInsertCommand : bool -> System.Data.SqlClient.SqlCommand
```

## Parameters

`useColumnsForParameterNames` Boolean Boolean

If true, generate parameter names matching column names if possible. If false, generate @p1, @p2, and so on.

## Returns

## SqlCommand SqlCommand

## Boolean Boolean

The automatically generated `SqlCommand` object that is required to perform insertions.

## Remarks

An application can use the [GetInsertCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use `GetInsertCommand` as the basis of a modified command. For example, you might call `GetInsertCommand` and modify the `CommandTimeout` value, and then explicitly set that on the `SqlDataAdapter`.

After the Transact-SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetInsertCommand](#) will still be using information from the previous statement, which might not be correct. The Transact-SQL statements are first generated when the application calls either [Update](#) or [GetInsertCommand](#).

The default behavior, when generating parameter names, is to use @p1, @p2, and so on for the various parameters.

Passing `true` for the `useColumnsForParameterNames` parameter lets you force the `OleDbCommandBuilder` to generate parameters based on the column names instead. This succeeds only if the following conditions are met:

- The [ParameterNameMaxLength](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection has been specified and its length is equal to or greater than the generated parameter name.
  - The generated parameter name meets the criteria specified in the [ParameterNamePattern](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection regular expression.
  - A [ParameterMarkerFormat](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection is specified.

For more information, see [Generating Commands with CommandBuilders](#).

See

## Connecting and Retrieving Data in ADO.NET

Also

# Using the .NET Framework Data Provider for SQL Server

## ADO.NET Overview

# SqlCommandBuilder.GetUpdateCommand SqlCommand Builder.GetUpdateCommand

In this Article

## Overloads

|  |   |
|--|---|
| <a href="#">GetUpdateCommand()</a> <b>GetUpdateCommand()</b>               | Gets the automatically generated <a href="#">SqlCommand</a> object required to perform updates on the database. |
| <a href="#">GetUpdateCommand(Boolean)</a> <b>GetUpdateCommand(Boolean)</b> | Gets the automatically generated <a href="#">SqlCommand</a> object required to perform updates on the database. |

## Remarks

An application can use the [GetUpdateCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetUpdateCommand](#) as the basis of a modified command. For example, you might call [GetUpdateCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the Transact-SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetUpdateCommand](#) will still be using information from the previous statement, which might not be correct. The Transact-SQL statements are first generated when the application calls either [Update](#) or [GetUpdateCommand](#).

For more information, see [Generating Commands with CommandBuilders](#).

## GetUpdateCommand() GetUpdateCommand()

Gets the automatically generated [SqlCommand](#) object required to perform updates on the database.

```
public System.Data.SqlClient.SqlCommand GetUpdateCommand ();  
override this.GetUpdateCommand : unit -> System.Data.SqlClient.SqlCommand
```

Returns

[SqlCommand](#) [SqlCommand](#)

The automatically generated [SqlCommand](#) object that is required to perform updates.

### Remarks

An application can use the [GetUpdateCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetUpdateCommand](#) as the basis of a modified command. For example, you might call [GetUpdateCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the Transact-SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetUpdateCommand](#) will still be using information from the previous statement, which might not be correct. The Transact-SQL statements are first generated when the application calls either [Update](#) or [GetUpdateCommand](#).

For more information, see [Generating Commands with CommandBuilders](#).

See

## Connecting and Retrieving Data in ADO.NET

Also

# Using the .NET Framework Data Provider for SQL Server

## ADO.NET Overview

## **GetUpdateCommand(Boolean) GetUpdateCommand(Boolean)**

Gets the automatically generated `SqlCommand` object required to perform updates on the database.

```
public System.Data.SqlClient.SqlCommand GetUpdateCommand (bool useColumnsForParameterNames);  
override this.GetUpdateCommand : bool -> System.Data.SqlClient.SqlCommand
```

## Parameters

`useColumnsForParameterNames` Boolean Boolean

If true, generate parameter names matching column names if possible. If false, generate @p1, @p2, and so on.

## Returns

## SqlCommand SqlCommand

## Boolean Boolean

The automatically generated `SqlCommand` object required to perform updates.

## Remarks

An application can use the [GetUpdateCommand](#) method for informational or troubleshooting purposes because it returns the [SqlCommand](#) object to be executed.

You can also use [GetUpdateCommand](#) as the basis of a modified command. For example, you might call [GetUpdateCommand](#) and modify the [CommandTimeout](#) value, and then explicitly set that on the [SqlDataAdapter](#).

After the Transact-SQL statement is first generated, the application must explicitly call [RefreshSchema](#) if it changes the statement in any way. Otherwise, the [GetUpdateCommand](#) will still be using information from the previous statement, which might not be correct. The Transact-SQL statements are first generated when the application calls either [Update](#) or [GetUpdateCommand](#).

The default behavior, when generating parameter names, is to use @p1, @p2, and so on for the various parameters.

Passing `true` for the `useColumnsForParameterNames` parameter lets you force the `OleDbCommandBuilder` to generate parameters based on the column names instead. This succeeds only if the following conditions are met:

- The [ParameterNameMaxLength](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection has been specified and its length is equal to or greater than the generated parameter name.
  - The generated parameter name meets the criteria specified in the [ParameterNamePattern](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection regular expression.
  - A [ParameterMarkerFormat](#) returned from the **GetSchema** method call and found in the [DataSourceInformation](#) collection is specified.

For more information, see [Generating Commands with CommandBuilders](#).

See

## Connecting and Retrieving Data in ADO.NET

Also

# Using the .NET Framework Data Provider for SQL Server

## ADO.NET Overview

# SqlCommandBuilder.QuoteIdentifier SqlCommand Builder.QuoteIdentifier

## In this Article

Given an unquoted identifier in the correct catalog case, returns the correct quoted form of that identifier. This includes correctly escaping any embedded quotes in the identifier.

```
public override string QuoteIdentifier (string unquotedIdentifier);  
override this.QuoteIdentifier : string -> string
```

### Parameters

unquotedIdentifier [String](#) [String](#)

The original unquoted identifier.

### Returns

[String](#) [String](#)

The quoted version of the identifier. Embedded quotes within the identifier are correctly escaped.

### See

[Connecting and Retrieving Data in ADO.NET](#)

### Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.QuotePrefix SqlCommandBuilder.QuotePrefix

## In this Article

Gets or sets the starting character or characters to use when specifying SQL Server database objects, such as tables or columns, whose names contain characters such as spaces or reserved tokens.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("SqlCommandBuilder_QuotePrefix")]
public override string QuotePrefix { get; set; }

member this.QuotePrefix : string with get, set
```

## Returns

[String](#)

The starting character or characters to use. The default is an empty string.

## Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

This property cannot be changed after an INSERT, UPDATE, or DELETE command has been generated.

## Remarks

### [Note](#)

Although you cannot change the [QuotePrefix](#) or [QuoteSuffix](#) properties after an insert, update, or delete command has been generated, you can change their settings after calling the [Update](#) method of a DataAdapter.

## See

[Connecting and Retrieving Data in ADO.NET](#)

## Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.QuoteSuffix SqlCommandBuilder.QuoteSuffix

## In this Article

Gets or sets the ending character or characters to use when specifying SQL Server database objects, such as tables or columns, whose names contain characters such as spaces or reserved tokens.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("SqlCommandBuilder_QuoteSuffix")]
public override string QuoteSuffix { get; set; }

member this.QuoteSuffix : string with get, set
```

### Returns

[String String](#)

The ending character or characters to use. The default is an empty string.

### Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

This property cannot be changed after an insert, update, or delete command has been generated.

## Remarks

### Note

Although you cannot change the [QuotePrefix](#) or [QuoteSuffix](#) properties after an insert, update, or delete operation has been generated, you can change their settings after calling the [Update](#) method of a DataAdapter.

### See

[Connecting and Retrieving Data in ADO.NET](#)

### Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.RefreshSchema SqlCommand Builder.RefreshSchema

## In this Article

```
public void RefreshSchema ();  
member this.RefreshSchema : unit -> unit
```

# SqlCommandBuilder.SchemaSeparator SqlCommandBuilder.SchemaSeparator

## In this Article

Gets or sets the character to be used for the separator between the schema identifier and any other identifiers.

```
[System.ComponentModel.Browsable(false)]
public override string SchemaSeparator { get; set; }

member this.SchemaSeparator : string with get, set
```

Returns

[String](#)

The character to be used as the schema separator.

Attributes

[BrowsableAttribute](#)

## Remarks

Generally, database servers indicate the schema for a identifier by separating the schema name from the identifier with some character. For example, SQL Server uses a period, creating complete identifiers such as Person.CustomerName, where "Person" is the schema name and "CustomerName" is the identifier. Setting this property lets developers modify this behavior.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder SqlCommandBuilder

In this Article

## Overloads

|  |  |
|--|--|
| <code>SqlCommandBuilder()</code>   | Initializes a new instance of the <a href="#">SqlCommandBuilder</a> class.   |
| <code>SqlCommandBuilder(SqlDataAdapter) SqlCommandBuilder(SqlDataAdapter)</code> | Initializes a new instance of the <a href="#">SqlCommandBuilder</a> class with the associated <a href="#">SqlDataAdapter</a> object. |

## SqlCommandBuilder()

Initializes a new instance of the [SqlCommandBuilder](#) class.

```
public SqlCommandBuilder ();
```

See

[ADO.NET Overview](#)

Also

## SqlCommandBuilder(SqlDataAdapter) SqlCommandBuilder(SqlDataAdapter)

Initializes a new instance of the [SqlCommandBuilder](#) class with the associated [SqlDataAdapter](#) object.

```
public SqlCommandBuilder (System.Data.SqlClient.SqlDataAdapter adapter);  
  
new System.Data.SqlClient.SqlCommandBuilder : System.Data.SqlClient.SqlDataAdapter ->  
System.Data.SqlClient.SqlCommandBuilder
```

Parameters

adapter

[SqlDataAdapter](#) [SqlDataAdapter](#)

The name of the [SqlDataAdapter](#).

Remarks

The [SqlCommandBuilder](#) registers itself as a listener for [RowUpdating](#) events that are generated by the [SqlDataAdapter](#) specified in this property.

When you create a new instance of [SqlCommandBuilder](#), any existing [SqlCommandBuilder](#) associated with this [SqlDataAdapter](#) is released.

See

[Connecting and Retrieving Data in ADO.NET](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandBuilder.UnquoteIdentifier SqlCommandBuilder.UnquoteIdentifier

## In this Article

Given a quoted identifier, returns the correct unquoted form of that identifier. This includes correctly unescaping any embedded quotes in the identifier.

```
public override string UnquoteIdentifier (string quotedIdentifier);  
override this.UnquoteIdentifier : string -> string
```

## Parameters

**quotedIdentifier** [String](#) [String](#)

The identifier that will have its embedded quotes removed.

## Returns

[String](#) [String](#)

The unquoted identifier, with embedded quotes properly unescaped.

## See

[Connecting and Retrieving Data in ADO.NET](#)

## Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlCommandColumnEncryptionSetting SqlCommand ColumnEncryptionSetting Enum

Specifies how data will be sent and received when reading and writing encrypted columns. Depending on your specific query, performance impact may be reduced by bypassing the Always Encrypted driver's processing when non-encrypted columns are being used. Note that these settings cannot be used to bypass encryption and gain access to plaintext data. For details, see [Always Encrypted \(Database Engine\)](#)

## Declaration

```
public enum SqlCommandColumnEncryptionSetting  
type SqlCommandColumnEncryptionSetting =
```

## Inheritance Hierarchy



## Remarks

- If Always Encrypted is disabled for a query and the query has parameters that need to be encrypted (parameters that correspond to encrypted columns), the query will fail.
- If Always Encrypted is disabled for a query and the query returns results from encrypted columns, the query will return encrypted values. The encrypted values will have the varbinary datatype.

## Fields

|          |          |
|----------|----------|
| Disabled | Disabled |
|----------|----------|

Disables Always Encrypted for the query.

|         |         |
|---------|---------|
| Enabled | Enabled |
|---------|---------|

Enables Always Encrypted for the query.

|               |               |
|---------------|---------------|
| ResultSetOnly | ResultSetOnly |
|---------------|---------------|

Specifies that only the results of the command should be processed by the Always Encrypted routine in the driver. Use this value when the command has no parameters that require encryption.

|                      |                      |
|----------------------|----------------------|
| UseConnectionSetting | UseConnectionSetting |
|----------------------|----------------------|

Specifies that the command should default to the Always Encrypted setting in the connection string.

## See Also

# SqlConnection SqlConnection Class

Represents a connection to a SQL Server database. This class cannot be inherited.

## Declaration

```
public sealed class SqlConnection : System.Data.Common.DbConnection, ICloneable, IDisposable  
  
type SqlConnection = class  
    inherit DbConnection  
    interface IDbConnection  
    interface ICloneable  
    interface IDisposable
```

## Inheritance Hierarchy



## Remarks

A [SqlConnection](#) object represents a unique session to a SQL Server data source. With a client/server database system, it is equivalent to a network connection to the server. [SqlConnection](#) is used together with [SqlDataAdapter](#) and [SqlCommand](#) to increase performance when connecting to a Microsoft SQL Server database. For all third-party SQL Server products and other OLE DB-supported data sources, use [OleDbConnection](#).

When you create an instance of [SqlConnection](#), all properties are set to their initial values. For a list of these values, see the [SqlConnection](#) constructor.

See [ConnectionString](#) for a list of the keywords in a connection string.

If the [SqlConnection](#) goes out of scope, it won't be closed. Therefore, you must explicitly close the connection by calling `Close` or `Dispose`. `Close` and `Dispose` are functionally equivalent. If the connection pooling value `Pooling` is set to `true` or `yes`, the underlying connection is returned back to the connection pool. On the other hand, if `Pooling` is set to `false` or `no`, the underlying connection to the server is actually closed.

### Note

Login and logout events will not be raised on the server when a connection is fetched from or returned to the connection pool, because the connection is not actually closed when it is returned to the connection pool. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

To ensure that connections are always closed, open the connection inside of a `using` block, as shown in the following code fragment. Doing so ensures that the connection is automatically closed when the code exits the block.

```
using (SqlConnection connection = new SqlConnection(connectionString))  
{  
    connection.Open();  
    // Do work here; connection closed on following line.  
}
```

### Note

To deploy high-performance applications, you must use connection pooling. When you use the .NET Framework Data Provider for SQL Server, you do not have to enable connection pooling because the provider manages this automatically, although you can modify some settings. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

If a [SqlException](#) is generated by the method executing a [SqlCommand](#), the [SqlConnection](#) remains open when the severity level is 19 or less. When the severity level is 20 or greater, the server ordinarily closes the [SqlConnection](#). However, the user can reopen the connection and continue.

An application that creates an instance of the [SqlConnection](#) object can require all direct and indirect callers to have sufficient permission to the code by setting declarative or imperative security demands. [SqlConnection](#) makes security demands using the [SqlClientPermission](#) object. Users can verify that their code has sufficient permissions by using the [SqlClientPermissionAttribute](#) object. Users and administrators can also use the [Caspol.exe \(Code Access Security Policy Tool\)](#) to modify security policy at the machine, user, and enterprise levels. For more information, see [Security in .NET](#). For an example demonstrating how to use security demands, see [Code Access Security and ADO.NET](#).

For more information about handling warning and informational messages from the server, see [Connection Events](#). For more information about SQL Server engine errors and error messages, see [Database Engine Events and Errors](#).

**Caution**

You can force TCP instead of shared memory. You can do that by prefixing tcp: to the server name in the connection string or you can use localhost.

## Constructors

[SqlConnection\(\)](#)

[SqlConnection\(\)](#)

Initializes a new instance of the [SqlConnection](#) class.

[SqlConnection\(String\)](#)

[SqlConnection\(String\)](#)

Initializes a new instance of the [SqlConnection](#) class when given a string that contains the connection string.

[SqlConnection\(String, SqlCredential\)](#)

[SqlConnection\(String, SqlCredential\)](#)

Initializes a new instance of the [SqlConnection](#) class given a connection string, that does not use [Integrated Security = true](#) and a [SqlCredential](#) object that contains the user ID and password.

## Properties

[AccessToken](#)

[AccessToken](#)

Gets or sets the access token for the connection.

[ClientConnectionId](#)

[ClientConnectionId](#)

The connection ID of the most recent connection attempt, regardless of whether the attempt succeeded or failed.

[ColumnEncryptionKeyCacheTtl](#)

#### **ColumnEncryptionKeyCacheTtl**

Gets or sets the time-to-live for column encryption key entries in the column encryption key cache for the [Always Encrypted](#) feature. The default value is 2 hours. 0 means no caching at all.

#### **ColumnEncryptionQueryMetadataCacheEnabled**

##### **ColumnEncryptionQueryMetadataCacheEnabled**

Gets or sets a value that indicates whether query metadata caching is enabled (true) or not (false) for parameterized queries running against [Always Encrypted](#) enabled databases. The default value is true.

#### **ColumnEncryptionTrustedMasterKeyPaths**

##### **ColumnEncryptionTrustedMasterKeyPaths**

Allows you to set a list of trusted key paths for a database server. If while processing an application query the driver receives a key path that is not on the list, the query will fail. This property provides additional protection against security attacks that involve a compromised SQL Server providing fake key paths, which may lead to leaking key store credentials.

#### **ConnectionString**

##### **ConnectionString**

Gets or sets the string used to open a SQL Server database.

#### **ConnectionTimeout**

##### **ConnectionTimeout**

Gets the time to wait while trying to establish a connection before terminating the attempt and generating an error.

#### **Credential**

##### **Credential**

Gets or sets the [SqlCredential](#) object for this connection.

#### **Credentials**

##### **Credentials**

#### **Database**

##### **Database**

Gets the name of the current database or the database to be used after a connection is opened.

#### **DataSource**

## DataSource

Gets the name of the instance of SQL Server to which to connect.

## FireInfoMessageEventOnUserErrors

### FireInfoMessageEventOnUserErrors

Gets or sets the [FireInfoMessageEventOnUserErrors](#) property.

## PacketSize

### PacketSize

Gets the size (in bytes) of network packets used to communicate with an instance of SQL Server.

## ServerVersion

### ServerVersion

Gets a string that contains the version of the instance of SQL Server to which the client is connected.

## State

### State

Indicates the state of the [SqlConnection](#) during the most recent network operation performed on the connection.

## StatisticsEnabled

### StatisticsEnabled

When set to `true`, enables statistics gathering for the current connection.

## WorkstationId

### WorkstationId

Gets a string that identifies the database client.

## Methods

### BeginTransaction()

### BeginTransaction()

Starts a database transaction.

### BeginTransaction(IsolationLevel)

### BeginTransaction(IsolationLevel)

Starts a database transaction with the specified isolation level.

```
BeginTransaction(String)  
BeginTransaction(String)
```

Starts a database transaction with the specified transaction name.

```
BeginTransaction(IsolationLevel, String)  
BeginTransaction(IsolationLevel, String)
```

Starts a database transaction with the specified isolation level and transaction name.

```
ChangeDatabase(String)  
ChangeDatabase(String)
```

Changes the current database for an open [SqlConnection](#).

```
ChangePassword(String, String)  
ChangePassword(String, String)
```

Changes the SQL Server password for the user indicated in the connection string to the supplied new password.

```
ChangePassword(String, SqlCredential, SecureString)  
ChangePassword(String, SqlCredential, SecureString)
```

Changes the SQL Server password for the user indicated in the [SqlCredential](#) object.

```
ClearAllPools()  
ClearAllPools()
```

Empties the connection pool.

```
ClearPool(SqlConnection)  
ClearPool(SqlConnection)
```

Empties the connection pool associated with the specified connection.

```
Close()  
Close()
```

Closes the connection to the database. This is the preferred method of closing any open connection.

```
CreateCommand()  
CreateCommand()
```

Creates and returns a [SqlCommand](#) object associated with the [SqlConnection](#).

```
EnlistDistributedTransaction(ITransaction)
```

```
EnlistDistributedTransaction(ITransaction)
```

Enlists in the specified transaction as a distributed transaction.

```
EnlistTransaction(Transaction)
```

```
EnlistTransaction(Transaction)
```

Enlists in the specified transaction as a distributed transaction.

```
GetSchema()
```

```
GetSchema()
```

Returns schema information for the data source of this [SqlConnection](#). For more information about schema, see [SQL Server Schema Collections](#).

```
GetSchema(String)
```

```
GetSchema(String)
```

Returns schema information for the data source of this [SqlConnection](#) using the specified string for the schema name.

```
GetSchema(String, String[])
```

```
GetSchema(String, String[])
```

Returns schema information for the data source of this [SqlConnection](#) using the specified string for the schema name and the specified string array for the restriction values.

```
Open()
```

```
Open()
```

Opens a database connection with the property settings specified by the [ConnectionString](#).

```
OpenAsync(CancellationToken)
```

```
OpenAsync(CancellationToken)
```

An asynchronous version of [Open\(\)](#), which opens a database connection with the property settings specified by the [ConnectionString](#). The cancellation token can be used to request that the operation be abandoned before the connection timeout elapses. Exceptions will be propagated via the returned Task. If the connection timeout time elapses without successfully connecting, the returned Task will be marked as faulted with an Exception. The implementation returns a Task without blocking the calling thread for both pooled and non-pooled connections.

```
RegisterColumnEncryptionKeyStoreProviders(IDictionary<String,SqlColumnEncryptionKeyStoreProvider>)
```

```
RegisterColumnEncryptionKeyStoreProviders(IDictionary<String,SqlColumnEncryptionKeyStoreProvider>)
```

Registers the column encryption key store providers.

```
ResetStatistics()  
ResetStatistics()
```

If statistics gathering is enabled, all values are reset to zero.

```
RetrieveStatistics()  
RetrieveStatistics()
```

Returns a name value pair collection of statistics at the point in time the method is called.

## Events

```
InfoMessage  
InfoMessage
```

Occurs when SQL Server returns a warning or informational message.

```
StateChange  
StateChange
```

```
IDbConnection.BeginTransaction()  
IDbConnection.BeginTransaction()
```

```
IDbConnection.BeginTransaction(IsolationLevel)  
IDbConnection.BeginTransaction(IsolationLevel)
```

```
IDbConnection.CreateCommand()  
IDbConnection.CreateCommand()
```

```
ICloneable.Clone()  
ICloneable.Clone()
```

Creates a new object that is a copy of the current instance.

## See Also

# SqlConnection.AccessToken SqlConnection.AccessToken

## In this Article

Gets or sets the access token for the connection.

```
[System.ComponentModel.Browsable(false)]
public string AccessToken { get; set; }

member this.AAccessToken : string with get, set
```

Returns

[String String](#)

The access token for the connection.

Attributes

[BrowsableAttribute](#)

# SqlConnection.BeginTransaction SqlConnection.BeginTransaction

In this Article

## Overloads

|   |  |
|---|--|
| <a href="#">BeginTransaction()</a> <a href="#">BeginTransaction()</a>   | Starts a database transaction.   |
| <a href="#">BeginTransaction(IsolationLevel)</a> <a href="#">BeginTransaction(IsolationLevel)</a>                 | Starts a database transaction with the specified isolation level.                      |
| <a href="#">BeginTransaction(String)</a> <a href="#">BeginTransaction(String)</a>                                 | Starts a database transaction with the specified transaction name.                     |
| <a href="#">BeginTransaction(IsolationLevel, String)</a> <a href="#">BeginTransaction(IsolationLevel, String)</a> | Starts a database transaction with the specified isolation level and transaction name. |

## BeginTransaction() BeginTransaction()

Starts a database transaction.

```
public System.Data.SqlClient.SqlTransaction BeginTransaction ();
override this.BeginTransaction : unit -> System.Data.SqlClient.SqlTransaction
```

Returns

[SqlTransaction](#) [SqlTransaction](#)

An object representing the new transaction.

Exceptions

[SqlException](#) [SqlException](#)

Parallel transactions are not allowed when using Multiple Active Result Sets (MARS).

[InvalidOperationException](#) [InvalidOperationException](#)

Parallel transactions are not supported.

Examples

The following example creates a [SqlConnection](#) and a [SqlTransaction](#). It also demonstrates how to use the [BeginTransaction](#), a [Commit](#), and [Rollback](#) methods.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction("SampleTransaction");

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();

            // Attempt to commit the transaction.
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Commit Exception Type: {0}", ex.GetType());
            Console.WriteLine(" Message: {0}", ex.Message);

            // Attempt to roll back the transaction.
            try
            {
                transaction.Rollback();
            }
            catch (Exception ex2)
            {
                // This catch block will handle any errors that may have occurred
                // on the server that would cause the rollback to fail, such as
                // a closed connection.
                Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
                Console.WriteLine(" Message: {0}", ex2.Message);
            }
        }
    }
}

```

## Remarks

This command maps to the SQL Server implementation of BEGIN TRANSACTION.

You must explicitly commit or roll back the transaction using the [Commit](#) or [Rollback](#) method. To make sure that the .NET Framework Data Provider for SQL Server transaction management model performs correctly, avoid using other transaction management models, such as the one provided by SQL Server.

#### Note

If you do not specify an isolation level, the default isolation level is used. To specify an isolation level with the [BeginTransaction](#) method, use the overload that takes the `iso` parameter ([BeginTransaction](#)). The isolation level set for a transaction persists after the transaction is completed and until the connection is closed or disposed. Setting the isolation level to **Snapshot** in a database where the snapshot isolation level is not enabled does not throw an exception. The transaction will complete using the default isolation level.

#### Caution

If a transaction is started and a level 16 or higher error occurs on the server, the transaction will not be rolled back until the [Read](#) method is invoked. No exception is thrown on [ExecuteReader](#).

#### Caution

When your query returns a large amount of data and calls [BeginTransaction](#), a [SqlException](#) is thrown because SQL Server does not allow parallel transactions when using MARS. To avoid this problem, always associate a transaction with the command, the connection, or both before any readers are open.

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

#### See

[Transactions and Concurrency](#)

#### Also

[Connecting to a Data Source in ADO.NET](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

## BeginTransaction(IsolationLevel) BeginTransaction(IsolationLevel)

Starts a database transaction with the specified isolation level.

```
public System.Data.SqlClient.SqlTransaction BeginTransaction (System.Data.IsolationLevel iso);  
override this.BeginTransaction : System.Data.IsolationLevel -> System.Data.SqlClient.SqlTransaction
```

#### Parameters

##### iso

[IsolationLevel](#) [IsolationLevel](#)

The isolation level under which the transaction should run.

#### Returns

[SqlTransaction](#) [SqlTransaction](#)

An object representing the new transaction.

#### Exceptions

[SqlException](#) [SqlException](#)

Parallel transactions are not allowed when using Multiple Active Result Sets (MARS).

[InvalidOperationException](#) [InvalidOperationException](#)

Parallel transactions are not supported.

#### Examples

The following example creates a [SqlConnection](#) and a [SqlTransaction](#). It also demonstrates how to use the [BeginTransaction](#), a [Commit](#), and [Rollback](#) methods.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction(IsolationLevel.ReadCommitted);

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception e)
        {
            try
            {
                transaction.Rollback();
            }
            catch (SqlException ex)
            {
                if (transaction.Connection != null)
                {
                    Console.WriteLine("An exception of type " + ex.GetType() +
                        " was encountered while attempting to roll back the transaction.");
                }
            }

            Console.WriteLine("An exception of type " + e.GetType() +
                " was encountered while inserting the data.");
            Console.WriteLine("Neither record was written to database.");
        }
    }
}

```

## Remarks

This command maps to the SQL Server implementation of BEGIN TRANSACTION.

You must explicitly commit or roll back the transaction using the [Commit](#) or [Rollback](#) method. To make sure that the .NET Framework Data Provider for SQL Server transaction management model performs correctly, avoid using other transaction management models, such as the one provided by SQL Server.

#### Note

After a transaction is committed or rolled back, the isolation level of the transaction persists for all subsequent commands that are in autocommit mode (the SQL Server default). This can produce unexpected results, such as an isolation level of REPEATABLE READ persisting and locking other users out of a row. To reset the isolation level to the default (READ COMMITTED), execute the Transact-SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement, or call [SqlConnection.BeginTransaction](#) followed immediately by [SqlTransaction.Commit](#). For more information on SQL Server isolation levels, see [Transaction Isolation Levels](#).

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

#### Caution

When your query returns a large amount of data and calls `BeginTransaction`, a [SqlException](#) is thrown because SQL Server does not allow parallel transactions when using MARS. To avoid this problem, always associate a transaction with the command, the connection, or both before any readers are open.

#### See

[Transactions \(ADO.NET\)](#)

#### Also

[Connecting to a Data Source \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## BeginTransaction(String) BeginTransaction(String)

Starts a database transaction with the specified transaction name.

```
public System.Data.SqlClient.SqlTransaction BeginTransaction (string transactionName);  
override this.BeginTransaction : string -> System.Data.SqlClient.SqlTransaction
```

#### Parameters

##### transactionName

[String](#) [String](#)

The name of the transaction.

#### Returns

[SqlTransaction](#) [SqlTransaction](#)

An object representing the new transaction.

#### Exceptions

[SqlException](#) [SqlException](#)

Parallel transactions are not allowed when using Multiple Active Result Sets (MARS).

[InvalidOperationException](#) [InvalidOperationException](#)

Parallel transactions are not supported.

#### Examples

The following example creates a [SqlConnection](#) and a [SqlTransaction](#). It also demonstrates how to use the [BeginTransaction](#), a [Commit](#), and [Rollback](#) methods.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction("SampleTransaction");

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();

            // Attempt to commit the transaction.
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Commit Exception Type: {0}", ex.GetType());
            Console.WriteLine(" Message: {0}", ex.Message);

            // Attempt to roll back the transaction.
            try
            {
                transaction.Rollback();
            }
            catch (Exception ex2)
            {
                // This catch block will handle any errors that may have occurred
                // on the server that would cause the rollback to fail, such as
                // a closed connection.
                Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
                Console.WriteLine(" Message: {0}", ex2.Message);
            }
        }
    }
}

```

## Remarks

This command maps to the SQL Server implementation of BEGIN TRANSACTION.

The length of the `transactionName` parameter must not exceed 32 characters; otherwise an exception will be thrown.

The value in the `transactionName` parameter can be used in later calls to [Commit](#) and in the `savePoint` parameter of the [Save](#) method.

You must explicitly commit or roll back the transaction using the [Commit](#) or [Rollback](#) method. To make sure that the .NET Framework Data Provider for SQL Server transaction management model performs correctly, avoid using other transaction management models, such as the one provided by SQL Server.

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

**Caution**

When your query returns a large amount of data and calls `BeginTransaction`, a `SqlException` is thrown because SQL Server does not allow parallel transactions when using MARS. To avoid this problem, always associate a transaction with the command, the connection, or both before any readers are open.

**See**

[Transactions \(ADO.NET\)](#)

**Also**

[Connecting to a Data Source \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **BeginTransaction(IsolationLevel, String)**

## **BeginTransaction(IsolationLevel, String)**

Starts a database transaction with the specified isolation level and transaction name.

```
public System.Data.SqlClient.SqlTransaction BeginTransaction (System.Data.IsolationLevel iso, string  
transactionName);  
  
override this.BeginTransaction : System.Data.IsolationLevel * string ->  
System.Data.SqlClient.SqlTransaction
```

**Parameters**

**iso**

[IsolationLevel IsolationLevel](#)

The isolation level under which the transaction should run.

**transactionName**

[String String](#)

The name of the transaction.

**Returns**

[SqlTransaction SqlTransaction](#)

An object representing the new transaction.

**Exceptions**

[SqlException SqlException](#)

Parallel transactions are not allowed when using Multiple Active Result Sets (MARS).

[InvalidOperationException InvalidOperationException](#)

Parallel transactions are not supported.

**Examples**

The following example creates a `SqlConnection` and a `SqlTransaction`. It also demonstrates how to use the `BeginTransaction`, a `Commit`, and `Rollback` methods.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction(
            IsolationLevel.ReadCommitted, "SampleTransaction");

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction.
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception e)
        {
            try
            {
                transaction.Rollback();
            }
            catch (SqlException ex)
            {
                if (transaction.Connection != null)
                {
                    Console.WriteLine("An exception of type " + ex.GetType() +
                        " was encountered while attempting to roll back the transaction.");
                }
            }

            Console.WriteLine("An exception of type " + e.GetType() +
                " was encountered while inserting the data.");
            Console.WriteLine("Neither record was written to database.");
        }
    }
}
}

```

## Remarks

This command maps to the SQL Server implementation of BEGIN TRANSACTION.

The value in the `transactionName` parameter can be used in later calls to `Rollback` and in the `savePoint` parameter of the `Save` method.

You must explicitly commit or roll back the transaction using the `Commit` or `Rollback` method. To make sure that the SQL Server transaction management model performs correctly, avoid using other transaction management models, such as the one provided by SQL Server.

**Note**

After a transaction is committed or rolled back, the isolation level of the transaction persists for all subsequent commands that are in autocommit mode (the SQL Server default). This can produce unexpected results, such as an isolation level of REPEATABLE READ persisting and locking other users out of a row. To reset the isolation level to the default (READ COMMITTED), execute the Transact-SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED statement, or call [SqlConnection.BeginTransaction](#) followed immediately by [SqlTransaction.Commit](#). For more information on SQL Server isolation levels, see [Transaction Isolation Levels](#).

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

**Caution**

When your query returns a large amount of data and calls `BeginTransaction`, a [SqlException](#) is thrown because SQL Server does not allow parallel transactions when using MARS. To avoid this problem, always associate a transaction with the command, the connection, or both before any readers are open.

See

[Transactions \(ADO.NET\)](#)

Also

[Connecting to a Data Source \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlConnection.ChangeDatabase SqlConnection.ChangeDatabase

## In this Article

Changes the current database for an open [SqlConnection](#).

```
public override void ChangeDatabase (string database);  
override this.ChangeDatabase : string -> unit
```

## Parameters

database [String](#) [String](#)

The name of the database to use instead of the current database.

## Exceptions

[ArgumentException](#) [ArgumentException](#)

The database name is not valid.

[InvalidOperationException](#) [InvalidOperationException](#)

The connection is not open.

[SqlException](#) [SqlException](#)

Cannot change the database.

## Examples

The following example creates a [SqlConnection](#) and displays some of its read-only properties.

```
private static void ChangeSqlDatabase(string connectionString)  
{  
    // Assumes connectionString represents a valid connection string  
    // to the AdventureWorks sample database.  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        connection.Open();  
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);  
        Console.WriteLine("Database: {0}", connection.Database);  
  
        connection.ChangeDatabase("Northwind");  
        Console.WriteLine("Database: {0}", connection.Database);  
    }  
}
```

## Remarks

The value supplied in the `database` parameter must be a valid database name. The `database` parameter cannot contain a null value, an empty string, or a string with only blank characters.

See

[Connecting to a Data Source in ADO.NET](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlConnection.ChangePassword SqlConnection.ChangePassword

In this Article

## Overloads

|   |   |
|---|---|
| <code>ChangePassword(String, String)</code> <code>ChangePassword(String, String)</code>   | Changes the SQL Server password for the user indicated in the connection string to the supplied new password. |
| <code>ChangePassword(String, SqlCredential, SecureString)</code> <code>ChangePassword(String, SqlCredential, SecureString)</code> | Changes the SQL Server password for the user indicated in the <code>SqlCredential</code> object.              |

## ChangePassword(String, String) ChangePassword(String, String)

Changes the SQL Server password for the user indicated in the connection string to the supplied new password.

```
public static void ChangePassword (string connectionString, string newPassword);  
static member ChangePassword : string * string -> unit
```

Parameters

connectionString `String` `String`

The connection string that contains enough information to connect to the server that you want. The connection string must contain the user ID and the current password.

newPassword `String` `String`

The new password to set. This password must comply with any password security policy set on the server, including minimum length, requirements for specific characters, and so on.

Exceptions

`ArgumentException` `ArgumentException`

The connection string includes the option to use integrated security.

Or

The `newPassword` exceeds 128 characters.

`ArgumentNullException` `ArgumentNullException`

Either the `connectionString` or the `newPassword` parameter is null.

Examples

The following is a simple example of changing a password:

```

class Program {
    static void Main(string[] args) {
        System.Data.SqlClient.SqlConnection.ChangePassword(
            "Data Source=a_server;Initial Catalog=a_database;UID=user;PWD=old_password",
            "new_password");
    }
}

```

```

Module Module1
    Sub Main()
        System.Data.SqlClient.SqlConnection.ChangePassword(
            "Data Source=a_server;Initial Catalog=a_database;UID=user;PWD=old_password",
            "new_password")
    End Sub
End Module

```

The following console application demonstrates the issues involved in changing a user's password because the current password has expired.

```

using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        try
        {
            DemonstrateChangePassword();
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }
        Console.WriteLine("Press ENTER to continue...");
        Console.ReadLine();
    }

    private static void DemonstrateChangePassword()
    {
        // Retrieve the connection string. In a production application,
        // this string should not be contained within the source code.
        string connectionString = GetConnectionString();

        using (SqlConnection cnn = new SqlConnection())
        {
            for (int i = 0; i <= 1; i++)
            {
                // Run this loop at most two times. If the first attempt fails,
                // the code checks the Number property of the SQLException object.
                // If that contains the special values 18487 or 18488, the code
                // attempts to set the user's password to a new value.
                // Assuming this succeeds, the second pass through
                // successfully opens the connection.
                // If not, the exception handler catches the exception.
                try
                {
                    cnn.ConnectionString = connectionString;
                    cnn.Open();
                    // Once this succeeds, just get out of the loop.
                    // No need to try again if the connection is already open.
                    break;
                }
            }
        }
    }
}

```

```

        catch (SqlException ex)
        {
            if (i == 0 && ((ex.Number == 18487) || (ex.Number == 18488)))
            {
                // You must reset the password.
                connectionString =
                    ModifyConnectionString(connectionString,
                        GetNewPassword());
            }
            else
                // Bubble all other SqlException occurrences
                // back up to the caller.
                throw;
        }
    }

    SqlCommand cmd = new SqlCommand(
        "SELECT ProductID, Name FROM Product", cnn);
    // Use the connection and command here...
}

private static string ModifyConnectionString(
    string connectionString, string NewPassword)
{
    // Use the SqlConnectionStringBuilder class to modify the
    // password portion of the connection string.
    SqlConnectionStringBuilder builder =
        new SqlConnectionStringBuilder(connectionString);
    builder.Password = NewPassword;
    return builder.ConnectionString;
}

private static string GetNewPassword()
{
    // In a real application, you might display a modal
    // dialog box to retrieve the new password. The concepts
    // are the same as for this simple console application, however.
    Console.Write("Your password must be reset. Enter a new password: ");
    return Console.ReadLine();
}

private static string GetConnectionString()
{
    // For this demonstration, the connection string must
    // contain both user and password information. In your own
    // application, you might want to retrieve this setting
    // from a config file, or from some other source.

    // In a production application, you would want to
    // display a modal form that could gather user and password
    // information.
    SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(
        "Data Source=(local);Initial Catalog=AdventureWorks");

    Console.Write("Enter your user id: ");
    builder.UserID = Console.ReadLine();
    Console.Write("Enter your password: ");
    builder.Password = Console.ReadLine();

    return builder.ConnectionString;
}
}

```

## Remarks

When you are using SQL Server on Windows Server, developers can take advantage of functionality that lets the client application supply both the current and a new password in order to change the existing password. Applications can implement functionality such as prompting the user for a new password during initial login if the old one has expired, and this operation can be completed without administrator intervention.

The [ChangePassword](#) method changes the SQL Server password for the user indicated in the supplied `connectionString` parameter to the value supplied in the `newPassword` parameter. If the connection string includes the option for integrated security (that is, "Integrated Security=True" or the equivalent), an exception is thrown.

To determine that the password has expired, calling the [Open](#) method raises a [SqlException](#). In order to indicate that the password that is contained within the connection string must be reset, the [Number](#) property for the exception contains the status value 18487 or 18488. The first value (18487) indicates that the password has expired and the second (18488) indicates that the password must be reset before logging in.

This method opens its own connection to the server, requests the password change, and closes the connection as soon as it has completed. This connection is not retrieved from, nor returned to, the SQL Server connection pool.

See

[Connection Strings \(ADO.NET\)](#)

Also

[Connecting to a Data Source \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **ChangePassword(String, SqlCredential, SecureString)**

## **ChangePassword(String, SqlCredential, SecureString)**

Changes the SQL Server password for the user indicated in the [SqlCredential](#) object.

```
public static void ChangePassword (string connectionString, System.Data.SqlClient.SqlCredential  
credential, System.Security.SecureString newSecurePassword);  
  
static member ChangePassword : string * System.Data.SqlClient.SqlCredential *  
System.Security.SecureString -> unit
```

### Parameters

connectionString

[String](#)

The connection string that contains enough information to connect to a server. The connection string should not use any of the following connection string keywords: `Integrated Security = true`, `UserId`, or `Password`; or `ContextConnection = true`.

credential

[SqlCredential](#)

A [SqlCredential](#) object.

newPassword

[SecureString](#)

### Exceptions

[ArgumentException](#)

1. The connection string contains any combination of `UserId`, `Password`, or `Integrated Security=true`.
2. The connection string contains `Context Connection=true`.
3. `newSecurePassword` is greater than 128 characters.

4. `newSecurePassword` is not read only.

5. `newSecurePassword` is an empty string.

#### [ArgumentNullException](#) [ArgumentNullException](#)

One of the parameters (`connectionString`, `credential`, or `newSecurePassword`) is null.

See

Also

[ADO.NET Overview](#)

# SqlConnection.ClearAllPools SqlConnection.ClearAllPools

## In this Article

Empties the connection pool.

```
public static void ClearAllPools ();  
static member ClearAllPools : unit -> unit
```

## Remarks

[ClearAllPools](#) resets (or empties) the connection pool. If there are connections in use at the time of the call, they are marked appropriately and will be discarded (instead of being returned to the pool) when [Close](#) is called on them.

See

[SQL Server Connection Pooling \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.ClearPool SqlConnection.ClearPool

## In this Article

Empties the connection pool associated with the specified connection.

```
public static void ClearPool (System.Data.SqlClient.SqlConnection connection);  
static member ClearPool : System.Data.SqlClient.SqlConnection -> unit
```

## Parameters

connection [SqlConnection SqlConnection](#)

The [SqlConnection](#) to be cleared from the pool.

## Remarks

[ClearPool](#) clears the connection pool that is associated with the `connection`. If additional connections associated with `connection` are in use at the time of the call, they are marked appropriately and are discarded (instead of being returned to the pool) when [Close](#) is called on them.

See

[SQL Server Connection Pooling \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.ClientConnectionId SqlConnection.ClientConnectionId

## In this Article

The connection ID of the most recent connection attempt, regardless of whether the attempt succeeded or failed.

```
public Guid ClientConnectionId { get; }  
member this.ClientConnectionId : Guid
```

Returns

[Guid](#)

The connection ID of the most recent connection attempt.

## Remarks

[ClientConnectionId](#) works regardless of which version of the server you connect to, but extended events logs and entry on connectivity ring buffer errors will not be present in SQL Server 2008 R2 and earlier.

You can locate the connection ID in the extended events log to see if the failure was on the server if the extended event for logging connection ID is enabled. You can also locate the connection ID in the connection ring buffer ([Connectivity troubleshooting in SQL Server 2008 with the Connectivity Ring Buffer](#)) for certain connection errors. If the connection ID is not in the connection ring buffer, you can assume a network error.

See

[ADO.NET Overview](#)

Also

# SqlConnection.Close SqlConnection.Close

## In this Article

Closes the connection to the database. This is the preferred method of closing any open connection.

```
public override void Close ();  
override this.Close : unit -> unit
```

## Exceptions

[SqlException](#) [SqlException](#)

The connection-level error that occurred while opening the connection.

## Examples

The following example creates a [SqlConnection](#), opens it, displays some of its properties. The connection is automatically closed at the end of the `using` block.

```
private static void OpenSqlConnection(string connectionString)  
{  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        connection.Open();  
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);  
        Console.WriteLine("State: {0}", connection.State);  
    }  
}
```

## Remarks

The [Close](#) method rolls back any pending transactions. It then releases the connection to the connection pool, or closes the connection if connection pooling is disabled.

### Note

Pending transactions started using Transact-SQL or [BeginTransaction](#) are automatically rolled back when the connection is reset if connection pooling is enabled. If connection pooling is off, the transaction is rolled back after [SqlConnection.Close](#) is called. Transactions started through [System.Transactions](#) are controlled through the [System.Transactions](#) infrastructure, and are not affected by [SqlConnection.Close](#).

An application can call [Close](#) more than one time. No exception is generated.

If the [SqlConnection](#) goes out of scope, it won't be closed. Therefore, you must explicitly close the connection by calling [Close](#) or [Dispose](#). [Close](#) and [Dispose](#) are functionally equivalent. If the connection pooling value [Pooling](#) is set to [true](#) or [yes](#), the underlying connection is returned back to the connection pool. On the other hand, if [Pooling](#) is set to [false](#) or [no](#), the underlying connection to the server is closed.

### Note

Login and logout events will not be raised on the server when a connection is fetched from or returned to the connection pool, because the connection is not actually closed when it is returned to the connection pool. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

### Caution

Do not call [Close](#) or [Dispose](#) on a Connection, a DataReader, or any other managed object in the [Finalize](#) method of your class. In a finalizer, you should only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a [Finalize](#) method in your class definition. For more

information, see [Garbage Collection](#).

See

Also

[SQL Server Connection Pooling \(ADO.NET\)](#)

[Connecting to a Data Source in ADO.NET](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlConnection.ColumnEncryptionKeyCacheTtl SqlConnection.ColumnEncryptionKeyCacheTtl

## In this Article

Gets or sets the time-to-live for column encryption key entries in the column encryption key cache for the [Always Encrypted](#) feature. The default value is 2 hours. 0 means no caching at all.

```
public static TimeSpan ColumnEncryptionKeyCacheTtl { get; set; }  
member this.ColumnEncryptionKeyCacheTtl : TimeSpan with get, set
```

## Returns

[TimeSpan](#) [TimeSpan](#)

The time interval.

# SqlConnection.ColumnEncryptionQueryMetadataCacheEnabled

## SqlConnection.ColumnEncryptionQueryMetadataCacheEnabled

### In this Article

Gets or sets a value that indicates whether query metadata caching is enabled (true) or not (false) for parameterized queries running against [Always Encrypted](#) enabled databases. The default value is true.

```
public static bool ColumnEncryptionQueryMetadataCacheEnabled { get; set; }  
member this.ColumnEncryptionQueryMetadataCacheEnabled : bool with get, set
```

### Returns

[Boolean](#) [Boolean](#)

Returns true if query metadata caching is enabled; otherwise false. true is the default.

## Remarks

For parameterized queries, `SqlClient` makes a roundtrip to SQL Server for parameter metadata, to see which parameter it needs to encrypt and how (which keys and algorithms should be used). If the application calls the same query multiple times, an extra roundtrip is made to the server each time, which degrades application performance.

With **ColumnEncryptionQueryMetadataCacheEnabled** set to true, if the same query is called multiple times, the roundtrip to the server will be made only once. The cache has a non-configurable Max size parameter that is set to 2000 queries.

# SqlConnection.ColumnEncryptionTrustedMasterKeyPaths SqlConnection.ColumnEncryptionTrustedMasterKeyPaths

## In this Article

Allows you to set a list of trusted key paths for a database server. If while processing an application query the driver receives a key path that is not on the list, the query will fail. This property provides additional protection against security attacks that involve a compromised SQL Server providing fake key paths, which may lead to leaking key store credentials.

```
public static  
System.Collections.Generic.IDictionary<string, System.Collections.Generic.IList<string>>  
ColumnEncryptionTrustedMasterKeyPaths { get; }  
  
member this.ColumnEncryptionTrustedMasterKeyPaths : System.Collections.Generic.IDictionary<string,  
System.Collections.Generic.IList<string>>
```

## Returns

[IDictionary<String,IList<String>>](#)

The list of trusted master key paths for the column encryption.

# SqlConnection.ConnectionString SqlConnection.ConnectionString

## In this Article

Gets or sets the string used to open a SQL Server database.

```
[System.Data.DataSysDescription("SqlConnection_ConnectionString")]
[System.ComponentModel.SettingsBindable(true)]
public override string ConnectionString { get; set; }

member this.ConnectionString : string with get, set
```

Returns

[String String](#)

The connection string that includes the source database name, and other parameters needed to establish the initial connection. The default value is an empty string.

Attributes

[DataSysDescriptionAttribute](#) [SettingsBindableAttribute](#)

Exceptions

[ArgumentException](#) [ArgumentException](#)

An invalid connection string argument has been supplied, or a required connection string argument has not been supplied.

## Examples

The following example creates a [SqlConnection](#) and sets the [ConnectionString](#) property before opening the connection.

```
private static void OpenSqlConnection()
{
    string connectionString = GetConnectionString();

    using (SqlConnection connection = new SqlConnection())
    {
        connection.ConnectionString = connectionString;

        connection.Open();

        Console.WriteLine("State: {0}", connection.State);
        Console.WriteLine("ConnectionString: {0}",
            connection.ConnectionString);
    }
}

static private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file.
    return "Data Source=MSSQL1;Initial Catalog=AdventureWorks";
    + "Integrated Security=true;";
}
```

## Remarks

The [ConnectionString](#) is similar to an OLE DB connection string, but is not identical. Unlike OLE DB or ADO, the connection string that is returned is the same as the user-set [ConnectionString](#), minus security information if the Persist Security Info value is set to `false` (default). The .NET Framework Data Provider for SQL Server does not persist or return the password in a connection string unless you set Persist Security Info to `true`.

You can use the [ConnectionString](#) property to connect to a database. The following example illustrates a typical connection string.

```
"Persist Security Info=False;Integrated Security=true;Initial Catalog=Northwind;server=(local)"
```

Use the new [SqlConnectionStringBuilder](#) to construct valid connection strings at run time. For more information, see [Connection String Builders](#).

The [ConnectionString](#) property can be set only when the connection is closed. Many of the connection string values have corresponding read-only properties. When the connection string is set, these properties are updated, except when an error is detected. In this case, none of the properties are updated. [SqlConnection](#) properties return only those settings that are contained in the [ConnectionString](#).

To connect to a local computer, specify "(local)" for the server. If a server name is not specified, a connection will be attempted to the default

instance on the local computer.

Resetting the [ConnectionString](#) on a closed connection resets all connection string values (and related properties) including the password. For example, if you set a connection string that includes "Database= AdventureWorks", and then reset the connection string to "Data Source=myserver;Integrated Security=true", the [Database](#) property is no longer set to "AdventureWorks".

The connection string is parsed immediately after being set. If errors in syntax are found when parsing, a runtime exception, such as [ArgumentException](#), is generated. Other errors can be found only when an attempt is made to open the connection.

The basic format of a connection string includes a series of keyword/value pairs separated by semicolons. The equal sign (=) connects each keyword and its value. To include values that contain a semicolon, single-quote character, or double-quote character, the value must be enclosed in double quotation marks. If the value contains both a semicolon and a double-quote character, the value can be enclosed in single quotation marks. The single quotation mark is also useful if the value starts with a double-quote character. Conversely, the double quotation mark can be used if the value starts with a single quotation mark. If the value contains both single-quote and double-quote characters, the quotation mark character used to enclose the value must be doubled every time it occurs within the value.

To include preceding or trailing spaces in the string value, the value must be enclosed in either single quotation marks or double quotation marks. Any leading or trailing spaces around integer, Boolean, or enumerated values are ignored, even if enclosed in quotation marks. However, spaces within a string literal keyword or value are preserved. Single or double quotation marks may be used within a connection string without using delimiters (for example, Data Source= my'Server or Data Source= my"Server), unless a quotation mark character is the first or last character in the value.

Keywords are not case sensitive.

The following table lists the valid names for keyword values within the [ConnectionString](#).

| KEYWORD                                  | DEFAULT   | DESCRIPTION  |
|--|-----------|--|
| Addr                                     | N/A       | Synonym of <a href="#">Data Source</a> .   |
| Address                                  | N/A       | Synonym of <a href="#">Data Source</a> .   |
| App                                      | N/A       | Synonym of <a href="#">Application Name</a> .  |
| Application Name                         | N/A       | <p>The name of the application, or '.NET SQLClient Data Provider' if no application name is provided.</p> <p>An application name can be 128 characters or less.</p>  |
| ApplicationIntent                        | ReadWrite | <p>Declares the application workload type when connecting to a server. Possible values are <code>ReadOnly</code> and <code>ReadWrite</code>. For example:</p> <p><code>ApplicationIntent=ReadOnly</code></p> <p>For more information about <a href="#">SqlClient Support for High Availability, Disaster Recovery</a>.</p>   |
| Asynchronous Processing<br>-or-<br>Async | 'false'   | <p>When <code>true</code>, enables asynchronous operation support. Recognized values are <code>true</code>, <code>false</code>, <code>yes</code>, and <code>no</code>.</p> <p>This property is ignored beginning in .NET Framework 4.5. For more information about <a href="#">SqlClient support for asynchronous programming</a>, see <a href="#">Asynchronous Programming</a>.</p> |

| KEYWORD  | DEFAULT | DESCRIPTION   |
|--|---------|---|
| AttachDBfilename<br>-or-<br>Extended Properties<br>-or-<br>Initial File Name | N/A     | <p>The name of the primary database file, including the full path name of an attachable database. AttachDBfilename is only supported for primary data files with an .mdf extension.</p> <p>If the value of the AttachDBFileName key is specified in the connection string, the database is attached and becomes the default database for the connection.</p> <p>If this key is not specified and if the database was previously attached, the database will not be reattached. The previously attached database will be used as the default database for the connection.</p> <p>If this key is specified together with the AttachDBFileName key, the value of this key will be used as the alias. However, if the name is already used in another attached database, the connection will fail.</p> <p>The path may be absolute or relative by using the DataDirectory substitution string. If DataDirectory is used, the database file must exist within a subdirectory of the directory pointed to by the substitution string. <b>Note:</b> Remote server, HTTP, and UNC path names are not supported.</p> <p>The database name must be specified with the keyword 'database' (or one of its aliases) as in the following:</p> <pre>"AttachDbFileName= DataDirectory \data\YourDB security=true;database=YourDatabase"</pre> |
| Authentication   | N/A     | <p>An error will be generated if a log file exists in the same directory as the data file and the 'database' keyword is used when attaching the primary data file. In this case, remove the log file. Once the database is attached, a new log file will be automatically generated based on the physical path.</p>   |
| Column Encryption Setting  | N/A     | <p>The authentication method used for <a href="#">Connecting to SQL Database By Using Azure Active Directory Authentication</a>.</p> <p>Valid values are:<br/>Active Directory Integrated, Active Directory Password, Sql Password.</p>   |
| Connect Timeout<br>-or-<br>Connection Timeout<br>-or-<br>Timeout             | 15      | <p>Enables or disables <a href="#">Always Encrypted</a> functionality for the connection.</p> <p>The length of time (in seconds) to wait for a connection to the server before terminating the attempt and generating an error.</p> <p>Valid values are greater than or equal to 0 and less than or equal to 2147483647.</p> <p>When opening a connection to a Azure SQL Database, set the connection timeout to 30 seconds.</p>  |

| KEYWORD   | DEFAULT | DESCRIPTION   |
|---|---------|---|
| Connection Lifetime<br>-or-<br>Load Balance Timeout | 0       | <p>When a connection is returned to the pool, its creation time is compared with the current time, and the connection is destroyed if that time span (in seconds) exceeds the value specified by <code>Connection Lifetime</code>. This is useful in clustered configurations to force load balancing between a running server and a server just brought online.</p> <p>A value of zero (0) causes pooled connections to have the maximum connection timeout.</p>   |
| ConnectRetryCount                                   | 1       | <p>Controls the number of reconnection attempts after the client identifies an idle connection failure. Valid values are 0 to 255. The default is 1. 0 means do not attempt to reconnect (disable connection resiliency).</p> <p>For additional information about idle connection resiliency, see <a href="#">Technical Article – Idle Connection Resiliency</a>.</p>   |
| ConnectRetryInterval                                | 10      | <p>Specifies the time between each connection retry attempt (ConnectRetryCount). Valid values are 1 to 60 seconds (default=10), applied after the first reconnection attempt. When a broken connection is detected, the client immediately attempts to reconnect; this is the first reconnection attempt and only occurs if ConnectRetryCount is greater than 0. If the first reconnection attempt fails and ConnectRetryCount is greater than 1, the client waits ConnectRetryInterval to try the second and subsequent reconnection attempts.</p> <p>For additional information about idle connection resiliency, see <a href="#">Technical Article – Idle Connection Resiliency</a>.</p> |
| Context Connection                                  | 'false' | <code>true</code> if an in-process connection to SQL Server should be made.   |
| Current Language<br>-or-<br>Language                | N/A     | <p>Sets the language used for database server warning or error messages.</p> <p>The language name can be 128 characters or less.</p>  |

| KEYWORD                       | DEFAULT | DESCRIPTION   |
|-------------------------------|---------|---|
| Data Source<br>-or-<br>Server | N/A     | The name or network address of the instance of SQL Server to which to connect. The port number can be specified after the server name:<br><br><code>server=tcp:servername, portnumber</code>  |
| -or-<br>Address               |         | When specifying a local instance, always use (local). To force a protocol, add one of the following prefixes:<br><br><code>np:(local), tcp:(local), lpc:(local)</code>  |
| -or-<br>Addr                  |         | Beginning in .NET Framework 4.5, you can also connect to a LocalDB database as follows:<br><br><code>server=(localdb)\myInstance</code>   |
| Network Address               |         | <p>For more information about LocalDB, see <a href="#">SqlClient Support for LocalDB</a>.</p> <p><b>Data Source</b> must use the TCP format or the Named Pipes format.</p> <p>TCP format is as follows:</p> <ul style="list-style-type: none"> <li>- <code>tcp:&lt;host name&gt;\&lt;instance name&gt;</code></li> <li>- <code>tcp:&lt;host name&gt;,&lt;TCP/IP port number&gt;</code></li> </ul> <p>The TCP format must start with the prefix "tcp:" and is followed by the database instance, as specified by a host name and an instance name. This format is not applicable when connecting to Azure SQL Database. TCP is automatically selected for connections to Azure SQL Database when no protocol is specified.</p> <p>The host name MUST be specified in one of the following ways:</p> <ul style="list-style-type: none"> <li>- NetBIOSName</li> <li>- IPv4Address</li> <li>- IPv6Address</li> </ul> <p>The instance name is used to resolve to a particular TCP/IP port number on which a database instance is hosted. Alternatively, specifying a TCP/IP port number directly is also allowed. If both instance name and port number are not present, the default database instance is used.</p> <p>The Named Pipes format is as follows:</p> <ul style="list-style-type: none"> <li>- <code>np:\\&lt;host name&gt;\pipe\&lt;pipe name&gt;</code></li> </ul> <p>The Named Pipes format MUST start with the prefix "np:" and is followed by a named pipe name.</p> <p>The host name MUST be specified in one of the following ways:</p> <ul style="list-style-type: none"> <li>- NetBIOSName</li> <li>- IPv4Address</li> <li>- IPv6Address</li> </ul> <p>The pipe name is used to identify the database instance to which the .NET Framework application will be connected.</p> <p>If the value of the <b>Network</b> key is specified, the prefixes "tcp:" and "np:" should not be specified.</p> <p><b>Note:</b> You can force the use of TCP instead of shared memory, either by prefixing <b>tcp:</b> to the server name in the connection string, or by using <b>localhost</b>.</p> |

| KEYWORD   | DEFAULT | DESCRIPTION   |
|---|---------|---|
| Encrypt   | 'false' | <p>When <code>true</code>, SQL Server uses SSL encryption for all data sent between the client and server if the server has a certificate installed. Recognized values are <code>true</code>, <code>false</code>, <code>yes</code>, and <code>no</code>. For more information, see <a href="#">Connection String Syntax</a>.</p> <p>Beginning in .NET Framework 4.5, when <code>TrustServerCertificate</code> is false and <code>Encrypt</code> is true, the server name (or IP address) in a SQL Server SSL certificate must exactly match the server name (or IP address) specified in the connection string. Otherwise, the connection attempt will fail. For information about support for certificates whose subject starts with a wildcard character (*), see <a href="#">Accepted wildcards used by server certificates for server authentication</a>.</p> |
| Enlist  | 'true'  | <p><code>true</code> indicates that the SQL Server connection pooler automatically enlists the connection in the creation thread's current transaction context.</p>   |
| Failover Partner                                  | N/A     | <p>The name of the failover partner server where database mirroring is configured.</p> <p>If the value of this key is "", then <b>Initial Catalog</b> must be present, and its value must not be "".</p> <p>The server name can be 128 characters or less.</p> <p>If you specify a failover partner but the failover partner server is not configured for database mirroring and the primary server (specified with the <code>Server</code> keyword) is not available, then the connection will fail.</p> <p>If you specify a failover partner and the primary server is not configured for database mirroring, the connection to the primary server (specified with the <code>Server</code> keyword) will succeed if the primary server is available.</p>  |
| Initial Catalog<br>-or-<br>Database               | N/A     | <p>The name of the database.</p> <p>The database name can be 128 characters or less.</p>  |
| Integrated Security<br>-or-<br>Trusted_Connection | 'false' | <p>When <code>False</code>, User ID and Password are specified in the connection. When <code>true</code>, the current Windows account credentials are used for authentication.</p> <p>Recognized values are <code>true</code>, <code>false</code>, <code>yes</code>, <code>no</code>, and <code>sspi</code> (strongly recommended), which is equivalent to <code>true</code>.</p> <p>If User ID and Password are specified and Integrated Security is set to true, the User ID and Password will be ignored and Integrated Security will be used.</p> <p><code>SqlCredential</code> is a more secure way to specify credentials for a connection that uses SQL Server Authentication (<code>Integrated Security=false</code>).</p>  |
| Max Pool Size                                     | 100     | <p>The maximum number of connections that are allowed in the pool.</p> <p>Valid values are greater than or equal to 1. Values that are less than <b>Min Pool Size</b> generate an error.</p>  |

| Keyword   | Default | Description   |
|---|---------|---|
| Min Pool Size                                     | 0       | <p>The minimum number of connections that are allowed in the pool.</p> <p>Valid values are greater than or equal to 0. Zero (0) in this field means no minimum connections are initially opened.</p> <p>Values that are greater than <b>Max Pool Size</b> generate an error.</p>  |
| MultipleActiveResultSets                          | 'false' | <p>When <code>true</code>, an application can maintain multiple active result sets (MARS). When <code>false</code>, an application must process or cancel all result sets from one batch before it can execute any other batch on that connection.</p> <p>Recognized values are <code>true</code> and <code>false</code>.</p> <p>For more information, see <a href="#">Multiple Active Result Sets (MARS)</a>.</p>  |
| <code>MultiSubnetFailover</code>                  | FALSE   | <p>Always specify <code>multiSubnetFailover=True</code> when connecting to the availability group listener of a SQL Server 2012 (or later) availability group or a SQL Server 2012 (or later) Failover Cluster Instance. <code>multiSubnetFailover=True</code> configures SqlClient to provide faster detection of and connection to the (currently) active server. Possible values are <code>Yes</code> and <code>No</code>, <code>True</code> and <code>False</code> or <code>1</code> and <code>0</code>. For example:</p> <pre><code>MultiSubnetFailover=True</code></pre> <p>The default is <code>False</code>. For more information about SqlClient's support for Always On AGs, see <a href="#">SqlClient Support for High Availability, Disaster Recovery</a>.</p>  |
| Network Library<br>-or-<br>Network<br>-or-<br>Net | N/A     | <p>The network library used to establish a connection to an instance of SQL Server. Supported values include:</p> <ul style="list-style-type: none"> <li><code>dbnmpntw</code> (Named Pipes)</li> <li><code>dbmsrpcn</code> (Multiprotocol, Windows RPC)</li> <li><code>dbmsadsn</code> (Apple Talk)</li> <li><code>dbmsgnet</code> (VIA)</li> <li><code>dbmssqlcn</code> (Shared Memory)</li> <li><code>dbmsspxn</code> (IPX/SPX)</li> <li><code>dbmssocn</code> (TCP/IP)</li> <li><code>Dbmsvinn</code> (Banyan Vines)</li> </ul> <p>The corresponding network DLL must be installed on the system to which you connect. If you do not specify a network and you use a local server (for example, <code>"."</code> or <code>"(local)"</code>), shared memory is used. In this example, the network library is Win32 Winsock TCP/IP (<code>dbmssocn</code>), and 1433 is the port being used.</p> <pre><code>Network Library=dbmssocn;Data Source=000.000.000.000,1433;</code></pre> |

| KEYWORD  | DEFAULT | DESCRIPTION  |
|--|---------|--|
| Packet Size  | 8000    | <p>Size in bytes of the network packets used to communicate with an instance of SQL Server.</p> <p>The packet size can be greater than or equal to 512 and less than or equal to 32768.</p>  |
| Password<br>-or-<br>PWD                              | N/A     | <p>The password for the SQL Server account logging on. Not recommended. To maintain a high level of security, we strongly recommend that you use the <a href="#">Integrated Security</a> or <a href="#">Trusted_Connection</a> keyword instead.</p> <p><a href="#">SqlCredential</a> is a more secure way to specify credentials for a connection that uses SQL Server Authentication.</p> <p>The password must be 128 characters or less.</p>         |
| Persist Security Info<br>-or-<br>PersistSecurityInfo | 'false' | <p>When set to <code>false</code> or <code>no</code> (strongly recommended), security-sensitive information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open state. Resetting the connection string resets all connection string values including the password. Recognized values are <code>true</code>, <code>false</code>, <code>yes</code>, and <code>no</code>.</p>          |
| PoolBlockingPeriod                                   | Auto    | Sets the blocking period behavior for a connection pool. See <a href="#">PoolBlockingPeriod</a> property for details.  |
| Pooling  | 'true'  | <p>When the value of this key is set to true, any newly created connection will be added to the pool when closed by the application. In a next attempt to open the same connection, that connection will be drawn from the pool.</p> <p>Connections are considered the same if they have the same connection string. Different connections have different connection strings.</p> <p>The value of this key can be "true", "false", "yes", or "no".</p> |
| Replication  | 'false' | <code>true</code> if replication is supported using the connection.  |

| KEYWORD             | DEFAULT         | DESCRIPTION  |
|---------------------|-----------------|--|
| Transaction Binding | Implicit Unbind | <p>Controls connection association with an enlisted <code>System.Transactions</code> transaction.</p> <p>Possible values are:</p> <pre>Transaction Binding=Implicit Unbind;</pre> <pre>Transaction Binding=Explicit Unbind;</pre> <p>Implicit Unbind causes the connection to detach from the transaction when it ends. After detaching, additional requests on the connection are performed in autocommit mode. The <code>System.Transactions.Transaction.Current</code> property is not checked when executing requests while the transaction is active. After the transaction has ended, additional requests are performed in autocommit mode.</p> <p>If the system ends the transaction (in the scope of a using block) before the last command completes, it will throw <code>InvalidOperationException</code>.</p> <p>Explicit Unbind causes the connection to remain attached to the transaction until the connection is closed or an explicit <code>SqlConnection.TransactionEnlist(null)</code> is called. Beginning in .NET Framework 4, changes to Implicit Unbind make Explicit Unbind obsolete. An <code>InvalidOperationException</code> is thrown if <code>Transaction.Current</code> is not the enlisted transaction or if the enlisted transaction is not active.</p> |

| KEYWORD                        | DEFAULT          | DESCRIPTION  |
|--------------------------------|------------------|--|
| TransparentNetworkIPResolution | See description. | <p>When the value of this key is set to <code>true</code>, the application is required to retrieve all IP addresses for a particular DNS entry and attempt to connect with the first one in the list. If the connection is not established within 0.5 seconds, the application will try to connect to all others in parallel. When the first answers, the application will establish the connection with the respondent IP address.</p> <p>If the <code>MultiSubnetFailover</code> key is set to <code>true</code>, <code>TransparentNetworkIPResolution</code> is ignored.</p> <p>If the <code>Failover Partner</code> key is set, <code>TransparentNetworkIPResolution</code> is ignored.</p> <p>The value of this key must be <code>true</code>, <code>false</code>, <code>yes</code>, or <code>no</code>.</p> <p>A value of <code>yes</code> is treated the same as a value of <code>true</code>.</p> <p>A value of <code>no</code> is treated the same as a value of <code>false</code>.</p> <p>The default values are as follows:</p> <ul style="list-style-type: none"> <li>• <code>false</code> when: <ul style="list-style-type: none"> <li>◦ Connecting to Azure SQL Database where the data source ends with: <ul style="list-style-type: none"> <li>▪ <code>.database.chinacloudapi.cn</code></li> <li>▪ <code>.database.usgovcloudapi.net</code></li> <li>▪ <code>.database.cloudapi.de</code></li> <li>▪ <code>.database.windows.net</code></li> </ul> </li> <li>◦ <code>Authentication</code> is 'Active Directory Password' or 'Active Directory Integrated'</li> </ul> </li> <li>• <code>true</code> in all other cases.</li> </ul> |
| TrustServerCertificate         | 'false'          | <p>When set to <code>true</code>, SSL is used to encrypt the channel when bypassing walking the certificate chain to validate trust. If <code>TrustServerCertificate</code> is set to <code>true</code> and <code>Encrypt</code> is set to <code>false</code>, the channel is not encrypted. Recognized values are <code>true</code>, <code>false</code>, <code>yes</code>, and <code>no</code>. For more information, see <a href="#">Connection String Syntax</a>.</p>   |

| Keyword                        | Default                 | Description  |
|--------------------------------|-------------------------|--|
| Type System Version            | N/A                     | <p>A string value that indicates the type system the application expects. The functionality available to a client application is dependent on the version of SQL Server and the compatibility level of the database. Explicitly setting the type system version that the client application was written for avoids potential problems that could cause an application to break if a different version of SQL Server is used. <b>Note:</b> The type system version cannot be set for common language runtime (CLR) code executing in-process in SQL Server. For more information, see <a href="#">SQL Server Common Language Runtime Integration</a>.</p> <p>Possible values are:</p> <div style="border: 1px solid black; padding: 2px;"><code>Type System Version=SQL Server 2012;</code></div> <div style="border: 1px solid black; padding: 2px;"><code>Type System Version=SQL Server 2008;</code></div> <div style="border: 1px solid black; padding: 2px;"><code>Type System Version=SQL Server 2005;</code></div> <div style="border: 1px solid black; padding: 2px;"><code>Type System Version=Latest;</code></div> <div style="border: 1px solid black; padding: 2px;"><code>Type System Version=SQL Server 2012;</code></div> <p>specifies that the application will require version 11.0.0.0 of Microsoft.SqlServer.Types.dll. The other <code>Type System Version</code> settings will require version 10.0.0.0 of Microsoft.SqlServer.Types.dll.</p> <div style="border: 1px solid black; padding: 2px;"><code>Latest</code></div> is obsolete and should not be used.<br><code>Latest</code> is equivalent to<br><div style="border: 1px solid black; padding: 2px;"><code>Type System Version=SQL Server 2008;</code></div> . |
| User ID<br>-or-<br>UID<br>-or- | N/A                     | <p>The SQL Server login account. Not recommended. To maintain a high level of security, we strongly recommend that you use the <code>Integrated Security</code> or <code>Trusted_Connection</code> keywords instead. <code>SqlCredential</code> is a more secure way to specify credentials for a connection that uses SQL Server Authentication.</p> <p>The user ID must be 128 characters or less.</p>   |
| User Instance                  | 'false'                 | <p>A value that indicates whether to redirect the connection from the default SQL Server Express instance to a runtime-initiated instance running under the account of the caller.</p>   |
| Workstation ID<br>-or-<br>WSID | The local computer name | <p>The name of the workstation connecting to SQL Server.</p> <p>The ID must be 128 characters or less.</p>   |

The following list contains the valid names for connection pooling values within the `ConnectionString`. For more information, see [SQL Server Connection Pooling \(ADO.NET\)](#).

- Connection Lifetime (or Load Balance Timeout)
- Enlist
- Max Pool Size
- Min Pool Size
- Pooling

When you are setting keyword or connection pooling values that require a Boolean value, you can use 'yes' instead of 'true', and 'no' instead of 'false'. Integer values are represented as strings.

**Note**

The .NET Framework Data Provider for SQL Server uses its own protocol to communicate with SQL Server. Therefore, it does not support the use of an ODBC data source name (DSN) when connecting to SQL Server because it does not add an ODBC layer.

**Note**

Universal data link (UDL) files are not supported for the .NET Framework Data Provider for SQL Server.

**Caution**

In this release, the application should use caution when constructing a connection string based on user input (for example when retrieving user ID and password information from a dialog box, and appending it to the connection string). The application should make sure that a user cannot embed additional connection string parameters in these values (for example, entering a password as "validpassword;database=somedb" in an attempt to attach to a different database). If you need to construct connection strings based on user input, use the new [SqlConnectionStringBuilder](#), which validates the connection string and helps to eliminate this problem. See [Connection String Builders](#) for more information.

**See**

Also

[Connection Strings in ADO.NET](#)

[SQL Server Connection Pooling \(ADO.NET\)](#)

[Connecting to a Data Source in ADO.NET](#)

[ADO.NET Managed Providers and DataSet Developer Center](#)

[Connection Strings in ADO.NET](#)

[SQL Server Connection Pooling \(ADO.NET\)](#)

[Connecting to a Data Source in ADO.NET](#)

[ADO.NET Overview](#)

# SqlConnection.ConnectionTimeout SqlConnection.ConnectionTimeout

## In this Article

Gets the time to wait while trying to establish a connection before terminating the attempt and generating an error.

```
[System.Data.DataSysDescription("SqlConnection_ConnectionTimeout")]
public override int ConnectionTimeout { get; }

member this.ConnectionTimeout : int
```

Returns

[Int32](#) [Int32](#)

The time (in seconds) to wait for a connection to open. The default value is 15 seconds.

Attributes

[DataSysDescriptionAttribute](#)

Exceptions

[ArgumentException](#) [ArgumentException](#)

The value set is less than 0.

## Examples

The following example creates a [SqlConnection](#) and sets the [Connection Timeout](#) to 30 seconds in the connection string. The code opens the connection and displays the [ConnectionTimeout](#) property in the console window.

```
private static void OpenSqlConnection()
{
    string connectionString = GetConnectionString();
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("State: {0}", connection.State);
        Console.WriteLine("ConnectionTimeout: {0}",
            connection.ConnectionTimeout);
    }
}

static private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file, using the
    // System.Configuration.ConfigurationSettings.AppSettings property
    return "Data Source=(local);Initial Catalog=AdventureWorks;" +
        "Integrated Security=SSPI;Connection Timeout=30";
}
```

## Remarks

You can set the amount of time a connection waits to time out by using the [Connect Timeout](#) or [Connection Timeout](#) keywords in the connection string. A value of 0 indicates no limit, and should be avoided in a [ConnectionString](#) because an attempt to connect waits indefinitely.

See

[Connection Strings in ADO.NET](#)

Also

[SQL Server Connection Pooling \(ADO.NET\)](#)

[Connecting to a Data Source in ADO.NET](#)



# SqlConnection.CreateCommand SqlConnection.CreateCommand

## In this Article

Creates and returns a [SqlCommand](#) object associated with the [SqlConnection](#).

```
public System.Data.SqlClient.SqlCommand CreateCommand ();  
override this.CreateCommand : unit -> System.Data.SqlClient.SqlCommand
```

Returns

[SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) object.

## Examples

```
using System.Data;  
using System.Data.SqlClient;  
public class A {  
    public static void Main() {  
        using (SqlConnection connection = new SqlConnection("Data Source=(local);Initial  
Catalog=Northwind;Integrated Security=SSPI;")) {  
            connection.Open();  
            SqlCommand command= connection.CreateCommand();  
            command.CommandText = "SELECT * FROM Categories ORDER BY CategoryID";  
            command.CommandTimeout = 15;  
            command.CommandType = CommandType.Text;  
        }  
    }  
}
```

See

[ADO.NET Overview](#)

Also

# SqlConnection.Credential SqlConnection.Credential

## In this Article

Gets or sets the [SqlCredential](#) object for this connection.

```
[System.ComponentModel.Browsable(false)]
public System.Data.SqlClient.SqlCredential Credential { get; set; }

member this.Credential : System.Data.SqlClient.SqlCredential with get, set
```

Returns

[SqlCredential](#) [SqlCredential](#)

The [SqlCredential](#) object for this connection.

Attributes

[BrowsableAttribute](#)

## Remarks

`Persist Security Info = true` is required to get the value of the [SqlCredential](#) object with [Credential](#).

The default value of [Credential](#) is null.

An [InvalidOperationException](#) exception will be raised:

- If [Credential](#) is set on an open connection.
- If [Credential](#) is set when `Context Connection=true`.
- If [Credential](#) is set when `Integrated Security = true`.
- If [Credential](#) is set when the connection string uses `Password`.
- If [Credential](#) is set when the connection string uses `UserID`.

See

[ADO.NET Overview](#)

Also

# SqlConnection.Credentials SqlConnection.Credentials

## In this Article

```
public System.Data.SqlClient.SqlCredential Credentials { get; set; }  
member this.Credentials : System.Data.SqlClient.SqlCredential with get, set
```

## Returns

[SqlCredential](#) [SqlCredential](#)

# SqlConnection.Database SqlConnection.Database

## In this Article

Gets the name of the current database or the database to be used after a connection is opened.

```
[System.Data.DataSysDescription("SqlConnection_Database")]
public override string Database { get; }

member this.Database : string
```

Returns

[String String](#)

The name of the current database or the name of the database to be used after a connection is opened. The default value is an empty string.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlConnection](#) and displays some of its read-only properties.

```
private static void ChangeSqlDatabase(string connectionString)
{
    // Assumes connectionString represents a valid connection string
    // to the AdventureWorks sample database.
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);
        Console.WriteLine("Database: {0}", connection.Database);

        connection.ChangeDatabase("Northwind");
        Console.WriteLine("Database: {0}", connection.Database);
    }
}
```

## Remarks

The [Database](#) property updates dynamically. If you change the current database using a Transact-SQL statement or the [ChangeDatabase](#) method, an informational message is sent and the property is updated automatically.

See

[Connecting to a Data Source in ADO.NET](#)

Also

[Commands and Parameters](#)

[ADO.NET Overview](#)

# SqlConnection.DataSource SqlConnection.DataSource

## In this Article

Gets the name of the instance of SQL Server to which to connect.

```
[System.ComponentModel.Browsable(true)]
[System.Data.DataSysDescription("SqlConnection_DataSource")]
public override string DataSource { get; }

member this.DataSource : string
```

Returns

[String](#)

The name of the instance of SQL Server to which to connect. The default value is an empty string.

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlConnection](#) and displays some of its read-only properties.

```
private static void OpenSqlConnection(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);
        Console.WriteLine("DataSource: {0}", connection.DataSource);
    }
}
```

## Remarks

### [Note](#)

The [DataSource](#) property returns `null` if the connection string for the [SqlConnection](#) is "context connection=true".

See

[Connecting to a Data Source in ADO.NET](#)

Also

[ADO.NET Overview](#)

# SqlConnection.EnlistDistributedTransaction SqlConnection.EnlistDistributedTransaction

## In this Article

Enlists in the specified transaction as a distributed transaction.

```
public void EnlistDistributedTransaction (System.EnterpriseServices.ITransaction transaction);  
member this.EnlistDistributedTransaction : System.EnterpriseServices.ITransaction -> unit
```

### Parameters

transaction [ITransaction](#) [ITransaction](#)

A reference to an existing [ITransaction](#) in which to enlist.

## Remarks

You can use the [EnlistTransaction](#) method to enlist in a distributed transaction. Because it enlists a connection in a [Transaction](#) instance, **EnlistTransaction** takes advantage of functionality available in the [System.Transactions](#) namespace for managing distributed transactions, making it preferable to **EnlistDistributedTransaction** for this purpose. For more information, see [Distributed Transactions](#).

You can continue to enlist in an existing distributed transaction using the **EnlistDistributedTransaction** method if auto-enlistment is disabled. Enlisting in an existing distributed transaction makes sure that, if the transaction is committed or rolled back, modifications made by the code at the data source are also committed or rolled back.

[EnlistDistributedTransaction](#) returns an exception if the [SqlConnection](#) has already started a transaction using [BeginTransaction](#). However, if the transaction is a local transaction started at the data source (for example, by explicitly executing the BEGIN TRANSACTION statement using an [SqlCommand](#) object), **EnlistDistributedTransaction** rolls back the local transaction and enlists in the existing distributed transaction as requested. You do not receive notice that the local transaction was rolled back, and are responsible for managing any local transactions not started using [BeginTransaction](#).

See

[Transactions \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.EnlistTransaction SqlConnection.EnlistTransaction

## In this Article

Enlists in the specified transaction as a distributed transaction.

```
public override void EnlistTransaction (System.Transactions.Transaction transaction);  
override this.EnlistTransaction : System.Transactions.Transaction -> unit
```

### Parameters

transaction [Transaction Transaction](#)

A reference to an existing [Transaction](#) in which to enlist.

## Remarks

You can use the [EnlistTransaction](#) method to enlist in a distributed transaction. Because it enlists a connection in a [Transaction](#) instance, **EnlistTransaction** takes advantage of functionality available in the [System.Transactions](#) namespace for managing distributed transactions, making it preferable to **EnlistDistributedTransaction**, which uses a [System.EnterpriseServices.ITransaction](#) object. It also has slightly different semantics: once a connection is explicitly enlisted on a transaction, it cannot be unenlisted or enlisted in another transaction until the first transaction finishes. For more information about distributed transactions, see [Distributed Transactions](#).

See

[Transactions \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.FireInfoMessageEventOnUserErrors SqlConnection.FireInfoMessageEventOnUserErrors

## In this Article

Gets or sets the [FireInfoMessageEventOnUserErrors](#) property.

```
public bool FireInfoMessageEventOnUserErrors { get; set; }  
member this.FireInfoMessageEventOnUserErrors : bool with get, set
```

Returns

**Boolean** Boolean

`true` if the [FireInfoMessageEventOnUserErrors](#) property has been set; otherwise `false`.

## Remarks

When you set [FireInfoMessageEventOnUserErrors](#) to `true`, errors that were previously treated as exceptions are now handled as [InfoMessage](#) events. All events fire immediately and are handled by the event handler. If [FireInfoMessageEventOnUserErrors](#) is set to `false`, then [InfoMessage](#) events are handled at the end of the procedure.

### **Note**

An error with a severity level of 17 or above that causes the server to stop processing the command needs to be handled as an exception. In this case, an exception is thrown regardless of how the error is handled in the [InfoMessage](#) event.

For more information on working with events, see [Connection Events](#). For more information on errors generated by the SQL Server engine, see [Database Engine Errors](#).

See

[Connection Events](#)

Also

[Connecting to a Data Source \(ADO.NET\)](#)

[ADO.NET Overview](#)

# SqlConnection.GetSchema SqlConnection.GetSchema

In this Article

## Overloads

|  |  |
|--|--|
| <code>GetSchema() GetSchema()</code>                                 | Returns schema information for the data source of this <a href="#">SqlConnection</a> . For more information about scheme, see <a href="#">SQL Server Schema Collections</a> .                  |
| <code>GetSchema(String) GetSchema(String)</code>                     | Returns schema information for the data source of this <a href="#">SqlConnection</a> using the specified string for the schema name.   |
| <code>GetSchema(String, String[]) GetSchema(String, String[])</code> | Returns schema information for the data source of this <a href="#">SqlConnection</a> using the specified string for the schema name and the specified string array for the restriction values. |

## Remarks

If you attempt to retrieve schema information for more than one versioned stored procedure, the schema for the latest one only is returned.

## GetSchema() GetSchema()

Returns schema information for the data source of this [SqlConnection](#). For more information about scheme, see [SQL Server Schema Collections](#).

```
public override System.Data.DataTable GetSchema ();
override this.GetSchema : unit -> System.Data.DataTable
```

Returns

[DataTable](#) [DataTable](#)

A [DataTable](#) that contains schema information.

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

## GetSchema(String) GetSchema(String)

Returns schema information for the data source of this [SqlConnection](#) using the specified string for the schema name.

```
public override System.Data.DataTable GetSchema (string collectionName);
override this.GetSchema : string -> System.Data.DataTable
```

Parameters

collectionName

[String](#) [String](#)

Specifies the name of the schema to return.

Returns

[DataTable](#) [DataTable](#)

A [DataTable](#) that contains schema information.

Exceptions

[ArgumentException](#) [ArgumentException](#)

`collectionName` is specified as null.

Remarks

You may need the schema information of the database, tables or columns. This sample:

- Uses GetSchema to get schema information.
- Use schema restrictions to get the specified information.
- Gets schema information of the database, tables, and some columns.

Before you run the sample, you need to create the sample database, using the following Transact-SQL:

```
USE [master]
GO

CREATE DATABASE [MySchool]

GO

USE [MySchool]
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Course]([CourseID] [nvarchar](10) NOT NULL,
[Year] [smallint] NOT NULL,
[Title] [nvarchar](100) NOT NULL,
[Credits] [int] NOT NULL,
[DepartmentID] [int] NOT NULL,
CONSTRAINT [PK_Course] PRIMARY KEY CLUSTERED
(
[CourseID] ASC,
[Year] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]

GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[Department]([DepartmentID] [int] IDENTITY(1,1) NOT NULL,
[Name] [nvarchar](50) NOT NULL,
[Budget] [money] NOT NULL,
[StartDate] [datetime] NOT NULL,
[Administrator] [int] NULL,
CONSTRAINT [PK_Department] PRIMARY KEY CLUSTERED
(
[DepartmentID] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS = ON,
ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]) ON [PRIMARY]
```

```
GO
```

```
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C1045', 2012, N'Calculus', 4, 7)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C1061', 2012, N'Physics', 4, 1)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C2021', 2012, N'Composition', 3, 2)
INSERT [dbo].[Course] ([CourseID], [Year], [Title], [Credits], [DepartmentID]) VALUES (N'C2042', 2012, N'Literature', 4, 2)
```

```
SET IDENTITY_INSERT [dbo].[Department] ON
```

```
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (1, N'Engineering', 350000.0000, CAST(0x0000999C00000000 AS DateTime), 2)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (2, N'English', 120000.0000, CAST(0x0000999C00000000 AS DateTime), 6)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (4, N'Economics', 200000.0000, CAST(0x0000999C00000000 AS DateTime), 4)
INSERT [dbo].[Department] ([DepartmentID], [Name], [Budget], [StartDate], [Administrator]) VALUES (7, N'Mathematics', 250024.0000, CAST(0x0000999C00000000 AS DateTime), 3)
SET IDENTITY_INSERT [dbo].[Department] OFF
```

```
ALTER TABLE [dbo].[Course] WITH CHECK ADD CONSTRAINT [FK_Course_Department] FOREIGN KEY([DepartmentID])
REFERENCES [dbo].[Department] ([DepartmentID])
GO
ALTER TABLE [dbo].[Course] CHECK CONSTRAINT [FK_Course_Department]
GO
```

[How to Get Schema Information from Database](#) has C# and Visual Basic versions of this code sample in a Visual Studio project.

```
using System;
using System.Data;
using System.Data.SqlClient;

class Program {
    static void Main(string[] args) {

        using (SqlConnection conn = new SqlConnection("Data Source=(local);Initial Catalog=MySchool;Integrated Security=True;Asynchronous Processing=true;")) {
            conn.Open();

            // Get the Meta Data for Supported Schema Collections
            DataTable metaDataTable = conn.GetSchema("MetaDataCollections");

            Console.WriteLine("Meta Data for Supported Schema Collections:");
            ShowDataTable(metaDataTable, 25);
            Console.WriteLine();

            // Get the schema information of Databases in your instance
            DataTable databasesSchemaTable = conn.GetSchema("Databases");

            Console.WriteLine("Schema Information of Databases:");
            ShowDataTable(databasesSchemaTable, 25);
            Console.WriteLine();

            // First, get schema information of all the tables in current database;
            DataTable allTablesSchemaTable = conn.GetSchema("Tables");

            Console.WriteLine("Schema Information of All Tables:");
            ShowDataTable(allTablesSchemaTable, 20);
            Console.WriteLine();
        }
    }
}
```

```

// You can specify the Catalog, Schema, Table Name, Table Type to get
// the specified table(s).
// You can use four restrictions for Table, so you should create a 4 members array.
String[] tableRestrictions = new String[4];

// For the array, 0-member represents Catalog; 1-member represents Schema;
// 2-member represents Table Name; 3-member represents Table Type.
// Now we specify the Table Name of the table what we want to get schema information.
tableRestrictions[2] = "Course";

DataTable courseTableSchemaTable = conn.GetSchema("Tables", tableRestrictions);

Console.WriteLine("Schema Information of Course Tables:");
ShowDataTable(courseTableSchemaTable, 20);
Console.WriteLine();

// First, get schema information of all the columns in current database.
DataTable allColumnsSchemaTable = conn.GetSchema("Columns");

Console.WriteLine("Schema Information of All Columns:");
ShowColumns(allColumnsSchemaTable);
Console.WriteLine();

// You can specify the Catalog, Schema, Table Name, Column Name to get the specified
// column(s).
// You can use four restrictions for Column, so you should create a 4 members array.
String[] columnRestrictions = new String[4];

// For the array, 0-member represents Catalog; 1-member represents Schema;
// 2-member represents Table Name; 3-member represents Column Name.
// Now we specify the Table_Name and Column_Name of the columns what we want to get schema
information.
columnRestrictions[2] = "Course";
columnRestrictions[3] = "DepartmentID";

DataTable departmentIDSchemaTable = conn.GetSchema("Columns", columnRestrictions);

Console.WriteLine("Schema Information of DepartmentID Column in Course Table:");
ShowColumns(departmentIDSchemaTable);
Console.WriteLine();

// First, get schema information of all the IndexColumns in current database
DataTable allIndexColumnsSchemaTable = conn.GetSchema("IndexColumns");

Console.WriteLine("Schema Information of All IndexColumns:");
ShowIndexColumns(allIndexColumnsSchemaTable);
Console.WriteLine();

// You can specify the Catalog, Schema, Table Name, Constraint Name, Column Name to
// get the specified column(s).
// You can use five restrictions for Column, so you should create a 5 members array.
String[] indexColumnsRestrictions = new String[5];

// For the array, 0-member represents Catalog; 1-member represents Schema;
// 2-member represents Table Name; 3-member represents Constraint Name; 4-member represents
Column Name.
// Now we specify the Table_Name and Column_Name of the columns what we want to get schema
information.
indexColumnsRestrictions[2] = "Course";
indexColumnsRestrictions[4] = "CourseID";

DataTable courseIdIndexSchemaTable = conn.GetSchema("IndexColumns",
indexColumnsRestrictions);

```

Console.WriteLine("Schema Information of CourseID Column in Course Table.");

```

        Console.WriteLine("Index Schema information of courseId column in Course table: ");
        ShowIndexColumns(courseIdIndexSchemaTable);
        Console.WriteLine();
    }

    Console.WriteLine("Please press any key to exit...");
    Console.ReadKey();
}

private static void ShowDataTable(DataTable table, Int32 length) {
    foreach ( DataColumn col in table.Columns) {
        Console.Write("{0,-" + length + "}", col.ColumnName);
    }
    Console.WriteLine();

    foreach ( DataRow row in table.Rows) {
        foreach ( DataColumn col in table.Columns) {
            if (col.DataType.Equals(typeof(DateTime)))
                Console.Write("{0,-" + length + ":d}", row[col]);
            else if (col.DataType.Equals(typeof(Decimal)))
                Console.Write("{0,-" + length + ":C}", row[col]);
            else
                Console.Write("{0,-" + length + "}", row[col]);
        }
        Console.WriteLine();
    }
}

private static void ShowDataTable(DataTable table) {
    ShowDataTable(table, 14);
}

private static void ShowColumns(DataTable columnsTable) {
    var selectedRows = from info in columnsTable.AsEnumerable()
        select new {
            TableCatalog = info["TABLE_CATALOG"],
            TableSchema = info["TABLE_SCHEMA"],
            TableName = info["TABLE_NAME"],
            ColumnName = info["COLUMN_NAME"],
            DataType = info["DATA_TYPE"]
        };
    Console.WriteLine("{0,-15}{1,-15}{2,-15}{3,-15}{4,-15}", "TableCatalog", "TABLE_SCHEMA",
        "TABLE_NAME", "COLUMN_NAME", "DATA_TYPE");
    foreach (var row in selectedRows) {
        Console.WriteLine("{0,-15}{1,-15}{2,-15}{3,-15}{4,-15}", row.TableCatalog,
            row.TableSchema, row.TableName, row.ColumnName, row.DataType);
    }
}

private static void ShowIndexColumns(DataTable indexColumnsTable) {
    var selectedRows = from info in indexColumnsTable.AsEnumerable()
        select new {
            TableSchema = info["table_schema"],
            TableName = info["table_name"],
            ColumnName = info["column_name"],
            ConstraintSchema = info["constraint_schema"],
            ConstraintName = info["constraint_name"],
            KeyType = info["KeyType"]
        };
    Console.WriteLine("{0,-14}{1,-11}{2,-14}{3,-18}{4,-16}{5,-8}", "table_schema", "table_name",
        "column_name", "constraint_schema", "constraint_name", "KeyType");
    foreach (var row in selectedRows) {
        Console.WriteLine("{0,-14}{1,-11}{2,-14}{3,-18}{4,-16}{5,-8}", row.TableSchema,
            row.TableName, row.ColumnName, row.ConstraintSchema, row.ConstraintName, row.KeyType);
    }
}

```

```
        }  
    }  
}
```

See

[Obtaining Schema Information from a Database](#)

Also

[ADO.NET Overview](#)

## GetSchema(String, String[]) GetSchema(String, String[])

Returns schema information for the data source of this [SqlConnection](#) using the specified string for the schema name and the specified string array for the restriction values.

```
public override System.Data.DataTable GetSchema (string collectionName, string[] restrictionValues);  
override this.GetSchema : string * string[] -> System.Data.DataTable
```

Parameters

collectionName [String](#)

Specifies the name of the schema to return.

restrictionValues [String\[\]](#)

A set of restriction values for the requested schema.

Returns

[DataTable](#)

A [DataTable](#) that contains schema information.

Exceptions

[ArgumentException](#)

`collectionName` is specified as null.

Remarks

The `restrictionValues` parameter can supply  $n$  depth of values, which are specified by the restrictions collection for a specific collection. In order to set values on a given restriction, and not set the values of other restrictions, you need to set the preceding restrictions to `null` and then put the appropriate value in for the restriction that you would like to specify a value for.

An example of this is the "Tables" collection. If the "Tables" collection has three restrictions--database, owner, and table name--and you want to get back only the tables associated with the owner "Carl", you need to pass in the following values: `null`, "Carl". If a restriction value is not passed in, the default values are used for that restriction. This is the same mapping as passing in `null`, which is different from passing in an empty string for the parameter value. In that case, the empty string ("") is considered to be the value for the specified parameter.

For a code sample demonstrating [GetSchema](#), see [GetSchema](#).

See

[GetSchema\(\)](#)

Also

[Obtaining Schema Information from a Database](#)

[ADO.NET Overview](#)

# SqlConnection.ICloneable.Clone

## In this Article

Creates a new object that is a copy of the current instance.

```
object ICloneable.Clone();
```

Returns

[Object](#)

A new object that is a copy of this instance.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlConnection](#) instance is cast to an [ICloneable](#) interface.

This member is only supported by the .NET Compact Framework.

See

[ADO.NET Overview](#)

Also

# SqlConnection.IDbConnection.BeginTransaction

In this Article

## Overloads

[IDbConnection.BeginTransaction\(\)](#)

[IDbConnection.BeginTransaction\(IsolationLevel\)](#)

### IDbConnection.BeginTransaction()

```
System.Data.IDbTransaction IDbConnection.BeginTransaction ();
```

Returns

[IDbTransaction](#)

### IDbConnection.BeginTransaction(IsolationLevel)

```
System.Data.IDbTransaction IDbConnection.BeginTransaction (System.Data.IsolationLevel iso);
```

Parameters

iso

[IsolationLevel](#)

Returns

[IDbTransaction](#)

# SqlConnection.IDbConnection.CreateCommand

## In this Article

```
System.Data.IDbCommand IDbConnection.CreateCommand ();
```

## Returns

[IDbCommand](#)

# SqlConnection.InfoMessage SqlConnection.InfoMessage

## In this Article

Occurs when SQL Server returns a warning or informational message.

```
[System.Data.DataSysDescription("DbConnection_InfoMessage")]
public event System.Data.SqlClient.SqlInfoMessageEventHandler InfoMessage;
member this.InfoMessage : System.Data.SqlClient.SqlInfoMessageEventHandler
```

Attributes

[DataSysDescriptionAttribute](#)

## Remarks

Clients that want to process warnings or informational messages sent by the server should create an [SqlInfoMessageEventHandler](#) delegate to listen to this event.

The [InfoMessage](#) event occurs when a message with a severity of 10 or less is returned by SQL Server. Messages that have a severity between 11 and 20 raise an error and messages that have a severity over 20 causes the connection to close. For more information on SQL Server error levels, see [Database Engine Error Severities](#).

For more information and an example, see [Connection Events](#).

See

[Working with Connection Events](#)

Also

[ADO.NET Overview](#)

# SqlConnection.Open SqlConnection.Open

## In this Article

Opens a database connection with the property settings specified by the [ConnectionString](#).

```
public override void Open ();  
override this.Open : unit -> unit
```

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Cannot open a connection without specifying a data source or server.

or

The connection is already open.

[SqlException](#) [SqlException](#)

A connection-level error occurred while opening the connection. If the [Number](#) property contains the value 18487 or 18488, this indicates that the specified password has expired or must be reset. See the [ChangePassword\(String, String\)](#) method for more information.

The `<system.data.localdb>` tag in the app.config file has invalid or unknown elements.

[ConfigurationErrorsException](#) [ConfigurationErrorsException](#)

There are two entries with the same name in the `<localdbinstances>` section.

## Examples

The following example creates a [SqlConnection](#), opens it, and displays some of its properties. The connection is automatically closed at the end of the `using` block.

```
private static void OpenSqlConnection(string connectionString)  
{  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        connection.Open();  
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);  
        Console.WriteLine("State: {0}", connection.State);  
    }  
}
```

## Remarks

The [SqlConnection](#) draws an open connection from the connection pool if one is available. Otherwise, it establishes a new connection to an instance of SQL Server.

### Note

If the [SqlConnection](#) goes out of scope, it is not closed. Therefore, you must explicitly close the connection by calling [Close](#).

**Note**

If you specify a port number other than 1433 when you are trying to connect to an instance of SQL Server and using a protocol other than TCP/IP, the [Open](#) method fails. To specify a port number other than 1433, include "server=machinename,port number" in the connection string, and use the TCP/IP protocol.

**Note**

The .NET Framework Data Provider for SQL Server requires the Security permission with "Allows calls to unmanaged assemblies" enabled ([SecurityPermission](#) with [SecurityPermissionFlag](#) set to [UnmanagedCode](#)) to open a [SqlConnection](#) with SQL Debugging enabled.

See

[Connection Strings in ADO.NET](#)

Also

[SQL Server Connection Pooling \(ADO.NET\)](#)

[Connecting to a Data Source in ADO.NET](#)

[ADO.NET Overview](#)

# SqlConnection.OpenAsync SqlConnection.OpenAsync

## In this Article

An asynchronous version of [Open\(\)](#), which opens a database connection with the property settings specified by the [ConnectionString](#). The cancellation token can be used to request that the operation be abandoned before the connection timeout elapses. Exceptions will be propagated via the returned Task. If the connection timeout time elapses without successfully connecting, the returned Task will be marked as faulted with an Exception. The implementation returns a Task without blocking the calling thread for both pooled and non-pooled connections.

```
public override System.Threading.Tasks.Task OpenAsync (System.Threading.CancellationToken cancellationToken);  
  
override this.OpenAsync : System.Threading.CancellationToken -> System.Threading.Tasks.Task
```

### Parameters

cancellationToken [CancellationToken CancellationToken](#)

The cancellation instruction.

### Returns

[Task Task](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

Calling [OpenAsync\(CancellationToken\)](#) more than once for the same instance before task completion.

`Context Connection=true` is specified in the connection string.

A connection was not available from the connection pool before the connection time out elapsed.

[SqlException SqlException](#)

Any error returned by SQL Server that occurred while opening the connection.

## Remarks

After calling [OpenAsync](#), [State](#) must return [Connecting](#) until the returned [Task](#) is completed. Then, if the connection was successful, [State](#) must return [Open](#). If the connection fails, [State](#) must return [Closed](#).

A call to [Close](#) will attempt to cancel or close the corresponding [OpenAsync](#) call.

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

# SqlConnection.PacketSize SqlConnection.PacketSize

## In this Article

Gets the size (in bytes) of network packets used to communicate with an instance of SQL Server.

```
[System.Data.DataSysDescription("SqlConnection_PacketSize")]
public int PacketSize { get; }

member this.PacketSize : int
```

## Returns

[Int32](#) [Int32](#)

The size (in bytes) of network packets. The default value is 8000.

## Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlConnection](#), including setting the [Packet Size](#) to 512 in the connection string. It displays the [PacketSize](#) and [ServerVersion](#) properties in the console window.

```
private static void OpenSqlConnection()
{
    string connectionString = GetConnectionString();
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);
        Console.WriteLine("PacketSize: {0}", connection.PacketSize);
    }
}

static private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file, using the
    // System.Configuration.ConfigurationSettings.AppSettings property
    return "Data Source=(local);Initial Catalog=AdventureWorks;" +
        "Integrated Security=SSPI;Packet Size=512";
}
```

## Remarks

If an application performs bulk copy operations, or sends or receives lots of text or image data, a packet size larger than the default may improve efficiency because it causes fewer network read and write operations. If an application sends and receives small amounts of information, you can set the packet size to 512 bytes (using the [Packet Size](#) value in the [ConnectionString](#)), which is sufficient for most data transfer operations. For most applications, the default packet size is best.

[PacketSize](#) may be a value in the range of 512 and 32767 bytes. An exception is generated if the value is outside this range.

Setting the default value to a number greater than 8000 will cause the packets to use the MultiPage allocator on the instance of SQL Server instead of the much more efficient SinglePage allocator, reducing the overall scalability of the SQL Server. For more information on how SQL Server uses memory, see [Memory Management Architecture Guide](#).

## See

[Connecting to a Data Source in ADO.NET](#)

Also

[ADO.NET Overview](#)

# SqlConnection.RegisterColumnEncryptionKeyStore Providers SqlConnection.RegisterColumnEncryptionKey StoreProviders

## In this Article

Registers the column encryption key store providers.

```
public static void RegisterColumnEncryptionKeyStoreProviders
(System.Collections.Generic.IDictionary<string, System.Data.SqlClient.SqlColumnEncryptionKeyStoreProvider> customProviders);

static member RegisterColumnEncryptionKeyStoreProviders :
System.Collections.Generic.IDictionary<string,
System.Data.SqlClient.SqlColumnEncryptionKeyStoreProvider> -> unit
```

## Parameters

customProviders [IDictionary<String,SqlColumnEncryptionKeyStoreProvider>](#)

The custom providers

# SqlConnection.ResetStatistics SqlConnection.ResetStatistics

## In this Article

If statistics gathering is enabled, all values are reset to zero.

```
public void ResetStatistics ();  
member this.ResetStatistics : unit -> unit
```

## Remarks

If statistics gathering is not enabled and this method is called, no error is thrown.

See

[Provider Statistics for SQL Server \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.RetrieveStatistics SqlConnection.RetrieveStatistics

## In this Article

Returns a name value pair collection of statistics at the point in time the method is called.

```
public System.Collections.IDictionary RetrieveStatistics ();  
member this.RetrieveStatistics : unit -> System.Collections.IDictionary
```

Returns

[IDictionary](#) [IDictionary](#)

Returns a reference of type [IDictionary](#) of [DictionaryEntry](#) items.

## Remarks

When this method is called, the values retrieved are those at the current point in time. If you continue using the connection, the values are incorrect. You need to re-execute the method to obtain the most current values.

See

[Provider Statistics for SQL Server \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.ServerVersion SqlConnection.ServerVersion

## In this Article

Gets a string that contains the version of the instance of SQL Server to which the client is connected.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("SqlConnection_ServerVersion")]
public override string ServerVersion { get; }

member this.ServerVersion : string
```

Returns

[String](#)

The version of the instance of SQL Server.

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The connection is closed.

[ServerVersion](#) was called while the returned Task was not completed and the connection was not opened after a call to [OpenAsync\(CancellationToken\)](#).

## Examples

The following example creates a [SqlConnection](#) and displays the [ServerVersion](#) property.

```
private static void CreateSqlConnection(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);
        Console.WriteLine("State: {0}", connection.State );
    }
}
```

## Remarks

The version is of the form `##.##.####`, where the first two digits are the major version, the next two digits are the minor version, and the last four digits are the release version. The string is of the form *major.minor.build*, where major and minor are exactly two digits and build is exactly four digits.

[ServerVersion](#) was called while the returned Task was not completed and the connection was not opened after a call to [OpenAsync](#).

See

[SQL Server and ADO.NET](#)

Also

[ADO.NET Overview](#)

# SqlConnection SqlConnection

In this Article

## Overloads

|  |   |
|--|---|
| <code>SqlConnection()</code>   | Initializes a new instance of the <a href="#">SqlConnection</a> class.  |
| <code>SqlConnection(String) SqlConnection(String)</code>                               | Initializes a new instance of the <a href="#">SqlConnection</a> class when given a string that contains the connection string.  |
| <code>SqlConnection(String, SqlCredential) SqlConnection(String, SqlCredential)</code> | Initializes a new instance of the <a href="#">SqlConnection</a> class given a connection string, that does not use <code>Integrated Security = true</code> and a <a href="#">SqlCredential</a> object that contains the user ID and password. |

## SqlConnection()

Initializes a new instance of the [SqlConnection](#) class.

```
public SqlConnection ();
```

### Examples

The following example creates and opens a [SqlConnection](#).

```
private static void OpenSqlConnection()
{
    string connectionString = GetConnectionString();
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);
        Console.WriteLine("State: {0}", connection.State);
    }
}

static private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file, using the
    // System.Configuration.ConfigurationManager.ConnectionStrings property
    return "Data Source=(local);Initial Catalog=AdventureWorks;" +
        "Integrated Security=SSPI;";
}
```

### Remarks

When a new instance of [SqlConnection](#) is created, the read/write properties are set to the following initial values unless they are specifically set using their associated keywords in the [ConnectionString](#) property.

| PROPERTIES                    | INITIAL VALUE     |
|-------------------------------|-------------------|
| <code>ConnectionString</code> | empty string ("") |

| Properties        | Initial Value     |
|-------------------|-------------------|
| ConnectionTimeout | 15                |
| Database          | empty string ("") |
| DataSource        | empty string ("") |

You can change the value for these properties only by using the [ConnectionString](#) property. The [SqlConnectionStringBuilder](#) class provides functionality for creating and managing the contents of connection strings.

See

[Connecting to a Data Source in ADO.NET](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

## SqlConnection(String) SqlConnection(String)

Initializes a new instance of the [SqlConnection](#) class when given a string that contains the connection string.

```
public SqlConnection (string connectionString);
new System.Data.SqlClient.SqlConnection : string -> System.Data.SqlClient.SqlConnection
```

Parameters

|                  |                        |
|------------------|------------------------|
| connectionString | <a href="#">String</a> |
|------------------|------------------------|

The connection used to open the SQL Server database.

Examples

The following example creates and opens a [SqlConnection](#).

```
private static void OpenSqlConnection()
{
    string connectionString = GetConnectionString();

    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        Console.WriteLine("State: {0}", connection.State);
        Console.WriteLine("ConnectionString: {0}",
            connection.ConnectionString);
    }
}

static private string GetConnectionString()
{
    // To avoid storing the connection string in your code,
    // you can retrieve it from a configuration file, using the
    // System.Configuration.ConfigurationSettings.AppSettings property
    return "Data Source=(local);Initial Catalog=AdventureWorks";
    + "Integrated Security=SSPI;";
}
```

Remarks

When a new instance of [SqlConnection](#) is created, the read/write properties are set to the following initial values unless they are specifically set using their associated keywords in the [ConnectionString](#) property.

| PROPERTIES        | INITIAL VALUE     |
|-------------------|-------------------|
| ConnectionString  | connectionString  |
| ConnectionTimeout | 15                |
| Database          | empty string ("") |
| DataSource        | empty string ("") |

You can change the value for these properties only by using the [ConnectionString](#) property. The [SqlConnection](#) class provides functionality for creating and managing the contents of connection strings.

See

[Connecting to a Data Source \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## SqlConnection(String, SqlCredential) SqlConnection(String, SqlCredential)

Initializes a new instance of the [SqlConnection](#) class given a connection string, that does not use [Integrated Security = true](#) and a [SqlCredential](#) object that contains the user ID and password.

```
public SqlConnection (string connectionString, System.Data.SqlClient.SqlCredential cred);
new System.Data.SqlClient.SqlConnection : string * System.Data.SqlClient.SqlCredential ->
System.Data.SqlClient.SqlConnection
```

Parameters

connectionString [String](#)

A connection string that does not use any of the following connection string keywords: [Integrated Security = true](#), [UserId](#), or [Password](#); or that does not use [ContextConnection = true](#).

credential [SqlCredential](#)

A [SqlCredential](#) object. If [credential](#) is null, [SqlConnection\(String, SqlCredential\)](#) is functionally equivalent to [SqlConnection\(String\)](#).

See

[ADO.NET Overview](#)

Also

# SqlConnection.State SqlConnection.State

## In this Article

Indicates the state of the [SqlConnection](#) during the most recent network operation performed on the connection.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("DbConnection_State")]
public override System.Data.ConnectionState State { get; }

member this.State : System.Data.ConnectionState
```

Returns

[ConnectionState](#) [ConnectionString](#)

An [ConnectionString](#) enumeration.

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Remarks

Returns an [ConnectionString](#) enumeration indicating the state of the [SqlConnection](#). Closing and reopening the connection will refresh the value of [State](#).

See

[Connecting to a Data Source \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.StateChange SqlConnection.StateChange

## In this Article

```
[System.Data.DataSysDescription("DbConnection_StateChange")]
public event System.Data.StateChangeEventHandler StateChange;

member this.StateChange : System.Data.StateChangeEventHandler
```

Attributes

[DataSysDescriptionAttribute](#)

# SqlConnection.StatisticsEnabled SqlConnection.StatisticsEnabled

## In this Article

When set to `true`, enables statistics gathering for the current connection.

```
public bool StatisticsEnabled { get; set; }  
member this.StatisticsEnabled : bool with get, set
```

Returns

**Boolean Boolean**

Returns `true` if statistics gathering is enabled; otherwise `false`. `false` is the default.

## Remarks

Enabling statistics gathering has a minor, but measurable effect on performance and therefore should be enabled only when it is required.

See

[Provider Statistics for SQL Server \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnection.WorkstationId SqlConnection.WorkstationId

## In this Article

Gets a string that identifies the database client.

```
[System.Data.DataSysDescription("SqlConnection_WorkstationId")]
public string WorkstationId { get; }

member this.WorkstationId : string
```

Returns

[String](#) [String](#)

A string that identifies the database client. If not specified, the name of the client computer. If neither is specified, the value is an empty string.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlConnection](#) and displays the [WorkstationId](#) property.

```
private static void OpenSqlConnection(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        Console.WriteLine("ServerVersion: {0}", connection.ServerVersion);
        Console.WriteLine("WorkstationId: {0}", connection.WorkstationId);
    }
}
```

## Remarks

The string typically contains the network name of the client. The [WorkstationId](#) property corresponds to the [Workstation ID](#) connection string property.

See

[SQL Server and ADO.NET](#)

Also

[ADO.NET Overview](#)

# SqlConnectionColumnEncryptionSetting SqlConnection ColumnEncryptionSetting Enum

Specifies that Always Encrypted functionality is enabled in a connection. Note that these settings cannot be used to bypass encryption and gain access to plaintext data. For details, see [Always Encrypted \(Database Engine\)](#).

## Declaration

```
public enum SqlConnectionColumnEncryptionSetting  
type SqlConnectionColumnEncryptionSetting =
```

## Inheritance Hierarchy



## Fields

`Disabled`  
`Disabled`

Specifies the connection does not use Always Encrypted. Should be used if no queries sent over the connection access encrypted columns.

`Enabled`  
`Enabled`

Enables Always Encrypted functionality for the connection. Query parameters that correspond to encrypted columns will be transparently encrypted and query results from encrypted columns will be transparently decrypted.

## See Also

# SqlConnectionStringBuilder Class

Provides a simple way to create and manage the contents of connection strings used by the [SqlConnection](#) class.

## Declaration

```
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlConnectionStringBuilder/SqlConnectionStringBuilderConverter))]
public sealed class SqlConnectionStringBuilder : System.Data.Common.DbConnectionStringBuilder

type SqlConnectionStringBuilder = class
    inherit DbConnectionStringBuilder
```

## Inheritance Hierarchy



## Remarks

The connection string builder lets developers programmatically create syntactically correct connection strings, and parse and rebuild existing connection strings, using properties and methods of the class. The connection string builder provides strongly typed properties corresponding to the known key/value pairs allowed by SQL Server. Developers needing to create connection strings as part of applications can use the [SqlConnectionStringBuilder](#) class to build and modify connection strings. The class also makes it easy to manage connection strings stored in an application configuration file.

The [SqlConnectionStringBuilder](#) performs checks for valid key/value pairs. Therefore, you cannot use this class to create invalid connection strings; trying to add invalid pairs will throw an exception. The class maintains a fixed collection of synonyms and can translate from a synonym to the corresponding well-known key name.

For example, when you use the **Item** property to retrieve a value, you can specify a string that contains any synonym for the key you need. For example, you can specify "Network Address", "addr", or any other acceptable synonym for this key within a connection string when you use any member that requires a string that contains the key name, such as the **Item** property or the [Remove](#) method. See the [ConnectionString](#) property for a full list of acceptable synonyms.

The **Item** property handles tries to insert malicious entries. For example, the following code, using the default **Item** property (the indexer, in C#) correctly escapes the nested key/value pair:

```
System.Data.SqlClient.SqlConnectionStringBuilder builder =
    new System.Data.SqlClient.SqlConnectionStringBuilder();
builder["Data Source"] = "(local)";
builder["integrated Security"] = true;
builder["Initial Catalog"] = "AdventureWorks;NewValue=Bad";
Console.WriteLine(builder.ConnectionString);
```

The result is the following connection string that handles the invalid value in a safe manner:

```
Source=(local);Initial Catalog="AdventureWorks;NewValue=Bad";
Integrated Security=True
```

## Constructors

```
SqlConnectionStringBuilder()
SqlConnectionStringBuilder()
```

Initializes a new instance of the [SqlConnectionStringBuilder](#) class.

```
SqlConnectionStringBuilder(String)  
SqlConnectionStringBuilder(String)
```

Initializes a new instance of the [SqlConnectionStringBuilder](#) class. The provided connection string provides the data for the instance's internal connection information.

## Properties

[ApplicationIntent](#)

```
ApplicationIntent
```

Declares the application workload type when connecting to a database in an SQL Server Availability Group. You can set the value of this property with [ApplicationIntent](#). For more information about [SqlClient](#) support for Always On Availability Groups, see [SqlClient Support for High Availability, Disaster Recovery](#).

[ApplicationName](#)

```
ApplicationName
```

Gets or sets the name of the application associated with the connection string.

[AsynchronousProcessing](#)

```
AsynchronousProcessing
```

Gets or sets a Boolean value that indicates whether asynchronous processing is allowed by the connection created by using this connection string.

[AttachDBFilename](#)

```
AttachDBFilename
```

Gets or sets a string that contains the name of the primary data file. This includes the full path name of an attachable database.

[Authentication](#)

```
Authentication
```

Gets the authentication of the connection string.

[ColumnEncryptionSetting](#)

```
ColumnEncryptionSetting
```

Gets and sets the column encryption settings for the connection string builder.

**ConnectionReset**

**ConnectionReset**

Obsolete. Gets or sets a Boolean value that indicates whether the connection is reset when drawn from the connection pool.

**ConnectRetryCount**

**ConnectRetryCount**

The number of reconnections attempted after identifying that there was an idle connection failure. This must be an integer between 0 and 255. Default is 1. Set to 0 to disable reconnecting on idle connection failures. An [ArgumentException](#) will be thrown if set to a value outside of the allowed range.

**ConnectRetryInterval**

**ConnectRetryInterval**

Amount of time (in seconds) between each reconnection attempt after identifying that there was an idle connection failure. This must be an integer between 1 and 60. The default is 10 seconds. An [ArgumentException](#) will be thrown if set to a value outside of the allowed range.

**ConnectTimeout**

**ConnectTimeout**

Gets or sets the length of time (in seconds) to wait for a connection to the server before terminating the attempt and generating an error.

**ContextConnection**

**ContextConnection**

Gets or sets a value that indicates whether a client/server or in-process connection to SQL Server should be made.

**CurrentLanguage**

**CurrentLanguage**

Gets or sets the SQL Server Language record name.

**DataSource**

**DataSource**

Gets or sets the name or network address of the instance of SQL Server to connect to.

**EnclaveAttestationUrl**

**EnclaveAttestationUrl**

Gets or sets the enclave attestation Url to be used with enclave based Always Encrypted.

## Encrypt

### Encrypt

Gets or sets a Boolean value that indicates whether SQL Server uses SSL encryption for all data sent between the client and server if the server has a certificate installed.

## Enlist

### Enlist

Gets or sets a Boolean value that indicates whether the SQL Server connection pooler automatically enlists the connection in the creation thread's current transaction context.

## FailoverPartner

### FailoverPartner

Gets or sets the name or address of the partner server to connect to if the primary server is down.

## InitialCatalog

### InitialCatalog

Gets or sets the name of the database associated with the connection.

## IntegratedSecurity

### IntegratedSecurity

Gets or sets a Boolean value that indicates whether User ID and Password are specified in the connection (when `false`) or whether the current Windows account credentials are used for authentication (when `true`).

## IsFixedSize

### IsFixedSize

Gets a value that indicates whether the [SqlConnectionStringBuilder](#) has a fixed size.

## Item[String]

### Item[String]

Gets or sets the value associated with the specified key. In C#, this property is the indexer.

## Keys

### Keys

Gets an [ICollection](#) that contains the keys in the [SqlConnectionStringBuilder](#).

## LoadBalanceTimeout

### LoadBalanceTimeout

Gets or sets the minimum time, in seconds, for the connection to live in the connection pool before being destroyed.

#### MaxPoolSize

##### MaxPoolSize

Gets or sets the maximum number of connections allowed in the connection pool for this specific connection string.

#### MinPoolSize

##### MinPoolSize

Gets or sets the minimum number of connections allowed in the connection pool for this specific connection string.

#### MultipleActiveResultSets

##### MultipleActiveResultSets

When true, an application can maintain multiple active result sets (MARS). When false, an application must process or cancel all result sets from one batch before it can execute any other batch on that connection.

For more information, see [Multiple Active Result Sets \(MARS\)](#).

#### MultiSubnetFailover

##### MultiSubnetFailover

If your application is connecting to an AlwaysOn availability group (AG) on different subnets, setting MultiSubnetFailover=true provides faster detection of and connection to the (currently) active server. For more information about SqlClient support for Always On Availability Groups, see [SqlClient Support for High Availability, Disaster Recovery](#).

#### NetworkLibrary

##### NetworkLibrary

Gets or sets a string that contains the name of the network library used to establish a connection to the SQL Server.

#### PacketSize

##### PacketSize

Gets or sets the size in bytes of the network packets used to communicate with an instance of SQL Server.

#### Password

##### Password

Gets or sets the password for the SQL Server account.

**PersistSecurityInfo****PersistSecurityInfo**

Gets or sets a Boolean value that indicates if security-sensitive information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open state.

**PoolBlockingPeriod****PoolBlockingPeriod**

The blocking period behavior for a connection pool.

**Pooling****Pooling**

Gets or sets a Boolean value that indicates whether the connection will be pooled or explicitly opened every time that the connection is requested.

**Replication****Replication**

Gets or sets a Boolean value that indicates whether replication is supported using the connection.

**TransactionBinding****TransactionBinding**

Gets or sets a string value that indicates how the connection maintains its association with an enlisted `System.Transactions` transaction.

**TransparentNetworkIPResolution****TransparentNetworkIPResolution**

When the value of this key is set to `true`, the application is required to retrieve all IP addresses for a particular DNS entry and attempt to connect with the first one in the list. If the connection is not established within 0.5 seconds, the application will try to connect to all others in parallel. When the first answers, the application will establish the connection with the respondent IP address.

**TrustServerCertificate****TrustServerCertificate**

Gets or sets a value that indicates whether the channel will be encrypted while bypassing walking the certificate chain to validate trust.

**TypeSystemVersion****TypeSystemVersion**

Gets or sets a string value that indicates the type system the application expects.

**UserID**

**UserID**

Gets or sets the user ID to be used when connecting to SQL Server.

**UserInstance**

**UserInstance**

Gets or sets a value that indicates whether to redirect the connection from the default SQL Server Express instance to a runtime-initiated instance running under the account of the caller.

**Values**

**Values**

Gets an [ICollection](#) that contains the values in the [SqlConnectionStringBuilder](#).

**WorkstationID**

**WorkstationID**

Gets or sets the name of the workstation connecting to SQL Server.

## Methods

**Clear()**

**Clear()**

Clears the contents of the [SqlConnectionStringBuilder](#) instance.

**ContainsKey(String)**

**ContainsKey(String)**

Determines whether the [SqlConnectionStringBuilder](#) contains a specific key.

**Remove(String)**

**Remove(String)**

Removes the entry with the specified key from the [SqlConnectionStringBuilder](#) instance.

**ShouldSerialize(String)**

**ShouldSerialize(String)**

Indicates whether the specified key exists in this [SqlConnectionStringBuilder](#) instance.

**TryGetValue(String, Object)**

**TryGetValue(String, Object)**

Retrieves a value corresponding to the supplied key from this [SqlConnectionStringBuilder](#).

## See Also

# SqlConnectionStringBuilder.ApplicationIntent Sql ConnectionStringBuilder.ApplicationIntent

## In this Article

Declares the application workload type when connecting to a database in an SQL Server Availability Group. You can set the value of this property with [ApplicationIntent](#). For more information about `SqlClient` support for Always On Availability Groups, see [SqlClient Support for High Availability, Disaster Recovery](#).

```
public System.Data.SqlClient.ApplicationIntent ApplicationIntent { get; set; }  
member this.ApplicationIntent : System.Data.SqlClient.ApplicationIntent with get, set
```

## Returns

[ApplicationIntent ApplicationIntent](#)

Returns the current value of the property (a value of type [ApplicationIntent](#)).

## See

[ADO.NET Overview](#)

## Also

# SqlConnectionStringBuilder.ApplicationName SqlConnectionStringBuilder.ApplicationName

## In this Article

Gets or sets the name of the application associated with the connection string.

```
public string ApplicationName { get; set; }  
member this.ApplicationName : string with get, set
```

Returns

[String String](#)

The name of the application, or ".NET SqlClient Data Provider" if no name has been supplied.

Exceptions

[ArgumentNullException ArgumentNullException](#)

To set the value to null, use [Value](#).

## Examples

The following example creates a new [SqlConnectionStringBuilder](#) and assigns a connection string in the object's constructor. The code displays the parsed and recreated version of the connection string, and then modifies the [ApplicationName](#) property of the object. Finally, the code displays the new connection string, including the new key/value pair.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        try  
        {  
            string connectString = "Server=(local);Initial Catalog=AdventureWorks;" +  
                "Integrated Security=true";  
            SqlConnectionStringBuilder builder =  
                new SqlConnectionStringBuilder(connectString);  
            Console.WriteLine("Original: " + builder.ConnectionString);  
            Console.WriteLine("ApplicationName={0}",  
                builder.ApplicationName);  
  
            builder.ApplicationName = "My Application";  
            Console.WriteLine("Modified: " + builder.ConnectionString);  
  
            Console.WriteLine("Press any key to finish.");  
            Console.ReadLine();  
  
        }  
        catch (Exception ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}
```

The sample displays the following text in the console window:

```
Original: Data Source=(local);Initial Catalog=AdventureWorks;Integrated Security=True  
ApplicationName=".Net SqlClient Data Provider"  
Modified: Data Source=(local);Initial Catalog=AdventureWorks;Integrated Security=True;Application  
Name="My Application"
```

## Remarks

This property corresponds to the "Application Name" and "app" keys within the connection string.

See

[Connection Strings \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.AynchronousProcessing SqlConnectionStringBuilder.AynchronousProcessing

## In this Article

Gets or sets a Boolean value that indicates whether asynchronous processing is allowed by the connection created by using this connection string.

```
public bool AsynchronousProcessing { get; set; }  
member this.AynchronousProcessing : bool with get, set
```

## Returns

[Boolean](#) [Boolean](#)

The value of the [AsynchronousProcessing](#) property, or `false` if no value has been supplied.

## Examples

The following example retrieves a connection string and verifies that the connection string is configured to allow for asynchronous processing. (In this case, the string comes from a procedure within the application, but in a production application, the connection string might come from a configuration file, or some other source.) Then, the example performs an asynchronous operation, updating values within a sample database on a background thread.

```
using System.Data.SqlClient;  
using System.Threading;  
  
class Program  
{  
    static void Main()  
    {  
        // Create a SqlConnectionStringBuilder instance,  
        // and ensure that it is set up for asynchronous processing.  
        SqlConnectionStringBuilder builder =  
            new SqlConnectionStringBuilder(GetConnectionString());  
        // Asynchronous method calls won't work unless you  
        // have added this option, or have added  
        // the clause "Asynchronous Processing=true"  
        // to the connection string.  
        builder.AynchronousProcessing = true;  
  
        string commandText =  
            "UPDATE Production.Product SET ReorderPoint = ReorderPoint + 1 " +  
            "WHERE ReorderPoint IS NOT Null;" +  
            "WAITFOR DELAY '0:0:3';" +  
            "UPDATE Production.Product SET ReorderPoint = ReorderPoint - 1 " +  
            "WHERE ReorderPoint IS NOT Null";  
        RunCommandAsynchronously(commandText, builder.ConnectionString);  
  
        Console.WriteLine("Press any key to finish.");  
        Console.ReadLine();  
    }  
  
    private static string GetConnectionString()  
    {  
        // To avoid storing the connection string in your code,  
        // you can retrieve it from a configuration file.  
        return "Data Source=(local);Integrated Security=SSPI;" +  
            "Initial Catalog=AdventureWorks";  
    }  
}
```

```

private static void RunCommandAsynchronously(string commandText,
    string connectionString)
{
    // Given command text and connection string, asynchronously execute
    // the specified command against the connection. For this example,
    // the code displays an indicator as it's working, verifying the
    // asynchronous behavior.
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        try
        {
            int count = 0;
            SqlCommand command = new SqlCommand(commandText, connection);
            connection.Open();
            IAsyncResult result = command.BeginExecuteNonQuery();
            while (!result.IsCompleted)
            {
                Console.WriteLine("Waiting {0}."., count);
                // Wait for 1/10 second, so the counter
                // doesn't consume all available resources
                // on the main thread.
                Thread.Sleep(100);
                count += 1;
            }
            Console.WriteLine("Command complete. Affected {0} rows.",
                command.EndExecuteNonQuery(result));
        }
        catch (SqlException ex)
        {
            Console.WriteLine(
                "Error {0}: System.Data.SqlClient.SqlConnectionStringBuilder",
                ex.Number, ex.Message);
        }
        catch (InvalidOperationException ex)
        {
            Console.WriteLine("Error: {0}", ex.Message);
        }
        catch (Exception ex)
        {
            // You might want to pass these errors
            // back out to the caller.
            Console.WriteLine("Error: {0}", ex.Message);
        }
    }
}
}

```

## Remarks

This property corresponds to the "Asynchronous Processing" and "async" keys within the connection string. In order to take advantage of the asynchronous processing provided by the [SqlCommand](#) object, this key/value pair must be included within the connection string of the associated [SqlConnection](#) object.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.AttachDBFilename Sql ConnectionStringBuilder.AttachDBFilename

## In this Article

Gets or sets a string that contains the name of the primary data file. This includes the full path name of an attachable database.

```
public string AttachDBFilename { get; set; }  
member this.AttachDBFilename : string with get, set
```

## Returns

[String](#) [String](#)

The value of the `AttachDBFilename` property, or `String.Empty` if no value has been supplied.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

To set the value to null, use [Value](#).

## Examples

The following example creates a new [SqlConnectionStringBuilder](#) instance, and sets the `AttachDBFilename` property in order to specify the name of an attached data file.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        try
        {
            string connectString =
                "Server=(local);" +
                "Integrated Security=true";
            SqlConnectionStringBuilder builder =
                new SqlConnectionStringBuilder(connectString);
            Console.WriteLine("Original: " + builder.ConnectionString);
            Console.WriteLine("AttachDBFileName={0}", builder.AttachDBFilename);

            builder.AttachDBFilename = @"C:\MyDatabase.mdf";
            Console.WriteLine("Modified: " + builder.ConnectionString);

            using (SqlConnection connection = new SqlConnection(builder.ConnectionString))
            {
                connection.Open();
                // Now use the open connection.
                Console.WriteLine("Database = " + connection.Database);
            }
            Console.WriteLine("Press any key to finish.");
            Console.ReadLine();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}

```

## Remarks

This property corresponds to the "AttachDBFilename", "extended properties", and "initial file name" keys within the connection string.

`AttachDBFilename` is only supported for primary data files with an .mdf extension.

An error will be generated if a log file exists in the same directory as the data file and the 'database' keyword is used when attaching the primary data file. In this case, remove the log file. Once the database is attached, a new log file will be automatically generated based on the physical path.

See

Also

[Working with Connection Strings](#)

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Authentication SqlConnectionStringBuilder.Authentication

## In this Article

Gets the authentication of the connection string.

```
public System.Data.SqlClient.SqlAuthenticationMethod Authentication { get; set; }  
member this.Authentication : System.Data.SqlClient.SqlAuthenticationMethod with get, set
```

Returns

[SqlAuthenticationMethod](#) [SqlAuthenticationMethod](#)

The authentication of the connection string.

# SqlConnectionStringBuilder.Clear SqlConnectionStringBuilder.Clear

## In this Article

Clears the contents of the [SqlConnectionStringBuilder](#) instance.

```
public override void Clear ();  
override this.Clear : unit -> unit
```

## Examples

The following example demonstrates calling the [Clear](#) method. This example populates the [SqlConnectionStringBuilder](#) with some key/value pairs, and then calls the [Clear](#) method and shows the results.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();  
        builder.DataSource = "(local)";  
        builder.IntegratedSecurity = true;  
        builder.InitialCatalog = "AdventureWorks";  
        Console.WriteLine("Initial connection string: " + builder.ConnectionString);  
  
        builder.Clear();  
        Console.WriteLine("After call to Clear, count = " + builder.Count);  
        Console.WriteLine("Cleared connection string: " + builder.ConnectionString);  
        Console.WriteLine();  
  
        Console.WriteLine("Press Enter to continue.");  
        Console.ReadLine();  
    }  
}
```

## Remarks

The [Clear](#) method removes all key/value pairs from the [SqlConnectionStringBuilder](#), and resets all corresponding properties. This includes setting the [Count](#) property to 0, and setting the [ConnectionString](#) property to an empty string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.ColumnEncryptionSetting

## SqlConnectionStringBuilder.ColumnEncryptionSetting

### In this Article

Gets and sets the column encryption settings for the connection string builder.

```
public System.Data.SqlClient.SqlConnectionColumnEncryptionSetting ColumnEncryptionSetting { get;  
set; }  
  
member this.ColumnEncryptionSetting : System.Data.SqlClient.SqlConnectionColumnEncryptionSetting  
with get, set
```

Returns

[SqlConnectionStringColumnEncryptionSetting](#) [SqlConnectionStringColumnEncryptionSetting](#)

The column encryption settings for the connection string builder.

# SqlConnectionStringBuilder.ConnectionReset Sql ConnectionStringBuilder.ConnectionReset

## In this Article

Obsolete. Gets or sets a Boolean value that indicates whether the connection is reset when drawn from the connection pool.

```
[System.ComponentModel.Browsable(false)]
[System.Obsolete("ConnectionReset has been deprecated. SqlConnection will ignore the 'connection
reset' keyword and always reset the connection")]
public bool ConnectionReset { get; set; }

member this.ConnectionReset : bool with get, set
```

## Returns

[Boolean](#)

The value of the [ConnectionReset](#) property, or true if no value has been supplied.

## Attributes

[BrowsableAttribute](#) [ObsoleteAttribute](#)

## Remarks

This property corresponds to the "Connection Reset" key within the [SqlConnection](#) connection string, which has been removed from version 3.5 SP1 of the .NET Framework.

## See

[ADO.NET Overview](#)

## Also

# SqlConnectionStringBuilder.ConnectRetryCount Sql ConnectionStringBuilder.ConnectRetryCount

## In this Article

The number of reconnections attempted after identifying that there was an idle connection failure. This must be an integer between 0 and 255. Default is 1. Set to 0 to disable reconnecting on idle connection failures. An [ArgumentException](#) will be thrown if set to a value outside of the allowed range.

```
public int ConnectRetryCount { get; set; }  
member this.ConnectRetryCount : int with get, set
```

## Returns

[Int32](#) [Int32](#)

The number of reconnections attempted after identifying that there was an idle connection failure.

# SqlConnectionStringBuilder.ConnectRetryInterval Sql ConnectionStringBuilder.ConnectRetryInterval

## In this Article

Amount of time (in seconds) between each reconnection attempt after identifying that there was an idle connection failure. This must be an integer between 1 and 60. The default is 10 seconds. An [ArgumentException](#) will be thrown if set to a value outside of the allowed range.

```
public int ConnectRetryInterval { get; set; }  
member this.ConnectRetryInterval : int with get, set
```

## Returns

[Int32](#) [Int32](#)

Amount of time (in seconds) between each reconnection attempt after identifying that there was an idle connection failure.

# SqlConnectionStringBuilder.ConnectTimeout Sql ConnectionStringBuilder.ConnectTimeout

## In this Article

Gets or sets the length of time (in seconds) to wait for a connection to the server before terminating the attempt and generating an error.

```
public int ConnectTimeout { get; set; }  
member this.ConnectTimeout : int with get, set
```

## Returns

Int32 Int32

The value of the [ConnectTimeout](#) property, or 15 seconds if no value has been supplied.

## Examples

The following example first displays the contents of a connection string that does not specify the "Connect Timeout" value, sets the [ConnectTimeout](#) property, and then displays the new connection string.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        try  
        {  
            string connectString =  
                "Server=(local);Initial Catalog=AdventureWorks;" +  
                "Integrated Security=true";  
            SqlConnectionStringBuilder builder =  
                new SqlConnectionStringBuilder(connectString);  
            Console.WriteLine("Original: " + builder.ConnectionString);  
            Console.WriteLine("ConnectTimeout={0}",  
                builder.ConnectTimeout);  
            builder.ConnectTimeout = 100;  
            Console.WriteLine("Modified: " + builder.ConnectionString);  
  
            Console.WriteLine("Press any key to finish.");  
            Console.ReadLine();  
  
        }  
        catch (Exception ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}
```

## Remarks

This property corresponds to the "Connect Timeout", "connection timeout", and "timeout" keys within the connection string.

When opening a connection to a Azure SQL Database, set the connection timeout to 30 seconds.

See

[Working with Connection Strings](#)



# SqlConnectionStringBuilder.ContainsKey SqlConnection StringBuilder.ContainsKey

## In this Article

Determines whether the [SqlConnectionStringBuilder](#) contains a specific key.

```
public override bool ContainsKey (string keyword);  
override this.ContainsKey : string -> bool
```

### Parameters

keyword [String](#) [String](#)

The key to locate in the [SqlConnectionStringBuilder](#).

### Returns

[Boolean](#) [Boolean](#)

true if the [SqlConnectionStringBuilder](#) contains an element that has the specified key; otherwise, false.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`keyword` is null (`Nothing` in Visual Basic)

## Examples

The following example creates a [SqlConnectionStringBuilder](#) instance, sets some of its properties, and then tries to determine whether various keys exist within the object by calling the **ContainsKey** method.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        SqlConnectionStringBuilder builder =
            new SqlConnectionStringBuilder(GetConnectionString());
        Console.WriteLine("Connection string = " + builder.ConnectionString);

        // Keys you have provided return true.
        Console.WriteLine(builder.ContainsKey("Server"));

        // Comparison is case insensitive, and synonyms
        // are automatically converted to their "well-known"
        // names.
        Console.WriteLine(builder.ContainsKey("Database"));

        // Keys that are valid but have not been set return true.
        Console.WriteLine(builder.ContainsKey("Max Pool Size"));

        // Keys that do not exist return false.
        Console.WriteLine(builder.ContainsKey("MyKey"));

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

    private static string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Server=(local);Integrated Security=SSPI;" +
            "Initial Catalog=AdventureWorks";
    }
}

```

The example displays the following output in the console window:

```

Connection string = Data Source=(local);Initial Catalog=AdventureWorks;Integrated Security=True
True
True
True
False

```

## Remarks

Because the [SqlConnectionStringBuilder](#) contains a fixed-size collection of key/value pairs, the [ContainsKey](#) method determines only if a particular key name is valid.

See

Also

[Working with Connection Strings](#)  
[ADO.NET Overview](#)

# SqlConnectionStringBuilder.ContextConnection Sql ConnectionStringBuilder.ContextConnection

## In this Article

Gets or sets a value that indicates whether a client/server or in-process connection to SQL Server should be made.

```
public bool ContextConnection { get; set; }  
member this.ContextConnection : bool with get, set
```

Returns

[Boolean Boolean](#)

The value of the [ContextConnection](#) property, or `False` if none has been supplied.

## Remarks

This property corresponds to the "Context Connection" key within the connection string.

 [Note](#)

The [DataSource](#) property returns `null` if the connection string for the [SqlConnection](#) is "context connection=true".

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.CurrentLanguage Sql ConnectionStringBuilder.CurrentLanguage

## In this Article

Gets or sets the SQL Server Language record name.

```
public string CurrentLanguage { get; set; }  
member this.CurrentLanguage : string with get, set
```

Returns

[String](#) [String](#)

The value of the [CurrentLanguage](#) property, or [String.Empty](#) if no value has been supplied.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

To set the value to null, use [Value](#).

## Remarks

This property corresponds to the "Current Language" and "language" keys within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.DataSource SqlConnection StringBuilder.DataSource

## In this Article

Gets or sets the name or network address of the instance of SQL Server to connect to.

```
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlConnectionStringBuilder/SqlDataSourceConverter))]
public string DataSource { get; set; }

member this.DataSource : string with get, set
```

Returns

[String](#) [String](#)

The value of the [DataSource](#) property, or [String.Empty](#) if none has been supplied.

Attributes

[TypeConverterAttribute](#)

Exceptions

[ArgumentNullException](#) [ArgumentException](#)

To set the value to null, use [Value](#).

## Examples

The following example demonstrates that the [SqlConnectionStringBuilder](#) class converts synonyms for the "Data Source" connection string key into the well-known key:

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(
            "Network Address=(local);Integrated Security=SSPI;" +
            "Initial Catalog=AdventureWorks");

        // Display the connection string, which should now
        // contain the "Data Source" key, as opposed to the
        // supplied "Network Address".
        Console.WriteLine(builder.ConnectionString);

        // Retrieve the DataSource property.
        Console.WriteLine("DataSource = " + builder.DataSource);

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

## Remarks

This property corresponds to the "Data Source", "server", "address", "addr", and "network address" keys within the connection string. Regardless of which of these values has been supplied within the supplied connection string, the connection string created by the [SqlConnectionStringBuilder](#) will use the well-known "Data Source" key.

See

Also

[Working with Connection Strings](#)

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.EnclaveAttestationUrl Sql ConnectionStringBuilder.EnclaveAttestationUrl

## In this Article

Gets or sets the enclave attestation Url to be used with enclave based Always Encrypted.

```
public string EnclaveAttestationUrl { get; set; }  
member this.EnclaveAttestationUrl : string with get, set
```

Returns

[String](#) [String](#)

The enclave attestation Url.

# SqlConnectionStringBuilder.Encrypt SqlConnectionString Builder.Encrypt

## In this Article

Gets or sets a Boolean value that indicates whether SQL Server uses SSL encryption for all data sent between the client and server if the server has a certificate installed.

```
public bool Encrypt { get; set; }  
member this.Encrypt : bool with get, set
```

## Returns

[Boolean](#)

The value of the [Encrypt](#) property, or `false` if none has been supplied.

## Remarks

This property corresponds to the "Encrypt" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Enlist SqlConnectionStringBuilder.Enlist

## In this Article

Gets or sets a Boolean value that indicates whether the SQL Server connection pooler automatically enlists the connection in the creation thread's current transaction context.

```
public bool Enlist { get; set; }  
member this.Enlist : bool with get, set
```

## Returns

[Boolean](#)

The value of the [Enlist](#) property, or `true` if none has been supplied.

## Remarks

This property corresponds to the "Enlist" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.FailoverPartner Sql ConnectionStringBuilder.FailoverPartner

## In this Article

Gets or sets the name or address of the partner server to connect to if the primary server is down.

```
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlConnectionStringBuilder/SqlData
SourceConverter))]
public string FailoverPartner { get; set; }

member this.FailoverPartner : string with get, set
```

Returns

[String](#) [String](#)

The value of the [FailoverPartner](#) property, or [String.Empty](#) if none has been supplied.

Attributes

[TypeConverterAttribute](#)

Exceptions

[ArgumentNullException](#) [ArgumentException](#)

To set the value to null, use [Value](#).

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.InitialCatalog SqlConnection StringBuilder.InitialCatalog

## In this Article

Gets or sets the name of the database associated with the connection.

```
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlConnectionStringBuilder/SqlInitialCatalogConverter))]  
public string InitialCatalog { get; set; }  
  
member this.InitialCatalog : string with get, set
```

Returns

[String](#) [String](#)

The value of the [InitialCatalog](#) property, or [String.Empty](#) if none has been supplied.

Attributes

[TypeConverterAttribute](#)

Exceptions

[ArgumentNullException](#) [ArgumentException](#)

To set the value to null, use [Value](#).

## Examples

The following example creates a simple connection string and then uses the [SqlConnectionStringBuilder](#) class to add the name of the database to the connection string. The code displays the contents of the [InitialCatalog](#) property, just to verify that the class was able to convert from the synonym ("Database") to the appropriate property value.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        try
        {
            string connectString = "Data Source=(local);" +
                "Integrated Security=true";

            SqlConnectionStringBuilder builder =
                new SqlConnectionStringBuilder(connectString);
            Console.WriteLine("Original: " + builder.ConnectionString);

            // Normally, you could simply set the InitialCatalog
            // property of the SqlConnectionStringBuilder object. This
            // example uses the default Item property (the C# indexer)
            // and the "Database" string, simply to demonstrate that
            // setting the value in this way results in the same
            // connection string:
            builder["Database"] = "AdventureWorks";
            Console.WriteLine("builder.InitialCatalog = "
                + builder.InitialCatalog);
            Console.WriteLine("Modified: " + builder.ConnectionString);

            using (SqlConnection connection =
                new SqlConnection(builder.ConnectionString))
            {
                connection.Open();
                // Now use the open connection.
                Console.WriteLine("Database = " + connection.Database);
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

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

```

## Remarks

This property corresponds to the "Initial Catalog" and "database" keys within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.IntegratedSecurity SqlConnectionStringBuilder.IntegratedSecurity

## In this Article

Gets or sets a Boolean value that indicates whether User ID and Password are specified in the connection (when `false`) or whether the current Windows account credentials are used for authentication (when `true`).

```
public bool IntegratedSecurity { get; set; }  
member this.IntegratedSecurity : bool with get, set
```

## Returns

[Boolean](#) [Boolean](#)

The value of the [IntegratedSecurity](#) property, or `false` if none has been supplied.

## Examples

The following example converts an existing connection string from using SQL Server Authentication to using integrated security. The example does its work by removing the user name and password from the connection string and then setting the [IntegratedSecurity](#) property of the [SqlConnectionStringBuilder](#) object.

### Note

This example includes a password to demonstrate how [SqlConnectionStringBuilder](#) works with connection strings. In your applications, we recommend that you use Windows Authentication. If you must use a password, do not include a hard-coded password in your application.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        try
        {
            string connectString =
                "Data Source=(local);User ID=ab;Password=MyPassword;" +
                "Initial Catalog=AdventureWorks";

            SqlConnectionStringBuilder builder =
                new SqlConnectionStringBuilder(connectString);
            Console.WriteLine("Original: " + builder.ConnectionString);

            // Use the Remove method
            // in order to reset the user ID and password back to their
            // default (empty string) values. Simply setting the
            // associated property values to an empty string won't
            // remove them from the connection string; you must
            // call the Remove method.
            builder.Remove("User ID");
            builder.Remove("Password");

            // Turn on integrated security:
            builder.IntegratedSecurity = true;

            Console.WriteLine("Modified: " + builder.ConnectionString);

            using (SqlConnection connection =
                new SqlConnection(builder.ConnectionString))
            {
                connection.Open();
                // Now use the open connection.
                Console.WriteLine("Database = " + connection.Database);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

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

```

## Remarks

This property corresponds to the "Integrated Security" and "trusted\_connection" keys within the connection string.

See

Also

[Working with Connection Strings](#)  
[ADO.NET Overview](#)

# SqlConnectionStringBuilder.IsFixedSize SqlConnection StringBuilder.IsFixedSize

## In this Article

Gets a value that indicates whether the [SqlConnectionStringBuilder](#) has a fixed size.

```
public override bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean](#) [Boolean](#)

`true` in every case, because the [SqlConnectionStringBuilder](#) supplies a fixed-size collection of key/value pairs.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Item[String] SqlConnection StringBuilder.Item[String]

## In this Article

Gets or sets the value associated with the specified key. In C#, this property is the indexer.

```
public override object this[string keyword] { get; set; }  
member this.Item(string) : obj with get, set
```

### Parameters

keyword String String

The key of the item to get or set.

### Returns

[Object Object](#)

The value associated with the specified key.

### Exceptions

[ArgumentNullException ArgumentNullException](#)

`keyword` is a null reference (`Nothing` in Visual Basic).

[KeyNotFoundException KeyNotFoundException](#)

Tried to add a key that does not exist within the available keys.

[FormatException FormatException](#)

Invalid value within the connection string (specifically, a Boolean or numeric value was expected but not supplied).

## Examples

The following code, in a console application, creates a new [SqlConnectionStringBuilder](#) and adds key/value pairs to its connection string, using the [Item\[String\]](#) property.

```
class Program  
{  
    static void Main()  
    {  
        SqlConnectionStringBuilder builder =  
            new SqlConnectionStringBuilder();  
        builder["Data Source"] = "(local)";  
        builder["Integrated Security"] = true;  
        builder["Initial Catalog"] = "AdventureWorks";  
  
        // Overwrite the existing value for the Data Source value.  
        builder["Data Source"] = ".";  
  
        Console.WriteLine(builder.ConnectionString);  
        Console.WriteLine();  
        Console.WriteLine("Press Enter to continue.");  
        Console.ReadLine();  
    }  
}
```

## Remarks

Because the [SqlConnectionStringBuilder](#) contains a fixed-size dictionary, trying to add a key that does not exist within the dictionary throws a [KeyNotFoundException](#).

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Keys SqlConnectionString Builder.Keys

## In this Article

Gets an [ICollection](#) that contains the keys in the [SqlConnectionStringBuilder](#).

```
public override System.Collections.ICollection Keys { get; }  
member this.Keys : System.Collections.ICollection
```

## Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) that contains the keys in the [SqlConnectionStringBuilder](#).

## Examples

The following console application example creates a new [SqlConnectionStringBuilder](#). The code loops through the [ICollection](#) returned by the [Keys](#) property displaying the key/value pairs.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        SqlConnectionStringBuilder builder =  
            new SqlConnectionStringBuilder();  
        builder.DataSource = "(local)";  
        builder.IntegratedSecurity = true;  
        builder.InitialCatalog = "AdventureWorks";  
  
        // Loop through the collection of keys, displaying  
        // the key and value for each item:  
        foreach (string key in builder.Keys)  
            Console.WriteLine("{0}={1}", key, builder[key]);  
  
        Console.WriteLine();  
        Console.WriteLine("Press Enter to continue.");  
        Console.ReadLine();  
    }  
}
```

## Remarks

The order of the values in the [ICollection](#) is unspecified, but it is the same order as the associated values in the [ICollection](#) returned by the [Values](#) property.

See

[Values](#)

Also

[Item\[String\]](#)

[Item\[String\]](#)

[Working with Connection Strings](#)

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.LoadBalanceTimeout Sql ConnectionStringBuilder.LoadBalanceTimeout

## In this Article

Gets or sets the minimum time, in seconds, for the connection to live in the connection pool before being destroyed.

```
public int LoadBalanceTimeout { get; set; }  
member this.LoadBalanceTimeout : int with get, set
```

Returns

[Int32](#) [Int32](#)

The value of the [LoadBalanceTimeout](#) property, or 0 if none has been supplied.

## Remarks

This property corresponds to the "Load Balance Timeout" and "connection lifetime" keys within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.MaxPoolSize SqlConnection StringBuilder.MaxPoolSize

## In this Article

Gets or sets the maximum number of connections allowed in the connection pool for this specific connection string.

```
public int MaxPoolSize { get; set; }  
member this.MaxPoolSize : int with get, set
```

## Returns

[Int32](#) [Int32](#)

The value of the [MaxPoolSize](#) property, or 100 if none has been supplied.

## Remarks

This property corresponds to the "Max Pool Size" key within the connection string.

### See

[Working with Connection Strings](#)

### Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.MinPoolSize SqlConnection StringBuilder.MinPoolSize

## In this Article

Gets or sets the minimum number of connections allowed in the connection pool for this specific connection string.

```
public int MinPoolSize { get; set; }  
member this.MinPoolSize : int with get, set
```

## Returns

[Int32](#) [Int32](#)

The value of the [MinPoolSize](#) property, or 0 if none has been supplied.

## Remarks

This property corresponds to the "Min Pool Size" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.MultipleActiveResultSets SqlConnectionStringBuilder.MultipleActiveResultSets

## In this Article

When true, an application can maintain multiple active result sets (MARS). When false, an application must process or cancel all result sets from one batch before it can execute any other batch on that connection.

For more information, see [Multiple Active Result Sets \(MARS\)](#).

```
public bool MultipleActiveResultSets { get; set; }  
member this.MultipleActiveResultSets : bool with get, set
```

Returns

[Boolean](#) [Boolean](#)

The value of the [MultipleActiveResultSets](#) property, or `false` if none has been supplied.

## Examples

The following example explicitly disables the Multiple Active Result Sets feature.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();  
        builder.DataSource = "(local)";  
        builder.IntegratedSecurity = true;  
        builder.InitialCatalog = "AdventureWorks";  
  
        // The connection does not allow multiple active result sets  
        // by default, so this line of code explicitly  
        // enables this feature. Note that this feature is  
        // only available when programming against SQL Server 2005  
        // or later.  
        builder.MultipleActiveResultSets = true;  
  
        Console.WriteLine(builder.ConnectionString);  
        Console.WriteLine();  
  
        Console.WriteLine("Press Enter to continue.");  
        Console.ReadLine();  
    }  
}
```

## Remarks

This property corresponds to the "MultipleActiveResultSets" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.MultiSubnetFailover Sql ConnectionStringBuilder.MultiSubnetFailover

## In this Article

If your application is connecting to an AlwaysOn availability group (AG) on different subnets, setting MultiSubnetFailover=true provides faster detection of and connection to the (currently) active server. For more information about SqlClient support for Always On Availability Groups, see [SqlClient Support for High Availability, Disaster Recovery](#).

```
public bool MultiSubnetFailover { get; set; }  
member this.MultiSubnetFailover : bool with get, set
```

## Returns

**Boolean Boolean**

Returns [Boolean](#) indicating the current value of the property.

## See

[ADO.NET Overview](#)

## Also

# SqlConnectionStringBuilder.NetworkLibrary Sql ConnectionStringBuilder.NetworkLibrary

## In this Article

Gets or sets a string that contains the name of the network library used to establish a connection to the SQL Server.

```
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlConnectionStringBuilder/Network  
LibraryConverter))]  
public string NetworkLibrary { get; set; }  
  
member this.NetworkLibrary : string with get, set
```

Returns

[String](#) [String](#)

The value of the [NetworkLibrary](#) property, or [String.Empty](#) if none has been supplied.

Attributes

[TypeConverterAttribute](#)

Exceptions

[ArgumentNullException](#) [ArgumentException](#)

To set the value to null, use [Value](#).

## Remarks

This property corresponds to the "Network Library", "network", and "net" keys within the connection string.

Supported values for this property include dbnmpntw (Named Pipes), dbmsrpcn (Multiprotocol), dbmsadsn (AppleTalk), dbmsgnet (VIA), dbmslpcn (Shared Memory) and dbmsspxn (IPX/SPX), and dbmssocn (TCP/IP). The corresponding network DLL must be installed on the system to which you connect. If you do not specify a network and you use a local server (for example, "." or "(local)"), Shared Memory is used.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.PacketSize SqlConnection StringBuilder.PacketSize

## In this Article

Gets or sets the size in bytes of the network packets used to communicate with an instance of SQL Server.

```
public int PacketSize { get; set; }  
  
member this.PacketSize : int with get, set
```

Returns

[Int32](#) [Int32](#)

The value of the [PacketSize](#) property, or 8000 if none has been supplied.

## Remarks

This property corresponds to the "Packet Size" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Password SqlConnection StringBuilder.Password

## In this Article

Gets or sets the password for the SQL Server account.

```
public string Password { get; set; }  
member this.Password : string with get, set
```

Returns

[String String](#)

The value of the [Password](#) property, or `String.Empty` if none has been supplied.

Exceptions

[ArgumentNullException ArgumentNullException](#)

The password was incorrectly set to null. See code sample below.

## Examples

The following example shows how to set [Password](#).

```
using System;  
using System.Data.SqlClient;  
  
class Program {  
    public static void Main() {  
        SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();  
  
        builder["Password"] = null;  
        string aa = builder.Password;  
        Console.WriteLine(aa.Length);  
  
        builder["Password"] = "??????";  
        aa = builder.Password;  
        Console.WriteLine(aa.Length);  
  
        try {  
            builder.Password = null;  
        }  
        catch (ArgumentNullException e) {  
            Console.WriteLine("{0}", e);  
        }  
    }  
}
```

## Remarks

This property corresponds to the "Password" and "pwd" keys within the connection string.

If [Password](#) has not been set and you retrieve the value, the return value is [Empty](#). To reset the password for the connection string, pass null to the [Item](#) property.

See

Also

[Working with Connection Strings](#)

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.PersistSecurityInfo Sql ConnectionStringBuilder.PersistSecurityInfo

## In this Article

Gets or sets a Boolean value that indicates if security-sensitive information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open state.

```
public bool PersistSecurityInfo { get; set; }  
member this.PersistSecurityInfo : bool with get, set
```

## Returns

[Boolean](#) [Boolean](#)

The value of the [PersistSecurityInfo](#) property, or `false` if none has been supplied.

## Remarks

This property corresponds to the "Persist Security Info" and "persistsecurityinfo" keys within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.PoolBlockingPeriod Sql ConnectionStringBuilder.PoolBlockingPeriod

## In this Article

The blocking period behavior for a connection pool.

```
public System.Data.SqlClient.PoolBlockingPeriod PoolBlockingPeriod { get; set; }  
member this.PoolBlockingPeriod : System.Data.SqlClient.PoolBlockingPeriod with get, set
```

Returns

[PoolBlockingPeriod](#) **PoolBlockingPeriod**

The available blocking period settings.

## Remarks

When connection pooling is enabled and a timeout error or other login error occurs, an exception will be thrown and subsequent connection attempts will fail for the next five seconds, the "blocking period". If the application attempts to connect within the blocking period, the first exception will be thrown again. Subsequent failures after a blocking period ends will result in a new blocking period that is twice as long as the previous blocking period, up to a maximum of one minute.

Attempting to connect to Azure SQL databases can fail with transient errors which are typically recovered within a few seconds. However, with the connection pool blocking period behavior, you may not be able to reach your database for extensive periods even though the database is available. This is especially problematic for apps that need to render fast. The **PoolBlockingPeriod** enables you to select the blocking period best suited for your app. See the [PoolBlockingPeriod](#) enumeration for available settings.

# SqlConnectionStringBuilder.Pooling SqlConnectionStringBuilder.Pooling

## In this Article

Gets or sets a Boolean value that indicates whether the connection will be pooled or explicitly opened every time that the connection is requested.

```
public bool Pooling { get; set; }  
member this.Pooling : bool with get, set
```

## Returns

[Boolean](#)

The value of the [Pooling](#) property, or `true` if none has been supplied.

## Remarks

This property corresponds to the "Pooling" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Remove SqlConnectionStringBuilder.Remove

## In this Article

Removes the entry with the specified key from the [SqlConnectionStringBuilder](#) instance.

```
public override bool Remove (string keyword);  
override this.Remove : string -> bool
```

### Parameters

keyword [String](#) [String](#)

The key of the key/value pair to be removed from the connection string in this [SqlConnectionStringBuilder](#).

### Returns

[Boolean](#) [Boolean](#)

`true` if the key existed within the connection string and was removed; `false` if the key did not exist.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`keyword` is null (`Nothing` in Visual Basic)

## Examples

The following example converts an existing connection string from using Windows Authentication to using integrated security. The example works by removing the user name and password from the connection string, and then setting the [IntegratedSecurity](#) property of the [SqlConnectionStringBuilder](#) object.

 **Note**

This example includes a password to demonstrate how [SqlConnectionStringBuilder](#) works with connection strings. In your applications, we recommend that you use Windows Authentication. If you must use a password, do not include a hard-coded password in your application.

```

using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        try
        {
            string connectString =
                "Data Source=(local);User ID=ab;Password= a1Pass@@11;" +
                "Initial Catalog=AdventureWorks";

            SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder(connectString);
            Console.WriteLine("Original: " + builder.ConnectionString);

            // Use the Remove method
            // in order to reset the user ID and password back to their
            // default (empty string) values.
            builder.Remove("User ID");
            builder.Remove("Password");

            // Turn on integrated security:
            builder.IntegratedSecurity = true;

            Console.WriteLine("Modified: " + builder.ConnectionString);

            using (SqlConnection
                    connection = new SqlConnection(builder.ConnectionString))
            {
                connection.Open();
                // Now use the open connection.
                Console.WriteLine("Database = " + connection.Database);
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

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

```

The example displays the following text in the console window:

```

Original: Data Source=(local);Initial Catalog=AdventureWorks;User ID=ab;Password= a1Pass@@11
Modified: Data Source=(local);Initial Catalog=AdventureWorks;Integrated Security=True
Database = AdventureWorks

```

## Remarks

Because the **Remove** method returns a value that indicates its success, it is not required to look for a key before trying to remove the key/value pair from the [SqlConnectionStringBuilder](#) instance. Because the [SqlConnectionStringBuilder](#) maintains a fixed-size collection of key/value pairs, calling the [Remove](#) method simply resets the value of the key/value pair back to its default value.

Because the collection of keys supported by the [SqlConnectionStringBuilder](#) is fixed, every item within the collection has a known default value. The following table lists the keys, and the value for each when the [SqlConnectionStringBuilder](#) is first initialized, or after the [Remove](#) method has been called.

| KEY                      | DEFAULT VALUE                  |
|--------------------------|--------------------------------|
| Application Name         | ".Net SqlClient Data Provider" |
| Asynchronous Processing  | False                          |
| AttachDBFilename         | Empty string                   |
| Connection Timeout       | 15                             |
| Context Connection       | False                          |
| Current Language         | Empty string                   |
| Data Source              | Empty string                   |
| Encrypt                  | False                          |
| Enlist                   | True                           |
| Failover Partner         | Empty string                   |
| Initial Catalog          | Empty string                   |
| Integrated Security      | False                          |
| Load Balance Timeout     | 0                              |
| Max Pool Size            | 100                            |
| Min Pool Size            | 0                              |
| MultipleActiveResultSets | False                          |
| Network Library          | Empty string                   |
| Packet Size              | 8000                           |
| Password                 | Empty string                   |
| Persist Security Info    | False                          |
| Pooling                  | True                           |
| Replication              | False                          |
| Transaction Binding      | Implicit Unbind                |
| User ID                  | Empty string                   |
| User Instance            | False                          |

| KEY            | DEFAULT VALUE |
|----------------|---------------|
| Workstation ID | Empty string  |

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Replication SqlConnection StringBuilder.Replication

## In this Article

Gets or sets a Boolean value that indicates whether replication is supported using the connection.

```
public bool Replication { get; set; }  
member this.Replication : bool with get, set
```

Returns

[Boolean](#)

The value of the [Replication](#) property, or false if none has been supplied.

## Remarks

This property corresponds to the "Replication" key within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.ShouldSerialize SqlConnectionStringBuilder.ShouldSerialize

## In this Article

Indicates whether the specified key exists in this [SqlConnectionStringBuilder](#) instance.

```
public override bool ShouldSerialize (string keyword);  
override this.ShouldSerialize : string -> bool
```

### Parameters

keyword [String](#) [String](#)

The key to locate in the [SqlConnectionStringBuilder](#).

### Returns

[Boolean](#) [Boolean](#)

`true` if the [SqlConnectionStringBuilder](#) contains an entry with the specified key; otherwise, `false`.

## Remarks

This method behaves identically to the [ContainsKey](#) method.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder SqlConnectionStringBuilder

In this Article

## Overloads

|  |  |
|--|--|
| <code>SqlConnectionStringBuilder()</code>  | Initializes a new instance of the <a href="#">SqlConnectionStringBuilder</a> class.  |
| <code>SqlConnectionStringBuilder(String) SqlConnectionStringBuilder(String)</code> | Initializes a new instance of the <a href="#">SqlConnectionStringBuilder</a> class. The provided connection string provides the data for the instance's internal connection information. |

## SqlConnectionStringBuilder()

Initializes a new instance of the [SqlConnectionStringBuilder](#) class.

```
public SqlConnectionStringBuilder () ;
```

See

[ADO.NET Overview](#)

Also

## SqlConnectionStringBuilder(String) SqlConnectionStringBuilder(String)

Initializes a new instance of the [SqlConnectionStringBuilder](#) class. The provided connection string provides the data for the instance's internal connection information.

```
public SqlConnectionStringBuilder (string connectionString);  
  
new System.Data.SqlClient.SqlConnectionStringBuilder : string ->  
System.Data.SqlClient.SqlConnectionStringBuilder
```

Parameters

connectionString [String](#) [String](#)

The basis for the object's internal connection information. Parsed into name/value pairs. Invalid key names raise [KeyNotFoundException](#).

Exceptions

[KeyNotFoundException](#) [KeyNotFoundException](#)

Invalid key name within the connection string.

[FormatException](#) [FormatException](#)

Invalid value within the connection string (specifically, when a Boolean or numeric value was expected but not supplied).

[ArgumentException](#) [ArgumentException](#)

The supplied `connectionString` is not valid.

## Examples

The following example supplies a simple SQL Server connection string in the [SqlConnectionStringBuilder](#) object's constructor, and then iterates through all the key/value pairs within the object. Note that the collection provides default values for each item. Also note that the [SqlConnectionStringBuilder](#) class converts synonyms for the well-known keys so that they are consistent with the well-known names.

### Note

This example includes a password to demonstrate how [SqlConnectionStringBuilder](#) works with connection strings. In your applications, we recommend that you use Windows Authentication. If you must use a password, do not include a hard-coded password in your application.

```
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        try
        {
            string connectString =
                "Server=(local);Database=AdventureWorks;UID=ab;Pwd= a!Pass@0";
            Console.WriteLine("Original: " + connectString);
            SqlConnectionStringBuilder builder =
                new SqlConnectionStringBuilder(connectString);
            Console.WriteLine("Modified: " + builder.ConnectionString);
            foreach (string key in builder.Keys)
                Console.WriteLine(key + "=" + builder[key].ToString());
            Console.WriteLine("Press any key to finish.");
            Console.ReadLine();

        }
        catch (System.Collections.Generic.KeyNotFoundException ex)
        {
            Console.WriteLine("KeyNotFoundException: " + ex.Message);
        }
        catch (System.FormatException ex)
        {
            Console.WriteLine("Format exception: " + ex.Message);
        }
    }
}
```

## Remarks

The [SqlConnectionStringBuilder](#) class provides a fixed internal collection of key/value pairs. Even if you supply only a small subset of the possible connection string values in the constructor, the object always provides default values for each key/value pair. When the `ConnectionString` property of the object is retrieved, the string contains only key/value pairs in which the value is not the default value for the item.

See

[Connection Strings \(ADO.NET\)](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.TransactionBinding SqlConnectionStringBuilder.TransactionBinding

## In this Article

Gets or sets a string value that indicates how the connection maintains its association with an enlisted `System.Transactions` transaction.

```
public string TransactionBinding { get; set; }  
member this.TransactionBinding : string with get, set
```

## Returns

[String](#) [String](#)

The value of the [TransactionBinding](#) property, or `String.Empty` if none has been supplied.

## Remarks

The Transaction Binding keywords in a [ConnectionString](#) control how a [SqlConnection](#) binds to an enlisted [Transaction](#).

The following table shows the possible values for the [TransactionBinding](#) property:

| VALUE           | DESCRIPTION  |
|-----------------|--|
| Implicit Unbind | The default. Causes the connection to detach from the transaction when it ends. After detaching, additional requests on the connection are performed in autocommit mode. The <a href="#">Current</a> property is not checked when executing requests while the transaction is active. After the transaction has ended, additional requests are performed in autocommit mode.   |
| Explicit Unbind | Causes the connection to remain attached to the transaction until the connection is closed or until <a href="#">EnlistTransaction</a> is called with a <code>null</code> ( <code>Nothing</code> in Visual Basic) value. An <a href="#">InvalidOperationException</a> is thrown if <a href="#">Current</a> is not the enlisted transaction or if the enlisted transaction is not active. This behavior enforces the strict scoping rules required for <a href="#">TransactionScope</a> support. |

See

[Working with Connection Strings](#)

Also

[Distributed Transactions \(ADO.NET\)](#)

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.TransparentNetworkIPResolution

## SqlConnectionStringBuilder.TransparentNetworkIPResolution

### In this Article

When the value of this key is set to `true`, the application is required to retrieve all IP addresses for a particular DNS entry and attempt to connect with the first one in the list. If the connection is not established within 0.5 seconds, the application will try to connect to all others in parallel. When the first answers, the application will establish the connection with the respondent IP address.

```
public bool TransparentNetworkIPResolution { get; set; }  
member this.TransparentNetworkIPResolution : bool with get, set
```

### Returns

[Boolean](#)

A boolean value.

## Remarks

If the `MultiSubnetFailover` key is set to `true`, `TransparentNetworkIPResolution` is ignored.

If the `Failover Partner` key is set, `TransparentNetworkIPResolution` is ignored.

The value of this key must be `true`, `false`, `yes`, or `no`.

A value of `yes` is treated the same as a value of `true`.

A value of `no` is treated the same as a value of `false`.

This key defaults to `false` when:

- Connecting to Azure SQL Database where the data source ends with:
  - `.database.chinacloudapi.cn`
  - `.database.usgovcloudapi.net`
  - `.database.cloudapi.de`
  - `.database.windows.net`
- `Authentication` is 'Active Directory Password' or 'Active Directory Integrated'

Otherwise it defaults to `true`.

# SqlConnectionStringBuilder.TrustServerCertificate Sql ConnectionStringBuilder.TrustServerCertificate

## In this Article

Gets or sets a value that indicates whether the channel will be encrypted while bypassing walking the certificate chain to validate trust.

```
public bool TrustServerCertificate { get; set; }  
member this.TrustServerCertificate : bool with get, set
```

## Returns

[Boolean](#)

A [Boolean](#). Recognized values are `true`, `false`, `yes`, and `no`.

## Remarks

When `TrustServerCertificate` is set to `true`, the transport layer will use SSL to encrypt the channel and bypass walking the certificate chain to validate trust. If `TrustServerCertificate` is set to `true` and encryption is turned on, the encryption level specified on the server will be used even if `Encrypt` is set to `false`. The connection will fail otherwise.

For more information, see [Encryption Hierarchy](#) and [Using Encryption Without Validation](#).

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.TryGetValue SqlConnection StringBuilder.TryGetValue

## In this Article

Retrieves a value corresponding to the supplied key from this [SqlConnectionStringBuilder](#).

```
public override bool TryGetValue (string keyword, out object value);  
override this.TryGetValue : string * -> bool
```

### Parameters

keyword [String](#) [String](#)

The key of the item to retrieve.

value [Object](#) [Object](#)

The value corresponding to `keyword`.

### Returns

[Boolean](#) [Boolean](#)

`true` if `keyword` was found within the connection string; otherwise, `false`.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`keyword` contains a null value (`Nothing` in Visual Basic).

## Examples

The following example demonstrates the behavior of the **TryGetValue** method.

```

using System.Data.SqlClient;
using System;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
        builder.ConnectionString = GetConnectionString();

        // Call TryGetValue method for multiple
        // key names. Note that these keys are converted
        // to well-known synonyms for data retrieval.
        DisplayValue(builder, "Data Source");
        DisplayValue(builder, "Trusted_Connection");
        DisplayValue(builder, "InvalidKey");
        DisplayValue(builder, null);

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

    private static void DisplayValue(
        SqlConnectionStringBuilder builder, string key)
    {
        object value = null;

        // Although TryGetValue handles missing keys,
        // it doesn't handle passing in a null
        // key. This example traps for that particular error, but
        // passes any other unknown exceptions back out to the
        // caller.
        try
        {
            if (builder.TryGetValue(key, out value))
            {
                Console.WriteLine("{0}={1}", key, value);
            }
            else
            {
                Console.WriteLine("Unable to retrieve value for '{0}'", key);
            }
        }
        catch (ArgumentNullException)
        {
            Console.WriteLine("Unable to retrieve value for null key.");
        }
    }

    private static string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Server=(local);Integrated Security=SSPI;" +
            "Initial Catalog=AdventureWorks";
    }
}

```

The sample displays the following results:

```
Data Source=(local)  
Trusted_Connection=True  
Unable to retrieve value for 'InvalidKey'  
Unable to retrieve value for null key.
```

## Remarks

The [TryGetValue](#) method lets developers safely retrieve a value from a [SqlConnectionStringBuilder](#) without needing to verify that the supplied key name is a valid key name. Because **TryGetValue** does not raise an exception when you call it, passing in a nonexistent key, you do not have to look for a key before retrieving its value. Calling **TryGetValue** with a nonexistent key will place the value null (`Nothing` in Visual Basic) in the `value` parameter.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.TypeSystemVersion Sql ConnectionStringBuilder.TypeSystemVersion

## In this Article

Gets or sets a string value that indicates the type system the application expects.

```
public string TypeSystemVersion { get; set; }  
member this.TypeSystemVersion : string with get, set
```

## Returns

[String](#) [String](#)

The following table shows the possible values for the [TypeSystemVersion](#) property:

| VALUE           | DESCRIPTION  |
|-----------------|--|
| SQL Server 2005 | Uses the SQL Server 2005 type system. No conversions are made for the current version of ADO.NET.  |
| SQL Server 2008 | Uses the SQL Server 2008 type system.  |
| Latest          | Use the latest version than this client-server pair can handle. This will automatically move forward as the client and server components are upgraded. |

## Remarks

The `TypeSystemVersion` property can be used to specify a down-level version of SQL Server for applications written against that version. This avoids possible problems with incompatible types in a newer version of SQL Server that may cause the application to break.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.UserID SqlConnectionString Builder.UserID

## In this Article

Gets or sets the user ID to be used when connecting to SQL Server.

```
public string UserID { get; set; }  
member this.UserID : string with get, set
```

Returns

[String String](#)

The value of the [UserID](#) property, or [String.Empty](#) if none has been supplied.

Exceptions

[ArgumentNullException ArgumentNullException](#)

To set the value to null, use [Value](#).

## Remarks

This property corresponds to the "User ID", "user", and "uid" keys within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.UserInstance SqlConnection StringBuilder.UserInstance

## In this Article

Gets or sets a value that indicates whether to redirect the connection from the default SQL Server Express instance to a runtime-initiated instance running under the account of the caller.

```
public bool UserInstance { get; set; }  
member this.UserInstance : bool with get, set
```

## Returns

[Boolean](#) [Boolean](#)

The value of the [UserInstance](#) property, or `False` if none has been supplied.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

To set the value to null, use [Value](#).

## Remarks

This property corresponds to the "User Instance" key within the connection string.

### Note

This feature is available only with the SQL Server Express Edition. For more information on user instances, see [SQL Server Express User Instances](#).

### See

[Working with Connection Strings](#)

### Also

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.Values SqlConnectionStringBuilder.Values

## In this Article

Gets an [ICollection](#) that contains the values in the [SqlConnectionStringBuilder](#).

```
public override System.Collections.ICollection Values { get; }  
member this.Values : System.Collections.ICollection
```

## Returns

[ICollection](#) [ICollection](#)

An [ICollection](#) that contains the values in the [SqlConnectionStringBuilder](#).

## Examples

The following example first creates a new [SqlConnectionStringBuilder](#), and then iterates through all the values within the object.

```
using System.Data.SqlClient;  
  
class Program  
{  
    static void Main()  
    {  
        SqlConnectionStringBuilder builder =  
            new SqlConnectionStringBuilder(GetConnectionString());  
  
        // Loop through each of the values, displaying the contents.  
        foreach (object value in builder.Values)  
            Console.WriteLine(value);  
  
        Console.WriteLine("Press any key to continue.");  
        Console.ReadLine();  
    }  
  
    private static string GetConnectionString()  
    {  
        // To avoid storing the connection string in your code,  
        // you can retrieve it from a configuration file.  
        return "Data Source=(local);Integrated Security=SSPI;" +  
            "Initial Catalog=AdventureWorks; Asynchronous Processing=true";  
    }  
}
```

## Remarks

The order of the values in the [ICollection](#) is unspecified, but it is the same order as the associated keys in the [ICollection](#) returned by the [Keys](#) property. Because each instance of the [SqlConnectionStringBuilder](#) always contains the same fixed set of keys, the [Values](#) property always returns the values corresponding to the fixed set of keys, in the same order as the keys.

See

[Keys](#)

Also

[Working with Connection Strings](#)

[ADO.NET Overview](#)

# SqlConnectionStringBuilder.WorkstationID Sql ConnectionStringBuilder.WorkstationID

## In this Article

Gets or sets the name of the workstation connecting to SQL Server.

```
public string WorkstationID { get; set; }  
member this.WorkstationID : string with get, set
```

Returns

[String](#) [String](#)

The value of the [WorkstationID](#) property, or [String.Empty](#) if none has been supplied.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

To set the value to null, use [Value](#).

## Remarks

This property corresponds to the "Workstation ID" and "wsid" keys within the connection string.

See

[Working with Connection Strings](#)

Also

[ADO.NET Overview](#)

# SqlCredential SqlCredential Class

[SqlCredential](#) provides a more secure way to specify the password for a login attempt using SQL Server Authentication.

[SqlCredential](#) is comprised of a user id and a password that will be used for SQL Server Authentication. The password in a [SqlCredential](#) object is of type [SecureString](#).

[SqlCredential](#) cannot be inherited.

Windows Authentication (`Integrated Security = true`) remains the most secure way to log in to a SQL Server database.

## Declaration

```
[Serializable]
public sealed class SqlCredential
type SqlCredential = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

Use [Credential](#) to get or set a connection's [SqlCredential](#) object. Use [ChangePassword](#) to change the password for a [SqlCredential](#) object. For information on how a [SqlCredential](#) object affects connection pool behavior, see [SQL Server Connection Pooling \(ADO.NET\)](#).

An [InvalidOperationException](#) exception will be raised if a non-null [SqlCredential](#) object is used in a connection with any of the following connection string keywords:

- `Integrated Security = true`
- `Password`
- `User ID`
- `Context Connection = true`

The following sample connects to a SQL Server database using [Credential](#):

```
// change connection string in the APP.CONFIG file
<connectionStrings>
  <add name="MyConnString"
    connectionString="Initial Catalog=myDB;Server=myServer"
    providerName="System.Data.SqlClient" />
</connectionStrings>

// then use the following snippet:
using System.Configuration;

System.Windows.Controls.TextBox txtUserId = new System.Windows.Controls.TextBox();
System.Windows.Controls.PasswordBox txtPwd = new System.Windows.Controls.PasswordBox();

Configuration config = Configuration.WebConfigurationManager.OpenWebConfiguration(null);
ConnectionStringSettings connString = config.ConnectionStrings.ConnectionString["MyConnString"];

using (SqlConnection conn = new SqlConnection(connString.ConnectionString))
{
  SecureString pwd = txtPwd.SecurePassword;
  pwd.MakeReadOnly();
  SqlCredential cred = new SqlCredential(txtUserId.Text, pwd);
  conn.Credential = cred;
  conn.Open();
```

## Constructors

[SqlCredential\(String, SecureString\)](#)

[SqlCredential\(String, SecureString\)](#)

Creates an object of type [SqlCredential](#).

## Properties

[Password](#)

[Password](#)

Returns the password component of the [SqlCredential](#) object.

[UserId](#)

[UserId](#)

Returns the user ID component of the [SqlCredential](#) object.

## See Also

# SqlCredential.Password SqlCredential.Password

## In this Article

Returns the password component of the [SqlCredential](#) object.

```
public System.Security.SecureString Password { get; }  
member this.Password : System.Security.SecureString
```

Returns

[SecureString](#) [SecureString](#)

Returns the password component of the [SqlCredential](#) object.

See

[ADO.NET Overview](#)

Also

# SqlCredential SqlCredential

## In this Article

Creates an object of type [SqlCredential](#).

```
public SqlCredential (string user, System.Security.SecureString password);  
new System.Data.SqlClient.SqlCredential : string * System.Security.SecureString ->  
System.Data.SqlClient.SqlCredential
```

## Parameters

userId [String](#) [String](#)

The user id.

password [SecureString](#) [SecureString](#)

The password; a [SecureString](#) value marked as read-only. Passing a read/write [SecureString](#) parameter will raise an [ArgumentException](#).

## Remarks

The constructor does not accept null parameters. A [Empty](#) value is allowed. An attempt to pass a null parameter in the constructor will raise an [ArgumentNullException](#) exception.

See [ADO.NET Overview](#)

Also

# SqlCredential.UserId SqlCredential.UserId

## In this Article

Returns the user ID component of the [SqlCredential](#) object.

```
public string UserId { get; }  
member this.UserId : string
```

Returns

[String String](#)

Returns the user ID component of the [SqlCredential](#) object..

See

[ADO.NET Overview](#)

Also

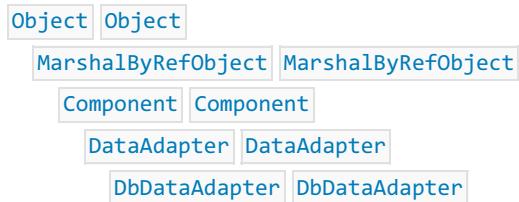
# SqlDataAdapter SqlDataAdapter Class

Represents a set of data commands and a database connection that are used to fill the [DataSet](#) and update a SQL Server database. This class cannot be inherited.

## Declaration

```
public sealed class SqlDataAdapter : System.Data.Common.DbDataAdapter, ICloneable  
  
type SqlDataAdapter = class  
    inherit DbDataAdapter  
    interface IDbDataAdapter  
    interface IDataAdapter  
    interface ICloneable
```

## Inheritance Hierarchy



## Remarks

The [SqlDataAdapter](#), serves as a bridge between a [DataSet](#) and SQL Server for retrieving and saving data. The [SqlDataAdapter](#) provides this bridge by mapping [Fill](#), which changes the data in the [DataSet](#) to match the data in the data source, and [Update](#), which changes the data in the data source to match the data in the [DataSet](#), using the appropriate Transact-SQL statements against the data source. The update is performed on a by-row basis. For every inserted, modified, and deleted row, the [Update](#) method determines the type of change that has been performed on it ([Insert](#), [Update](#), or [Delete](#)). Depending on the type of change, the [Insert](#), [Update](#), or [Delete](#) command template executes to propagate the modified row to the data source. When the [SqlDataAdapter](#) fills a [DataSet](#), it creates the necessary tables and columns for the returned data if they do not already exist. However, primary key information is not included in the implicitly created schema unless the [MissingSchemaAction](#) property is set to [AddWithKey](#). You may also have the [SqlDataAdapter](#) create the schema of the [DataSet](#), including primary key information, before filling it with data using [FillSchema](#). For more information, see [Adding Existing Constraints to a DataSet](#).

[SqlDataAdapter](#) is used in conjunction with [SqlConnection](#) and [SqlCommand](#) to increase performance when connecting to a SQL Server database.

### Note

If you are using SQL Server stored procedures to edit or delete data using a [DataAdapter](#), make sure that you do not use SET NOCOUNT ON in the stored procedure definition. This causes the rows affected count returned to be zero, which the [DataAdapter](#) interprets as a concurrency conflict. In this event, a [DBConcurrencyException](#) will be thrown.

The [SqlDataAdapter](#) also includes the [SelectCommand](#), [InsertCommand](#), [DeleteCommand](#), [UpdateCommand](#), and [TableMappings](#) properties to facilitate the loading and updating of data.

When an instance of [SqlDataAdapter](#) is created, the read/write properties are set to initial values. For a list of these values, see the [SqlDataAdapter](#) constructor.

The [InsertCommand](#), [DeleteCommand](#), and [UpdateCommand](#) are generic templates that are automatically filled with individual values from every modified row through the parameters mechanism.

For every column that you propagate to the data source on [Update](#), a parameter should be added to the

`InsertCommand`, `UpdateCommand`, or `DeleteCommand`. The `SourceColumn` property of the `DbParameter` object should be set to the name of the column. This setting indicates that the value of the parameter is not set manually, but is taken from the particular column in the currently processed row.

**Note**

An `InvalidOperationException` will occur if the `Fill` method is called and the table contains a user-defined type that is not available on the client computer. For more information, see [CLR User-Defined Types](#).

## Constructors

`SqlDataAdapter()`

`SqlDataAdapter()`

Initializes a new instance of the `SqlDataAdapter` class.

`SqlDataAdapter(SqlCommand)`

`SqlDataAdapter(SqlCommand)`

Initializes a new instance of the `SqlDataAdapter` class with the specified `SqlCommand` as the `SelectCommand` property.

`SqlDataAdapter(String, SqlConnection)`

`SqlDataAdapter(String, SqlConnection)`

Initializes a new instance of the `SqlDataAdapter` class with a `SelectCommand` and a `SqlConnection` object.

`SqlDataAdapter(String, String)`

`SqlDataAdapter(String, String)`

Initializes a new instance of the `SqlDataAdapter` class with a `SelectCommand` and a connection string.

## Properties

`DeleteCommand`

`DeleteCommand`

Gets or sets a Transact-SQL statement or stored procedure to delete records from the data set.

`InsertCommand`

`InsertCommand`

Gets or sets a Transact-SQL statement or stored procedure to insert new records into the data source.

`SelectCommand`

`SelectCommand`

Gets or sets a Transact-SQL statement or stored procedure used to select records in the data source.

[UpdateBatchSize](#)

[UpdateBatchSize](#)

Gets or sets the number of rows that are processed in each round-trip to the server.

[UpdateCommand](#)

[UpdateCommand](#)

Gets or sets a Transact-SQL statement or stored procedure used to update records in the data source.

## Events

[RowUpdated](#)

[RowUpdated](#)

Occurs during [Update\(DataSet\)](#) after a command is executed against the data source. The attempt to update is made, so the event fires.

[RowUpdating](#)

[RowUpdating](#)

Occurs during [Update\(DataSet\)](#) before a command is executed against the data source. The attempt to update is made, so the event fires.

[IDbDataAdapter.DeleteCommand](#)

[IDbDataAdapter.DeleteCommand](#)

For a description of this member, see [DeleteCommand](#).

[IDbDataAdapter.InsertCommand](#)

[IDbDataAdapter.InsertCommand](#)

For a description of this member, see [InsertCommand](#).

[IDbDataAdapter.SelectCommand](#)

[IDbDataAdapter.SelectCommand](#)

For a description of this member, see [SelectCommand](#).

[IDbDataAdapter.UpdateCommand](#)

[IDbDataAdapter.UpdateCommand](#)

For a description of this member, see [UpdateCommand](#).

[ICloneable.Clone\(\)](#)

`ICloneable.Clone()`

For a description of this member, see [Clone\(\)](#).

## See Also

# SqlDataAdapter.DeleteCommand SqlDataAdapter.DeleteCommand

## In this Article

Gets or sets a Transact-SQL statement or stored procedure to delete records from the data set.

```
[System.Data.DataSysDescription("DbDataAdapter_DeleteCommand")]
public System.Data.SqlClient.SqlCommand DeleteCommand { get; set; }

member this.DeleteCommand : System.Data.SqlClient.SqlCommand with get, set
```

## Returns

[SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) used during [Update\(DataSet\)](#) to delete records in the database that correspond to deleted rows in the [DataSet](#).

## Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlDataAdapter](#) and sets the [SelectCommand](#), [InsertCommand](#), [UpdateCommand](#), and [DeleteCommand](#) properties. It assumes you have already created a [SqlConnection](#) object.

```

public static SqlDataAdapter CreateCustomerAdapter(
    SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter();

    // Create the SelectCommand.
    SqlCommand command = new SqlCommand("SELECT * FROM Customers " +
        "WHERE Country = @Country AND City = @City", connection);

    // Add the parameters for the SelectCommand.
    command.Parameters.Add("@Country", SqlDbType.NVarChar, 15);
    command.Parameters.Add("@City", SqlDbType.NVarChar, 15);

    adapter.SelectCommand = command;

    // Create the InsertCommand.
    command = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);

    // Add the parameters for the InsertCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");

    adapter.InsertCommand = command;

    // Create the UpdateCommand.
    command = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);

    // Add the parameters for the UpdateCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");
    SqlParameter parameter = command.Parameters.Add(
        "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.UpdateCommand = command;

    // Create the DeleteCommand.
    command = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Add the parameters for the DeleteCommand.
    parameter = command.Parameters.Add(
        "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand = command;

    return adapter;
}

```

## Remarks

During [Update](#), if this property is not set and primary key information is present in the [DataSet](#), the [DeleteCommand](#) can be generated automatically if you set the [SelectCommand](#) property and use the [SqlCommandBuilder](#). Then, any additional commands that you do not set are generated by the [SqlCommandBuilder](#). This generation logic requires key column information to be present in the [DataSet](#). For more information, see [Generating Commands with CommandBuilders](#).

When `DeleteCommand` is assigned to a previously created `SqlCommand`, the `SqlCommand` is not cloned. The `DeleteCommand` maintains a reference to the previously created `SqlCommand` object.

For every column that you propagate to the data source on `Update`, a parameter should be added to the `InsertCommand`, `UpdateCommand`, or `DeleteCommand`. The `SourceColumn` property of the parameter should be set to the name of the column. This indicates that the value of the parameter is not set manually, but is taken from the particular column in the currently processed row.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlDataAdapter.ICloneable.Clone

## In this Article

For a description of this member, see [Clone\(\)](#).

```
object ICloneable.Clone();
```

Returns

[Object](#)

A new object that is a copy of the current instance.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlDataAdapter](#) instance is cast to an [ICloneable](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlDataAdapter.IDbDataAdapter.DeleteCommand

## In this Article

For a description of this member, see [DeleteCommand](#).

```
System.Data.IDbCommand System.Data.IDbDataAdapter.DeleteCommand { get; set; }
```

Returns

[IDbCommand](#)

An [IDbCommand](#) that is used during [Update\(DataSet\)](#) to delete records in the data source for deleted rows in the data set.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlDataAdapter](#) instance is cast to an [IDbDataAdapter](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlDataAdapter.IDbDataAdapter.InsertCommand

## In this Article

For a description of this member, see [InsertCommand](#).

```
System.Data.IDbCommand System.Data.IDbDataAdapter.InsertCommand { get; set; }
```

Returns

[IDbCommand](#)

An [IDbCommand](#) that is used during [Update\(DataSet\)](#) to insert records in the data source for new rows in the data set.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlDataAdapter](#) instance is cast to an [IDbDataAdapter](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlDataAdapter.IDbDataAdapter.SelectCommand

## In this Article

For a description of this member, see [SelectCommand](#).

```
System.Data.IDbCommand System.Data.IDbDataAdapter.SelectCommand { get; set; }
```

Returns

[IDbCommand](#)

An [IDbCommand](#) that is used during [Update\(DataSet\)](#) to select records from data source for placement in the data set.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlDataAdapter](#) instance is cast to an [IDbDataAdapter](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlDataAdapter.IDbDataAdapter.UpdateCommand

## In this Article

For a description of this member, see [UpdateCommand](#).

```
System.Data.IDbCommand System.Data.IDbDataAdapter.UpdateCommand { get; set; }
```

Returns

[IDbCommand](#)

An [IDbCommand](#) that is used during [Update\(DataSet\)](#) to update records in the data source for modified rows in the data set.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlDataAdapter](#) instance is cast to an [IDbDataAdapter](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlDataAdapter.InsertCommand SqlDataAdapter.InsertCommand

## In this Article

Gets or sets a Transact-SQL statement or stored procedure to insert new records into the data source.

```
[System.Data.DataSysDescription("DbDataAdapter_InsertCommand")]
public System.Data.SqlClient.SqlCommand InsertCommand { get; set; }

member this.InsertCommand : System.Data.SqlClient.SqlCommand with get, set
```

## Returns

[SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) used during [Update\(DataSet\)](#) to insert records into the database that correspond to new rows in the [DataSet](#).

## Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlDataAdapter](#) and sets the [SelectCommand](#), [InsertCommand](#), [UpdateCommand](#), and [DeleteCommand](#) properties. It assumes you have already created a [SqlConnection](#) object.

```

public static SqlDataAdapter CreateCustomerAdapter(
    SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter();

    // Create the SelectCommand.
    SqlCommand command = new SqlCommand("SELECT * FROM Customers " +
        "WHERE Country = @Country AND City = @City", connection);

    // Add the parameters for the SelectCommand.
    command.Parameters.Add("@Country", SqlDbType.NVarChar, 15);
    command.Parameters.Add("@City", SqlDbType.NVarChar, 15);

    adapter.SelectCommand = command;

    // Create the InsertCommand.
    command = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);

    // Add the parameters for the InsertCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");

    adapter.InsertCommand = command;

    // Create the UpdateCommand.
    command = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);

    // Add the parameters for the UpdateCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");
    SqlParameter parameter = command.Parameters.Add(
        "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.UpdateCommand = command;

    // Create the DeleteCommand.
    command = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Add the parameters for the DeleteCommand.
    parameter = command.Parameters.Add(
        "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand = command;

    return adapter;
}

```

## Remarks

During [Update](#), if this property is not set and primary key information is present in the [DataSet](#), the [InsertCommand](#) can be generated automatically if you set the [SelectCommand](#) property and use the [SqlCommandBuilder](#). Then, any additional commands that you do not set are generated by the [SqlCommandBuilder](#). This generation logic requires key column information to be present in the [DataSet](#). For more information, see [Generating Commands with CommandBuilders](#).

When `InsertCommand` is assigned to a previously created `SqlCommand`, the `SqlCommand` is not cloned. The `InsertCommand` maintains a reference to the previously created `SqlCommand` object.

If execution of this command returns rows, these rows can be added to the `DataSet` depending on how you set the `UpdatedRowSource` property of the `SqlCommand` object.

For every column that you propagate to the data source on `Update`, a parameter should be added to `InsertCommand`, `UpdateCommand`, or `DeleteCommand`. The `SourceColumn` property of the parameter should be set to the name of the column. This indicates that the value of the parameter is not set manually, but is taken from the particular column in the currently processed row.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlDataAdapter.RowUpdated SqlDataAdapter.RowUpdated

## In this Article

Occurs during [Update\(DataSet\)](#) after a command is executed against the data source. The attempt to update is made, so the event fires.

```
[System.Data.DataSysDescription("DbDataAdapter_RowUpdated")]
public event System.Data.SqlClient.SqlRowUpdatedEventHandler RowUpdated;
member this.RowUpdated : System.Data.SqlClient.SqlRowUpdatedEventHandler
```

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example shows how to use both the [RowUpdating](#) and [RowUpdated](#) events.

The [RowUpdating](#) event returns this output:

```
event args: (command=System.Data.SqlClient.SqlCommand commandType=2 status=0)
```

The [RowUpdated](#) event returns this output:

```
event args: (command=System.Data.SqlClient.SqlCommand commandType=2 recordsAffected=1
row=System.DataDataRow[37] status=0)
```

```

// handler for RowUpdating event
private static void OnRowUpdating(object sender, SqlRowUpdatingEventArgs e)
{
    PrintEventArgs(e);
}

// handler for RowUpdated event
private static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs e)
{
    PrintEventArgs(e);
}

public static int Main()
{
    const string connectionString =
        "Integrated Security=SSPI;database=Northwind;server=MSSQL1";
    const string queryString = "SELECT * FROM Products";

    // create DataAdapter
    SqlDataAdapter adapter = new SqlDataAdapter(queryString, connectionString);
    SqlCommandBuilder builder = new SqlCommandBuilder(adapter);

    // Create and fill DataSet (select only first 5 rows)
    DataSet dataSet = new DataSet();
    adapter.Fill(dataSet, 0, 5, "Table");

    // Modify DataSet
    DataTable table = dataSet.Tables["Table"];
    table.Rows[0][1] = "new product";

    // add handlers
    adapter.RowUpdating += new SqlRowUpdatingEventHandler( OnRowUpdating );
    adapter.RowUpdated += new SqlRowUpdatedEventHandler( OnRowUpdated );

    // update, this operation fires two events
    // (RowUpdating/RowUpdated) per changed row
    adapter.Update(dataSet, "Table");

    // remove handlers
    adapter.RowUpdating -= new SqlRowUpdatingEventHandler( OnRowUpdating );
    adapter.RowUpdated -= new SqlRowUpdatedEventHandler( OnRowUpdated );
    return 0;
}

private static void PrintEventArgs(SqlRowUpdatingEventArgs args)
{
    Console.WriteLine("OnRowUpdating");
    Console.WriteLine("  event args: (" +
        " command=" + args.Command +
        " commandType=" + args.StatementType +
        " status=" + args.Status + ")");
}

private static void PrintEventArgs(SqlRowUpdatedEventArgs args)
{
    Console.WriteLine("OnRowUpdated");
    Console.WriteLine("  event args: (" +
        " command=" + args.Command +
        " commandType=" + args.StatementType +
        " recordsAffected=" + args.RecordsAffected +
        " status=" + args.Status + ")");
}

```

## Remarks

When using [Update](#), there are two events that occur per data row updated. The order of execution is as follows:

1. The values in the [DataRow](#) are moved to the parameter values.
2. The [OnRowUpdating](#) event is raised.
3. The command executes.
4. If the command is set to [FirstReturnedRecord](#), the first returned result is placed in the [DataRow](#).
5. If there are output parameters, they are placed in the [DataRow](#).
6. The [OnRowUpdated](#) event is raised.
7. [AcceptChanges](#) is called.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlDataAdapter.RowUpdating SqlDataAdapter.RowUpdating

## In this Article

Occurs during [Update\(DataSet\)](#) before a command is executed against the data source. The attempt to update is made, so the event fires.

```
[System.Data.DataSysDescription("DbDataAdapter_RowUpdating")]
public event System.Data.SqlClient.SqlRowUpdatingEventHandler RowUpdating;
member this.RowUpdating : System.Data.SqlClient.SqlRowUpdatingEventHandler
```

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example shows how to use both the [RowUpdating](#) and [RowUpdated](#) events.

The [RowUpdating](#) event returns this output:

```
event args: (command=System.Data.SqlClient.SqlCommand commandType=2 status=0)
```

The [RowUpdated](#) event returns this output:

```
event args: (command=System.Data.SqlClient.SqlCommand commandType=2 recordsAffected=1
row=System.DataDataRow[37] status=0)
```

```

// handler for RowUpdating event
private static void OnRowUpdating(object sender, SqlRowUpdatingEventArgs e)
{
    PrintEventArgs(e);
}

// handler for RowUpdated event
private static void OnRowUpdated(object sender, SqlRowUpdatedEventArgs e)
{
    PrintEventArgs(e);
}

public static int Main()
{
    const string connectionString =
        "Integrated Security=SSPI;database=Northwind;server=MSSQL1";
    const string queryString = "SELECT * FROM Products";

    // create DataAdapter
    SqlDataAdapter adapter = new SqlDataAdapter(queryString, connectionString);
    SqlCommandBuilder builder = new SqlCommandBuilder(adapter);

    // Create and fill DataSet (select only first 5 rows)
    DataSet dataSet = new DataSet();
    adapter.Fill(dataSet, 0, 5, "Table");

    // Modify DataSet
    DataTable table = dataSet.Tables["Table"];
    table.Rows[0][1] = "new product";

    // add handlers
    adapter.RowUpdating += new SqlRowUpdatingEventHandler( OnRowUpdating );
    adapter.RowUpdated += new SqlRowUpdatedEventHandler( OnRowUpdated );

    // update, this operation fires two events
    // (RowUpdating/RowUpdated) per changed row
    adapter.Update(dataSet, "Table");

    // remove handlers
    adapter.RowUpdating -= new SqlRowUpdatingEventHandler( OnRowUpdating );
    adapter.RowUpdated -= new SqlRowUpdatedEventHandler( OnRowUpdated );
    return 0;
}

private static void PrintEventArgs(SqlRowUpdatingEventArgs args)
{
    Console.WriteLine("OnRowUpdating");
    Console.WriteLine("  event args: (" +
        " command=" + args.Command +
        " commandType=" + args.StatementType +
        " status=" + args.Status + ")");
}

private static void PrintEventArgs(SqlRowUpdatedEventArgs args)
{
    Console.WriteLine("OnRowUpdated");
    Console.WriteLine("  event args: (" +
        " command=" + args.Command +
        " commandType=" + args.StatementType +
        " recordsAffected=" + args.RecordsAffected +
        " status=" + args.Status + ")");
}

```

## Remarks

When using [Update](#), there are two events that occur per data row updated. The order of execution is as follows:

1. The values in the [DataRow](#) are moved to the parameter values.
2. The [OnRowUpdating](#) event is raised.
3. The command executes.
4. If the command is set to [FirstReturnedRecord](#), the first returned result is placed in the [DataRow](#).
5. If there are output parameters, they are placed in the [DataRow](#).
6. The [OnRowUpdated](#) event is raised.
7. [AcceptChanges](#) is called.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlDataAdapter.SelectCommand SqlDataAdapter.SelectCommand

## In this Article

Gets or sets a Transact-SQL statement or stored procedure used to select records in the data source.

```
[System.Data.DataSysDescription("DbDataAdapter_SelectCommand")]
public System.Data.SqlClient.SqlCommand SelectCommand { get; set; }

member this.SelectCommand : System.Data.SqlClient.SqlCommand with get, set
```

## Returns

[SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) used during [Fill\(DataSet\)](#) to select records from the database for placement in the [DataSet](#).

## Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlDataAdapter](#) and sets the [SelectCommand](#), [InsertCommand](#), [UpdateCommand](#), and [DeleteCommand](#) properties. It assumes you have already created a [SqlConnection](#) object.

```

public static SqlDataAdapter CreateCustomerAdapter(
    SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter();

    // Create the SelectCommand.
    SqlCommand command = new SqlCommand("SELECT * FROM Customers " +
        "WHERE Country = @Country AND City = @City", connection);

    // Add the parameters for the SelectCommand.
    command.Parameters.Add("@Country", SqlDbType.NVarChar, 15);
    command.Parameters.Add("@City", SqlDbType.NVarChar, 15);

    adapter.SelectCommand = command;

    // Create the InsertCommand.
    command = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);

    // Add the parameters for the InsertCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");

    adapter.InsertCommand = command;

    // Create the UpdateCommand.
    command = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);

    // Add the parameters for the UpdateCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");
    SqlParameter parameter = command.Parameters.Add(
        "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.UpdateCommand = command;

    // Create the DeleteCommand.
    command = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Add the parameters for the DeleteCommand.
    parameter = command.Parameters.Add(
        "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand = command;

    return adapter;
}

```

## Remarks

When [SelectCommand](#) is assigned to a previously created [SqlCommand](#), the [SqlCommand](#) is not cloned. The [SelectCommand](#) maintains a reference to the previously created [SqlCommand](#) object.

If the [SelectCommand](#) does not return any rows, no tables are added to the [DataSet](#), and no exception is raised.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)



# SqlDataAdapter SqlDataAdapter

In this Article

## Overloads

|  |   |
|--|---|
| <a href="#">SqlDataAdapter()</a>   | Initializes a new instance of the <a href="#">SqlDataAdapter</a> class.   |
| <a href="#">SqlDataAdapter(SqlCommand)</a> <a href="#">SqlDataAdapter(SqlCommand, SqlCommand)</a>    | Initializes a new instance of the <a href="#">SqlDataAdapter</a> class with the specified <a href="#">SqlCommand</a> as the <a href="#">SelectCommand</a> property. |
| <a href="#">SqlDataAdapter(String, SqlConnection)</a> <a href="#">SqlDataAdapter(String, String)</a> | Initializes a new instance of the <a href="#">SqlDataAdapter</a> class with a <a href="#">SelectCommand</a> and a <a href="#">SqlConnection</a> object.             |
| <a href="#">SqlDataAdapter(String, String)</a>   | Initializes a new instance of the <a href="#">SqlDataAdapter</a> class with a <a href="#">SelectCommand</a> and a connection string.                                |

## SqlDataAdapter()

Initializes a new instance of the [SqlDataAdapter](#) class.

```
public SqlDataAdapter ();
```

## Examples

The following example creates a [SqlDataAdapter](#) and sets some of its properties.

```

public static SqlDataAdapter CreateSqlDataAdapter(SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter();
    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

    // Create the commands.
    adapter.SelectCommand = new SqlCommand(
        "SELECT CustomerID, CompanyName FROM CUSTOMERS", connection);
    adapter.InsertCommand = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);
    adapter.UpdateCommand = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);
    adapter.DeleteCommand = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Create the parameters.
    adapter.InsertCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.InsertCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");

    adapter.UpdateCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.UpdateCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");
    adapter.UpdateCommand.Parameters.Add("@oldCustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion =
        DataRowVersion.Original;

    adapter.DeleteCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion =
        DataRowVersion.Original;

    return adapter;
}

```

## Remarks

When an instance of [SqlDataAdapter](#) is created, the following read/write properties are set to the following initial values.

| Properties           | Initial Value                    |
|----------------------|----------------------------------|
| MissingMappingAction | MissingMappingAction.Passthrough |
| MissingSchemaAction  | MissingSchemaAction.Add          |

You can change the value of any of these properties through a separate call to the property.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## SqlDataAdapter(SqlCommand) SqlDataAdapter(SqlCommand)

Initializes a new instance of the [SqlDataAdapter](#) class with the specified [SqlCommand](#) as the [SelectCommand](#) property.

```
public SqlDataAdapter (System.Data.SqlClient.SqlCommand selectCommand);  
new System.Data.SqlClient.SqlDataAdapter : System.Data.SqlClient.SqlCommand ->  
System.Data.SqlClient.SqlDataAdapter
```

## Parameters

selectCommand [SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) that is a Transact-SQL SELECT statement or stored procedure and is set as the [SelectCommand](#) property of the [SqlDataAdapter](#).

## Examples

The following example creates a [SqlDataAdapter](#) and sets some of its properties.

```
public static SqlDataAdapter CreateSqlDataAdapter(SqlCommand selectCommand,  
    SqlConnection connection)  
{  
    SqlDataAdapter adapter = new SqlDataAdapter(selectCommand);  
    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;  
  
    // Create the other commands.  
    adapter.InsertCommand = new SqlCommand(  
        "INSERT INTO Customers (CustomerID, CompanyName) " +  
        "VALUES (@CustomerID, @CompanyName)", connection);  
  
    adapter.UpdateCommand = new SqlCommand(  
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +  
        "WHERE CustomerID = @oldCustomerID", connection);  
  
    adapter.DeleteCommand = new SqlCommand(  
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);  
  
    // Create the parameters.  
    adapter.InsertCommand.Parameters.Add("@CustomerID",  
        SqlDbType.Char, 5, "CustomerID");  
    adapter.InsertCommand.Parameters.Add("@CompanyName",  
        SqlDbType.VarChar, 40, "CompanyName");  
  
    adapter.UpdateCommand.Parameters.Add("@CustomerID",  
        SqlDbType.Char, 5, "CustomerID");  
    adapter.UpdateCommand.Parameters.Add("@CompanyName",  
        SqlDbType.VarChar, 40, "CompanyName");  
    adapter.UpdateCommand.Parameters.Add("@oldCustomerID",  
        SqlDbType.Char, 5, "CustomerID").SourceVersion = DataRowVersion.Original;  
  
    adapter.DeleteCommand.Parameters.Add("@CustomerID",  
        SqlDbType.Char, 5, "CustomerID").SourceVersion = DataRowVersion.Original;  
  
    return adapter;  
}
```

## Remarks

This implementation of the [SqlDataAdapter](#) constructor sets the [SelectCommand](#) property to the value specified in the [selectCommand](#) parameter.

When an instance of [SqlDataAdapter](#) is created, the following read/write properties are set to the following initial values.

| Properties           | Initial Value                    |
|----------------------|----------------------------------|
| MissingMappingAction | MissingMappingAction.Passthrough |
| MissingSchemaAction  | MissingSchemaAction.Add          |

You can change the value of any of these properties through a separate call to the property.

When [SelectCommand](#) (or any of the other command properties) is assigned to a previously created [SqlCommand](#), the [SqlCommand](#) is not cloned. The [SelectCommand](#) maintains a reference to the previously created [SqlCommand](#) object.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## **SqlDataAdapter(String, SqlConnection) SqlDataAdapter(String, SqlConnection)**

Initializes a new instance of the [SqlDataAdapter](#) class with a [SelectCommand](#) and a [SqlConnection](#) object.

```
public SqlDataAdapter (string selectCommandText, System.Data.SqlClient.SqlConnection
selectConnection);

new System.Data.SqlClient.SqlDataAdapter : string * System.Data.SqlClient.SqlConnection ->
System.Data.SqlClient.SqlDataAdapter
```

Parameters

selectCommandText

[String](#)

A [String](#) that is a Transact-SQL SELECT statement or stored procedure to be used by the [SelectCommand](#) property of the [SqlDataAdapter](#).

selectConnection

[SqlConnection](#)

A [SqlConnection](#) that represents the connection. If your connection string does not use `Integrated Security = true`, you can use [SqlCredential](#) to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

### Examples

The following example creates a [SqlDataAdapter](#) and sets some of its properties.

```

public static SqlDataAdapter CreateSqlDataAdapter(string commandText,
    SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter(commandText, connection);

    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

    // Create the other commands.
    adapter.InsertCommand = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)");

    adapter.UpdateCommand = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID");

    adapter.DeleteCommand = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID");

    // Create the parameters.
    adapter.InsertCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.InsertCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");

    adapter.UpdateCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.UpdateCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");
    adapter.UpdateCommand.Parameters.Add("@oldCustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion = DataRowVersion.Original;

    return adapter;
}

```

## Remarks

This implementation of the [SqlDataAdapter](#) opens and closes a [SqlConnection](#) if it is not already open. This can be useful in an application that must call the [Fill](#) method for two or more [SqlDataAdapter](#) objects. If the [SqlConnection](#) is already open, you must explicitly call **Close** or **Dispose** to close it.

When an instance of [SqlDataAdapter](#) is created, the following read/write properties are set to the following initial values.

| PROPERTIES                           | INITIAL VALUE                                 |
|--------------------------------------|---|
| <a href="#">MissingMappingAction</a> | <code>MissingMappingAction.Passthrough</code> |
| <a href="#">MissingSchemaAction</a>  | <code>MissingSchemaAction.Add</code>          |

You can change the value of either of these properties through a separate call to the property.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## **SqlDataAdapter(String, String) SqlDataAdapter(String, String)**

Initializes a new instance of the [SqlDataAdapter](#) class with a [SelectCommand](#) and a connection string.

```
public SqlDataAdapter (string selectCommandText, string selectConnectionString);
new System.Data.SqlClient.SqlDataAdapter : string * string -> System.Data.SqlClient.SqlDataAdapter
```

## Parameters

selectCommandText String String

A [String](#) that is a Transact-SQL SELECT statement or stored procedure to be used by the [SelectCommand](#) property of the [SqlDataAdapter](#).

selectConnectionString String String

The connection string. If your connection string does not use `Integrated Security = true`, you can use [SqlDataAdapter\(String, SqlConnection\)](#) and [SqlCredential](#) to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

## Examples

The following example creates a [SqlDataAdapter](#) and sets some of its properties.

```
public static SqlDataAdapter CreateSqlDataAdapter(string commandText,
    string connectionString)
{
    SqlDataAdapter adapter = new SqlDataAdapter(commandText, connectionString);
    SqlConnection connection = adapter.SelectCommand.Connection;

    adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;

    // Create the commands.
    adapter.InsertCommand = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);

    adapter.UpdateCommand = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);

    adapter.DeleteCommand = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Create the parameters.
    adapter.InsertCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.InsertCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");

    adapter.UpdateCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID");
    adapter.UpdateCommand.Parameters.Add("@CompanyName",
        SqlDbType.VarChar, 40, "CompanyName");
    adapter.UpdateCommand.Parameters.Add("@oldCustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand.Parameters.Add("@CustomerID",
        SqlDbType.Char, 5, "CustomerID").SourceVersion = DataRowVersion.Original;

    return adapter;
}
```

## Remarks

This overload of the [SqlDataAdapter](#) constructor uses the `selectCommandText` parameter to set the [SelectCommand](#) property. The [SqlDataAdapter](#) will create and maintain the connection created with the `selectConnectionString` parameter.

When an instance of [SqlDataAdapter](#) is created, the following read/write properties are set to the following initial values.

| PROPERTIES                           | INITIAL VALUE                                 |
|--------------------------------------|---|
| <a href="#">MissingMappingAction</a> | <code>MissingMappingAction.Passthrough</code> |
| <a href="#">MissingSchemaAction</a>  | <code>MissingSchemaAction.Add</code>          |

You can change the value of any of these properties through a separate call to the property.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlDataAdapter.UpdateBatchSize

## SqlDataAdapter.UpdateBatchSize

### In this Article

Gets or sets the number of rows that are processed in each round-trip to the server.

```
public override int UpdateBatchSize { get; set; }  
member this.UpdateBatchSize : int with get, set
```

### Returns

[Int32](#)

The number of rows to process per-batch.

| VALUE IS | EFFECT  |
|----------|---|
| 0        | There is no limit on the batch size..   |
| 1        | Disables batch updating.  |
| >1       | Changes are sent using batches of <a href="#">UpdateBatchSize</a> operations at a time. |

When setting this to a value other than 1, all the commands associated with the [SqlDataAdapter](#) have to have their **UpdatedRowSource** property set to [None](#) or [OutputParameters](#). An exception is thrown otherwise.

## Remarks

Gets or sets a value that enables or disables batch processing support, and specifies the number of commands that can be executed in a batch.

Use the [UpdateBatchSize](#) property to update a data source with changes from a [DataSet](#). This can increase application performance by reducing the number of round-trips to the server.

Executing an extremely large batch could decrease performance. Therefore, you should test for the optimum batch size setting before implementing your application.

An [ArgumentOutOfRangeException](#) is thrown if the value is set to a number less than zero.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlDataAdapter.UpdateCommand SqlDataAdapter.UpdateCommand

## In this Article

Gets or sets a Transact-SQL statement or stored procedure used to update records in the data source.

```
[System.Data.DataSysDescription("DbDataAdapter_UpdateCommand")]
public System.Data.SqlClient.SqlCommand UpdateCommand { get; set; }

member this.UpdateCommand : System.Data.SqlClient.SqlCommand with get, set
```

Returns

[SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) used during [Update\(DataSet\)](#) to update records in the database that correspond to modified rows in the [DataSet](#).

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlDataAdapter](#) and sets the [SelectCommand](#), [InsertCommand](#), [UpdateCommand](#) and [DeleteCommand](#) properties. It assumes you have already created a [SqlConnection](#) object.

```

public static SqlDataAdapter CreateCustomerAdapter(
    SqlConnection connection)
{
    SqlDataAdapter adapter = new SqlDataAdapter();

    // Create the SelectCommand.
    SqlCommand command = new SqlCommand("SELECT * FROM Customers " +
        "WHERE Country = @Country AND City = @City", connection);

    // Add the parameters for the SelectCommand.
    command.Parameters.Add("@Country", SqlDbType.NVarChar, 15);
    command.Parameters.Add("@City", SqlDbType.NVarChar, 15);

    adapter.SelectCommand = command;

    // Create the InsertCommand.
    command = new SqlCommand(
        "INSERT INTO Customers (CustomerID, CompanyName) " +
        "VALUES (@CustomerID, @CompanyName)", connection);

    // Add the parameters for the InsertCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");

    adapter.InsertCommand = command;

    // Create the UpdateCommand.
    command = new SqlCommand(
        "UPDATE Customers SET CustomerID = @CustomerID, CompanyName = @CompanyName " +
        "WHERE CustomerID = @oldCustomerID", connection);

    // Add the parameters for the UpdateCommand.
    command.Parameters.Add("@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    command.Parameters.Add("@CompanyName", SqlDbType.NVarChar, 40, "CompanyName");
    SqlParameter parameter = command.Parameters.Add(
        "@oldCustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.UpdateCommand = command;

    // Create the DeleteCommand.
    command = new SqlCommand(
        "DELETE FROM Customers WHERE CustomerID = @CustomerID", connection);

    // Add the parameters for the DeleteCommand.
    parameter = command.Parameters.Add(
        "@CustomerID", SqlDbType.NChar, 5, "CustomerID");
    parameter.SourceVersion = DataRowVersion.Original;

    adapter.DeleteCommand = command;

    return adapter;
}

```

## Remarks

During [Update](#), if this property is not set and primary key information is present in the [DataSet](#), the [UpdateCommand](#) can be generated automatically if you set the [SelectCommand](#) property and use the [SqlCommandBuilder](#). Then, any additional commands that you do not set are generated by the [SqlCommandBuilder](#). This generation logic requires key column information to be present in the [DataSet](#). For more information, see [Generating Commands with CommandBuilders](#).

When [UpdateCommand](#) is assigned to a previously created [SqlCommand](#), the [SqlCommand](#) is not cloned. The [UpdateCommand](#) maintains a reference to the previously created [SqlCommand](#) object.

**Note**

If execution of this command returns rows, the updated rows may be merged with the [DataSet](#) depending on how you set the **UpdatedRowSource** property of the [SqlCommand](#) object.

For every column that you propagate to the data source on [Update](#), a parameter should be added to [InsertCommand](#), [UpdateCommand](#), or [DeleteCommand](#).

The [SourceColumn](#) property of the parameter should be set to the name of the column. This indicates that the value of the parameter is not set manually, but taken from the particular column in the currently processed row.

See

[Manipulating Data \(ADO.NET\)](#)

Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlDataReader SqlDataReader Class

Provides a way of reading a forward-only stream of rows from a SQL Server database. This class cannot be inherited.

## Declaration

```
public class SqlDataReader : System.Data.Common.DbDataReader, IDisposable  
type SqlDataReader = class  
    inherit DbDataReader  
    interface IDataReader  
    interface IDisposable  
    interface IDataRecord
```

## Inheritance Hierarchy



## Remarks

To create a `SqlDataReader`, you must call the `ExecuteReader` method of the `SqlCommand` object, instead of directly using a constructor.

While the `SqlDataReader` is being used, the associated `SqlConnection` is busy serving the `SqlDataReader`, and no other operations can be performed on the `SqlConnection` other than closing it. This is the case until the `Close` method of the `SqlDataReader` is called. For example, you cannot retrieve output parameters until after you call `Close`.

Changes made to a result set by another process or thread while data is being read may be visible to the user of the `SqlDataReader`. However, the precise behavior is timing dependent.

`IsClosed` and `RecordsAffected` are the only properties that you can call after the `SqlDataReader` is closed. Although the `RecordsAffected` property may be accessed while the `SqlDataReader` exists, always call `Close` before returning the value of `RecordsAffected` to guarantee an accurate return value.

When using sequential access (`CommandBehavior.SequentialAccess`), an `InvalidOperationException` will be raised if the `SqlDataReader` position is advanced and another read operation is attempted on the previous column.

### Note

For optimal performance, `SqlDataReader` avoids creating unnecessary objects or making unnecessary copies of data. Therefore, multiple calls to methods such as `GetValue` return a reference to the same object. Use caution if you are modifying the underlying value of the objects returned by methods such as `GetValue`.

## Properties

### Connection

#### Connection

Gets the `SqlConnection` associated with the `SqlDataReader`.

### Depth

#### Depth

Gets a value that indicates the depth of nesting for the current row.

**FieldCount**

**FieldCount**

Gets the number of columns in the current row.

**HasRows**

**HasRows**

Gets a value that indicates whether the [SqlDataReader](#) contains one or more rows.

**IsClosed**

**IsClosed**

Retrieves a Boolean value that indicates whether the specified [SqlDataReader](#) instance has been closed.

**Item[String]**

**Item[String]**

Gets the value of the specified column in its native format given the column name.

**Item[Int32]**

**Item[Int32]**

Gets the value of the specified column in its native format given the column ordinal.

**RecordsAffected**

**RecordsAffected**

Gets the number of rows changed, inserted, or deleted by execution of the Transact-SQL statement.

**VisibleFieldCount**

**VisibleFieldCount**

Gets the number of fields in the [SqlDataReader](#) that are not hidden.

## Methods

**Close()**

**Close()**

Closes the [SqlDataReader](#) object.

**GetBoolean(Int32)**

**GetBoolean(Int32)**

Gets the value of the specified column as a Boolean.

```
GetByte(Int32)
```

```
GetByte(Int32)
```

Gets the value of the specified column as a byte.

```
GetBytes(Int32, Int64, Byte[], Int32, Int32)
```

```
GetBytes(Int32, Int64, Byte[], Int32, Int32)
```

Reads a stream of bytes from the specified column offset into the buffer an array starting at the given buffer offset.

```
GetChar(Int32)
```

```
GetChar(Int32)
```

Gets the value of the specified column as a single character.

```
GetChars(Int32, Int64, Char[], Int32, Int32)
```

```
GetChars(Int32, Int64, Char[], Int32, Int32)
```

Reads a stream of characters from the specified column offset into the buffer as an array starting at the given buffer offset.

```
GetColumnSchema()
```

```
GetColumnSchema()
```

```
GetData(Int32)
```

```
GetData(Int32)
```

```
GetDataTypeName(Int32)
```

```
GetDataTypeName(Int32)
```

Gets a string representing the data type of the specified column.

```
GetDateTime(Int32)
```

```
GetDateTime(Int32)
```

Gets the value of the specified column as a [DateTime](#) object.

```
GetDateTimeOffset(Int32)
```

```
GetDateTimeOffset(Int32)
```

Retrieves the value of the specified column as a [DateTimeOffset](#) object.

```
GetDecimal(Int32)  
GetDecimal(Int32)
```

Gets the value of the specified column as a [Decimal](#) object.

```
GetDouble(Int32)  
GetDouble(Int32)
```

Gets the value of the specified column as a double-precision floating point number.

```
GetEnumerator()  
GetEnumerator()
```

Returns an [IEnumerator](#) that iterates through the [SqlDataReader](#).

```
GetFieldType(Int32)  
GetFieldType(Int32)
```

Gets the [Type](#) that is the data type of the object.

```
GetFieldValue<T>(Int32)  
GetFieldValue<T>(Int32)
```

Synchronously gets the value of the specified column as a type. [GetFieldValueAsync<T>\(Int32, CancellationToken\)](#) is the asynchronous version of this method.

```
GetFieldValueAsync<T>(Int32, CancellationToken)  
GetFieldValueAsync<T>(Int32, CancellationToken)
```

Asynchronously gets the value of the specified column as a type. [GetFieldValue<T>\(Int32\)](#) is the synchronous version of this method.

```
GetFloat(Int32)  
GetFloat(Int32)
```

Gets the value of the specified column as a single-precision floating point number.

```
GetGuid(Int32)  
GetGuid(Int32)
```

Gets the value of the specified column as a globally unique identifier (GUID).

```
GetInt16(Int32)  
GetInt16(Int32)
```

Gets the value of the specified column as a 16-bit signed integer.

```
GetInt32(Int32)
```

```
GetInt32(Int32)
```

Gets the value of the specified column as a 32-bit signed integer.

```
GetInt64(Int32)
```

```
GetInt64(Int32)
```

Gets the value of the specified column as a 64-bit signed integer.

```
GetName(Int32)
```

```
GetName(Int32)
```

Gets the name of the specified column.

```
GetOrdinal(String)
```

```
GetOrdinal(String)
```

Gets the column ordinal, given the name of the column.

```
GetProviderSpecificFieldType(Int32)
```

```
GetProviderSpecificFieldType(Int32)
```

Gets an `Object` that is a representation of the underlying provider-specific field type.

```
GetProviderSpecificValue(Int32)
```

```
GetProviderSpecificValue(Int32)
```

Gets an `Object` that is a representation of the underlying provider specific value.

```
GetProviderSpecificValues(Object[])
```

```
GetProviderSpecificValues(Object[])
```

Gets an array of objects that are a representation of the underlying provider specific values.

```
GetSchemaTable()
```

```
GetSchemaTable()
```

Returns a `DataTable` that describes the column metadata of the `SqlDataReader`.

```
GetSqlBinary(Int32)
```

```
GetSqlBinary(Int32)
```

Gets the value of the specified column as a [SqlBinary](#).

```
GetSqlBoolean(Int32)  
GetSqlBoolean(Int32)
```

Gets the value of the specified column as a [SqlBoolean](#).

```
GetSqlByte(Int32)  
GetSqlByte(Int32)
```

Gets the value of the specified column as a [SqlByte](#).

```
GetSqlBytes(Int32)  
GetSqlBytes(Int32)
```

Gets the value of the specified column as [SqlBytes](#).

```
GetSqlChars(Int32)  
GetSqlChars(Int32)
```

Gets the value of the specified column as [SqlChars](#).

```
GetSqlDateTime(Int32)  
GetSqlDateTime(Int32)
```

Gets the value of the specified column as a [SqlDateTime](#).

```
GetSqlDecimal(Int32)  
GetSqlDecimal(Int32)
```

Gets the value of the specified column as a [SqlDecimal](#).

```
GetSqlDouble(Int32)  
GetSqlDouble(Int32)
```

Gets the value of the specified column as a [SqlDouble](#).

```
GetSqlGuid(Int32)  
GetSqlGuid(Int32)
```

Gets the value of the specified column as a [SqlGuid](#).

```
GetSqlInt16(Int32)
```

```
GetSqlInt16(Int32)
```

Gets the value of the specified column as a [SqlInt16](#).

```
GetSqlInt32(Int32)
```

```
GetSqlInt32(Int32)
```

Gets the value of the specified column as a [SqlInt32](#).

```
GetSqlInt64(Int32)
```

```
GetSqlInt64(Int32)
```

Gets the value of the specified column as a [SqlInt64](#).

```
GetSqlMoney(Int32)
```

```
GetSqlMoney(Int32)
```

Gets the value of the specified column as a [SqlMoney](#).

```
GetSqlSingle(Int32)
```

```
GetSqlSingle(Int32)
```

Gets the value of the specified column as a [SqlSingle](#).

```
GetSqlString(Int32)
```

```
GetSqlString(Int32)
```

Gets the value of the specified column as a [SqlString](#).

```
GetSqlValue(Int32)
```

```
GetSqlValue(Int32)
```

Returns the data value in the specified column as a SQL Server type.

```
GetSqlValues(Object[])
```

```
GetSqlValues(Object[])
```

Fills an array of [Object](#) that contains the values for all the columns in the record, expressed as SQL Server types.

```
GetSqlXml(Int32)
```

```
GetSqlXml(Int32)
```

Gets the value of the specified column as an XML value.

```
GetStream(Int32)  
GetStream(Int32)
```

Retrieves binary, image, varbinary, UDT, and variant data types as a [Stream](#).

```
GetString(Int32)  
GetString(Int32)
```

Gets the value of the specified column as a string.

```
GetTextReader(Int32)  
GetTextReader(Int32)
```

Retrieves Char, NChar, NText, NVarChar, text, varChar, and Variant data types as a [TextReader](#).

```
GetTimeSpan(Int32)  
GetTimeSpan(Int32)
```

Retrieves the value of the specified column as a [TimeSpan](#) object.

```
GetValue(Int32)  
GetValue(Int32)
```

Gets the value of the specified column in its native format.

```
GetValues(Object[])  
GetValues(Object[])
```

Populates an array of objects with the column values of the current row.

```
GetXmLReader(Int32)  
GetXmLReader(Int32)
```

Retrieves data of type XML as an [XmlReader](#).

```
IsCommandBehavior(CommandBehavior)  
IsCommandBehavior(CommandBehavior)
```

Determines whether the specified [CommandBehavior](#) matches that of the [SqlDataReader](#).

```
IsDBNull(Int32)  
IsDBNull(Int32)
```

Gets a value that indicates whether the column contains non-existent or missing values.

```
IsDBNullAsync(Int32, CancellationToken)
IsDBNullAsync(Int32, CancellationToken)
```

An asynchronous version of [IsDBNull\(Int32\)](#), which gets a value that indicates whether the column contains non-existent or missing values.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
NextResult()
NextResult()
```

Advances the data reader to the next result, when reading the results of batch Transact-SQL statements.

```
NextResultAsync(CancellationToken)
NextResultAsync(CancellationToken)
```

An asynchronous version of [NextResult\(\)](#), which advances the data reader to the next result, when reading the results of batch Transact-SQL statements.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
Read()
Read()
```

Advances the [SqlDataReader](#) to the next record.

```
ReadAsync(CancellationToken)
ReadAsync(CancellationToken)
```

An asynchronous version of [Read\(\)](#), which advances the [SqlDataReader](#) to the next record.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
IEnumerable.GetEnumerator()
IEnumerable.GetEnumerator()
```

```
IDataRecord.GetData(Int32)
IDataRecord.GetData(Int32)
```

Returns an [IDataReader](#) for the specified column ordinal.

```
IDisposable.Dispose()
IDisposable.Dispose()
```

## See Also

# SqlDataReader.Close SqlDataReader.Close

## In this Article

Closes the [SqlDataReader](#) object.

```
public override void Close ();  
override this.Close : unit -> unit
```

## Examples

The following example creates a [SqlConnection](#), a [SqlCommand](#), and a [SqlDataReader](#). The example reads through the data, writing it out to the console window. The code then closes the [SqlDataReader](#). The [SqlConnection](#) is closed automatically at the end of the `using` code block.

```
private static void ReadOrderData(string connectionString)  
{  
    string queryString =  
        "SELECT OrderID, CustomerID FROM dbo.Orders;";  
  
    using (SqlConnection connection =  
        new SqlConnection(connectionString))  
    {  
        connection.Open();  
  
        using (SqlCommand command =  
            new SqlCommand(queryString, connection))  
        {  
            using (SqlDataReader reader = command.ExecuteReader())  
            {  
                // Call Read before accessing data.  
                while (reader.Read())  
                {  
                    Console.WriteLine(String.Format("{0}, {1}",  
                        reader[0], reader[1]));  
                }  
  
                // Call Close when done reading.  
                reader.Close();  
            }  
        }  
    }  
}
```

## Remarks

You must explicitly call the [Close](#) method when you are through using the [SqlDataReader](#) to use the associated [SqlConnection](#) for any other purpose.

The `Close` method fills in the values for output parameters, return values and `RecordsAffected`, increasing the time that it takes to close a [SqlDataReader](#) that was used to process a large or complex query. When the return values and the number of records affected by a query are not significant, the time that it takes to close the [SqlDataReader](#) can be reduced by calling the [Cancel](#) method of the associated [SqlCommand](#) object before calling the `Close` method.

### Caution

Do not call `Close` or `Dispose` on a Connection, a DataReader, or any other managed object in the `Finalize` method of your class. In a finalizer, you should only release unmanaged resources that your class owns directly. If your class does not own any unmanaged resources, do not include a `Finalize` method in your class definition. For more

information, see [Garbage Collection](#).

See

Also

[DataAdapters and DataReaders](#)

[SQL Server and ADO.NET](#)

[SQL Server Connection Pooling \(ADO.NET\)](#)

[ADO.NET Overview](#)

# SqlDataReader.Connection SqlDataReader.Connection

## In this Article

Gets the [SqlConnection](#) associated with the [SqlDataReader](#).

```
protected System.Data.SqlClient.SqlConnection Connection { get; }  
member this.Connection : System.Data.SqlClient.SqlConnection
```

Returns

[SqlConnection](#) [SqlConnection](#)

The [SqlConnection](#) associated with the [SqlDataReader](#).

See

[SQL Server Connection Pooling \(ADO.NET\)](#)

Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.Depth SqlDataReader.Depth

## In this Article

Gets a value that indicates the depth of nesting for the current row.

```
public override int Depth { get; }  
member this.Depth : int
```

Returns

[Int32 Int32](#)

The depth of nesting for the current row.

## Remarks

The outermost table has a depth of zero. The .NET Framework Data Provider for SQL Server does not support nesting and always returns zero.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.FieldCount SqlDataReader.FieldCount

## In this Article

Gets the number of columns in the current row.

```
public override int FieldCount { get; }  
member this.FieldCount : int
```

Returns

[Int32](#) [Int32](#)

When not positioned in a valid recordset, 0; otherwise the number of columns in the current row. The default is -1.

Exceptions

[NotSupportedException](#) [NotSupportedException](#)

There is no current connection to an instance of SQL Server.

## Remarks

Executing a query that, by its nature, does not return rows (such as a DELETE query), sets [FieldCount](#) to 0. However, this should not be confused with a query that returns 0 rows (such as `SELECT * FROM table WHERE 1 = 2`) in which case [FieldCount](#) returns the number of columns in the table, including hidden fields. Use [VisibleFieldCount](#) to exclude hidden fields.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetBoolean SqlDataReader.GetBoolean

## In this Article

Gets the value of the specified column as a Boolean.

```
public override bool GetBoolean (int i);  
override this.GetBoolean : int -> bool
```

### Parameters

i Int32 Int32

The zero-based column ordinal.

### Returns

[Boolean Boolean](#)

The value of the column.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a Boolean, or an exception is generated.

Call [IsDBNull IsDBNull](#) to check for null values before calling this method.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetByte SqlDataReader.GetByte

## In this Article

Gets the value of the specified column as a byte.

```
public override byte GetByte (int i);  
override this.GetByte : int -> byte
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Byte Byte](#)

The value of the specified column as a byte.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a byte.

Call [IsDBNull IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetBytes SqlDataReader.GetBytes

## In this Article

Reads a stream of bytes from the specified column offset into the buffer an array starting at the given buffer offset.

```
public override long GetBytes (int i, long dataIndex, byte[] buffer, int bufferIndex, int length);  
override this.GetBytes : int * int64 * byte[] * int * int -> int64
```

## Parameters

i Int32 Int32

The zero-based column ordinal.

dataIndex Int64 Int64

The index within the field from which to begin the read operation.

buffer Byte[]

The buffer into which to read the stream of bytes.

bufferIndex Int32 Int32

The index within the `buffer` where the write operation is to start.

length Int32 Int32

The maximum length to copy into the buffer.

## Returns

[Int64 Int64](#)

The actual number of bytes read.

## Remarks

[GetBytes](#) returns the number of available bytes in the field. Most of the time this is the exact length of the field. However, the number returned may be less than the true length of the field if `GetBytes` has already been used to obtain bytes from the field. This may be the case, for example, if the [SqlDataReader](#) is reading a large data structure into a buffer. For more information, see the `SequentialAccess` setting for [CommandBehavior](#).

If you pass a buffer that is `null`, [GetBytes](#) returns the length of the entire field in bytes, not the remaining size based on the buffer offset parameter.

No conversions are performed; therefore, the data retrieved must already be a byte array.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetChar SqlDataReader.GetChar

## In this Article

Gets the value of the specified column as a single character.

```
public override char GetChar (int i);  
override this.GetChar : int -> char
```

### Parameters

i [Int32](#) [Int32](#)

The zero-based column ordinal.

### Returns

[Char](#) [Char](#)

The value of the specified column.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The specified cast is not valid.

## Remarks

Not supported for [System.Data.SqlClient](#).

### See

[ADO.NET Overview](#)

### Also

# SqlDataReader.GetChars SqlDataReader.GetChars

## In this Article

Reads a stream of characters from the specified column offset into the buffer as an array starting at the given buffer offset.

```
public override long GetChars (int i, long dataIndex, char[] buffer, int bufferIndex, int length);  
override this.GetChars : int * int64 * char[] * int * int -> int64
```

## Parameters

i Int32 Int32

The zero-based column ordinal.

dataIndex Int64 Int64

The index within the field from which to begin the read operation.

buffer Char[]

The buffer into which to read the stream of bytes.

bufferIndex Int32 Int32

The index within the `buffer` where the write operation is to start.

length Int32 Int32

The maximum length to copy into the buffer.

## Returns

Int64 Int64

The actual number of characters read.

## Remarks

`GetChars` returns the number of available characters in the field. Frequently this is the exact length of the field. However, the number returned may be less than the true length of the field if `GetChars` has already been used to obtain characters from the field. This may be the case, for example, if the `SqlDataReader` is reading a large data structure into a buffer. For more information, see the `SequentialAccess` setting for `CommandBehavior`.

The actual number of characters read can be less than the requested length, if the end of the field is reached. If you pass a buffer that is `null`, `GetChars` returns the length of the entire field in characters, not the remaining size based on the buffer offset parameter.

No conversions are performed; therefore, the data retrieved must already be a character array.

### Note

The `GetChars` method returns 0 when `dataIndex` is negative.

## See

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetColumnSchema SqlDataReader.GetColumnSchema

## In this Article

```
public System.Collections.ObjectModel.ReadOnlyCollection<System.Data.Common.DbColumn>
GetColumnSchema ();

abstract member GetColumnSchema : unit ->
System.Collections.ObjectModel.ReadOnlyCollection<System.Data.Common.DbColumn>
override this.GetColumnSchema : unit ->
System.Collections.ObjectModel.ReadOnlyCollection<System.Data.Common.DbColumn>
```

## Returns

[ReadOnlyCollection<DbColumn>](#)

# SqlDataReader.GetData SqlDataReader.GetData

## In this Article

```
public System.Data.IDataReader GetData (int i);  
  
abstract member GetData : int -> System.Data.IDataReader  
override this.GetData : int -> System.Data.IDataReader
```

## Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

## Returns

IDataReader IDataReader

# SqlDataReader.GetDataTypeName SqlDataReader.Get DataTypeName

## In this Article

Gets a string representing the data type of the specified column.

```
public override string GetDataTypeName (int i);  
override this.GetDataTypeName : int -> string
```

## Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based ordinal position of the column to find.

## Returns

[String String](#)

The string representing the data type of the specified column.

## Remarks

Returns the name of the back-end data type.

`numeric` is a synonym in SQL Server for the `decimal` data type. `GetDataTypeName` will return "decimal" for a column defined as either decimal or numeric.

## See

[ADO.NET Overview](#)

## Also

# SqlDataReader.GetDateTime SqlDataReader.GetDateTime

## In this Article

Gets the value of the specified column as a [DateTime](#) object.

```
public override DateTime GetDateTime (int i);  
override this.GetDateTime : int -> DateTime
```

## Parameters

|   |                       |                       |
|---|-----------------------|-----------------------|
| i | <a href="#">Int32</a> | <a href="#">Int32</a> |
|---|-----------------------|-----------------------|

The zero-based column ordinal.

## Returns

[DateTime](#) [DateTime](#)

The value of the specified column.

## Exceptions

[InvalidCastException](#) [InvalidCastException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a [DateTime](#) object.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetDateTimeOffset SqlDataReader.GetDateTimeOffset

## In this Article

Retrieves the value of the specified column as a [DateTimeOffset](#) object.

```
public virtual DateTimeOffset GetDateTimeOffset (int i);  
  
abstract member GetDateTimeOffset : int -> DateTimeOffset  
override this.GetDateTimeOffset : int -> DateTimeOffset
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[DateTimeOffset](#)

The value of the specified column.

## Exceptions

[InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a [DateTimeOffset](#) object.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[Date and Time Data Types \(Katmai\)](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetDecimal SqlDataReader.GetDecimal

## In this Article

Gets the value of the specified column as a [Decimal](#) object.

```
public override decimal GetDecimal (int i);  
override this.GetDecimal : int -> decimal
```

### Parameters

|   |                       |                       |
|---|-----------------------|-----------------------|
| i | <a href="#">Int32</a> | <a href="#">Int32</a> |
|---|-----------------------|-----------------------|

The zero-based column ordinal.

### Returns

[Decimal](#) [Decimal](#)

The value of the specified column.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a [Decimal](#) object.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetDouble SqlDataReader.GetDouble

## In this Article

Gets the value of the specified column as a double-precision floating point number.

```
public override double GetDouble (int i);  
override this.GetDouble : int -> double
```

### Parameters

i Int32 Int32

The zero-based column ordinal.

### Returns

[Double Double](#)

The value of the specified column.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed. Therefore, the data retrieved must already be a double-precision floating point number.

Call [IsDBNull IsDBNull](#) to check for null values before calling this method.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetEnumerator SqlDataReader.Get Enumerator

## In this Article

Returns an [IEnumerator](#) that iterates through the [SqlDataReader](#).

```
public override System.Collections.IEnumerator GetEnumerator ();  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [SqlDataReader](#).

## Remarks

Although you can use this method to retrieve an explicit enumerator, in languages that support a `foreach` construct, it is simpler to use the looping construct directly in order to iterate through the rows in the data reader.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetFieldType SqlDataReader.GetFieldType

## In this Article

Gets the [Type](#) that is the data type of the object.

```
public override Type GetFieldType (int i);  
override this.GetFieldType : int -> Type
```

### Parameters

|   |                       |                       |
|---|-----------------------|-----------------------|
| i | <a href="#">Int32</a> | <a href="#">Int32</a> |
|---|-----------------------|-----------------------|

The zero-based column ordinal.

### Returns

[Type](#)

The [Type](#) that is the data type of the object. If the type does not exist on the client, in the case of a User-Defined Type (UDT) returned from the database, **GetFieldType** returns null.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetValue

## In this Article

Synchronously gets the value of the specified column as a type. [GetFieldValueAsync<T>\(Int32, CancellationToken\)](#) is the asynchronous version of this method.

```
public override T GetFieldValue<T> (int i);  
override this.GetValue : int -> 'T
```

### Type Parameters

T

The type of the value to be returned.

### Parameters

i Int32 Int32

The column to be retrieved.

### Returns

T T

The returned type object.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The connection drops or is closed during the data retrieval.

The [SqlDataReader](#) is closed during the data retrieval.

There is no data ready to be read (for example, the first [Read\(\)](#) hasn't been called, or returned false).

Tried to read a previously-read column in sequential mode.

There was an asynchronous operation in progress. This applies to all Get\* methods when running in sequential mode, as they could be called while reading a stream.

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

Trying to read a column that does not exist.

[SqlNullValueException](#) [SqlNullValueException](#)

The value of the column was null (`IsDBNull(Int32) == true`), retrieving a non-SQL type.

[InvalidCastException](#) [InvalidCastException](#)

`T` doesn't match the type returned by SQL Server or cannot be cast.

## Remarks

`T` can be one of the following types:

|                |   |             |            |
|----------------|---|-------------|------------|
|                |   |             |            |
| Boolean        | Byte  | Char        | DateTime   |
| DateTimeOffset | Decimal   | Double      | Float      |
| Guid           | Int16   | Int32       | Int64      |
| SqlBoolean     | SqlByte   | SqlDateTime | SqlDecimal |
| SqlDouble      | SqlGuid   | SqlInt16    | SqlInt32   |
| SqlInt64       | SqlMoney  | SqlSingle   | SqlString  |
| String         | UDT, which can be any CLR type marked with <a href="#">SqlUserDefinedTypeAttribute</a> .<br>. |             |            |

For more information, see [SqlClient Streaming Support](#).

# SqlDataReader.GetFieldValueAsync SqlDataReader.GetFieldValueAsync

## In this Article

Asynchronously gets the value of the specified column as a type. [GetFieldValue<T>\(Int32\)](#) is the synchronous version of this method.

```
public override System.Threading.Tasks.Task<T> GetFieldValueAsync<T> (int i,  
System.Threading.CancellationToken cancellationToken);  
  
override this.GetFieldValueAsync : int * System.Threading.CancellationToken ->  
System.Threading.Tasks.Task<'T>
```

## Type Parameters

T

The type of the value to be returned.

## Parameters

|   |       |       |
|---|-------|-------|
| i | Int32 | Int32 |
|---|-------|-------|

The column to be retrieved.

|                   |                   |                   |
|-------------------|-------------------|-------------------|
| cancellationToken | CancellationToken | CancellationToken |
|-------------------|-------------------|-------------------|

The cancellation instruction, which propagates a notification that operations should be canceled. This does not guarantee the cancellation. A setting of [CancellationToken.None](#) makes this method equivalent to [IsDBNull\(Int32\)](#).

The returned task must be marked as cancelled.

## Returns

[Task<T>](#)

The returned type object.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlDataReader](#) is closed during the data retrieval.

The [SqlDataReader](#) is closed during the data retrieval.

There is no data ready to be read (for example, the first [Read\(\)](#) hasn't been called, or returned false).

Tried to read a previously-read column in sequential mode.

There was an asynchronous operation in progress. This applies to all Get\* methods when running in sequential mode, as they could be called while reading a stream.

[Context Connection=true](#) is specified in the connection string.

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

Trying to read a column that does not exist.

[SqlNullValueException](#) [SqlNullValueException](#)

The value of the column was null ([IsDBNull\(Int32\) == true](#)), retrieving a non-SQL type.

## InvalidOperationException InvalidOperationException

T doesn't match the type returned by SQL Server or cannot be cast.

## Remarks

T can be one of the following types:

| Boolean        | Byte  | Char        | DateTime   |
|----------------|---|-------------|------------|
| DateTimeOffset | Decimal   | Double      | Float      |
| Guid           | Int16   | Int32       | Int64      |
| SqlBoolean     | SqlByte   | SqlDateTime | SqlDecimal |
| SqlDouble      | SqlGuid   | SqlInt16    | SqlInt32   |
| SqlInt64       | SqlMoney  | SqlSingle   | SqlString  |
| String         | UDT, which can be any CLR type marked with <a href="#">SqlUserDefinedTypeAttribute</a> .<br>. |             |            |

For more information, see [SqlClient Streaming Support](#).

# SqlDataReader.GetFloat SqlDataReader.GetFloat

## In this Article

Gets the value of the specified column as a single-precision floating point number.

```
public override float GetFloat (int i);  
override this.GetFloat : int -> single
```

### Parameters

i Int32 Int32

The zero-based column ordinal.

### Returns

[Single Single](#)

The value of the specified column.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed. Therefore, the data retrieved must already be a single-precision floating point number.

Call [IsDBNull IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetGuid SqlDataReader.GetGuid

## In this Article

Gets the value of the specified column as a globally unique identifier (GUID).

```
public override Guid GetGuid (int i);  
override this.GetGuid : int -> Guid
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Guid](#) [Guid](#)

The value of the specified column.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a GUID.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetInt16 SqlDataReader.GetInt16

## In this Article

Gets the value of the specified column as a 16-bit signed integer.

```
public override short GetInt16 (int i);  
override this.GetInt16 : int -> int16
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Int16](#) [Int16](#)

The value of the specified column.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a 16-bit signed integer.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetInt32 SqlDataReader.GetInt32

## In this Article

Gets the value of the specified column as a 32-bit signed integer.

```
public override int GetInt32 (int i);  
override this.GetInt32 : int -> int
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Int32](#) [Int32](#)

The value of the specified column.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a 32-bit signed integer.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetInt64 SqlDataReader.GetInt64

## In this Article

Gets the value of the specified column as a 64-bit signed integer.

```
public override long GetInt64 (int i);  
override this.GetInt64 : int -> int64
```

### Parameters

i [Int32](#) [Int32](#)

The zero-based column ordinal.

### Returns

[Int64](#) [Int64](#)

The value of the specified column.

### Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a 64-bit signed integer.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetName SqlDataReader.GetName

## In this Article

Gets the name of the specified column.

```
public override string GetName (int i);  
override this.GetName : int -> string
```

## Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

## Returns

[String String](#)

The name of the specified column.

## See

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetOrdinal SqlDataReader.GetOrdinal

## In this Article

Gets the column ordinal, given the name of the column.

```
public override int GetOrdinal (string name);  
override this.GetOrdinal : string -> int
```

### Parameters

|      |                        |
|------|------------------------|
| name | <a href="#">String</a> |
|------|------------------------|

The name of the column.

### Returns

[Int32](#) [Int32](#)

The zero-based column ordinal.

### Exceptions

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

The name specified is not a valid column name.

## Examples

The following example demonstrates how to use the [GetOrdinal](#) method.

```
private static void ReadGetOrdinal(string connectionString)  
{  
    string queryString = "SELECT DISTINCT CustomerID FROM dbo.Orders;";  
    using (SqlConnection connection =  
        new SqlConnection(connectionString))  
    {  
        SqlCommand command =  
            new SqlCommand(queryString, connection);  
        connection.Open();  
  
        SqlDataReader reader = command.ExecuteReader();  
  
        // Call GetOrdinal and assign value to variable.  
        int customerID = reader.GetOrdinal("CustomerID");  
  
        // Use variable with GetString inside of loop.  
        while (reader.Read())  
        {  
            Console.WriteLine("CustomerID={0}", reader.GetString(customerID));  
        }  
  
        // Call Close when done reading.  
        reader.Close();  
    }  
}
```

## Remarks

[GetOrdinal](#) performs a case-sensitive lookup first. If it fails, a second, case-insensitive search occurs (a case-insensitive comparison is done using the database collation). Unexpected results can occur when comparisons are affected by

culture-specific casing rules. For example, in Turkish, the following example yields the wrong results because the file system in Turkish does not use linguistic casing rules for the letter 'i' in "file". The method throws an `IndexOutOfRangeException` exception if the zero-based column ordinal is not found.

`GetOrdinal` is kana-width insensitive.

Because ordinal-based lookups are more efficient than named lookups, it is inefficient to call `GetOrdinal` within a loop. Save time by calling `GetOrdinal` once and assigning the results to an integer variable for use within the loop.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetProviderSpecificFieldType SqlDataReader.GetProviderSpecificFieldType

## In this Article

Gets an `Object` that is a representation of the underlying provider-specific field type.

```
public override Type GetProviderSpecificFieldType (int i);  
override this.GetProviderSpecificFieldType : int -> Type
```

## Parameters

`i` `Int32` `Int32`

An `Int32` representing the column ordinal.

## Returns

`Type` `Type`

Gets an `Object` that is a representation of the underlying provider-specific field type.

## See

[DataAdapters and DataReaders \(ADO.NET\)](#)

## Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetProviderSpecificValue SqlDataReader.GetProviderSpecificValue

## In this Article

Gets an `Object` that is a representation of the underlying provider specific value.

```
public override object GetProviderSpecificValue (int i);  
override this.GetProviderSpecificValue : int -> obj
```

## Parameters

`i` `Int32` `Int32`

An `Int32` representing the column ordinal.

## Returns

`Object` `Object`

An `Object` that is a representation of the underlying provider specific value.

## See

[Working with SqlDbTypes](#)

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetProviderSpecificValues SqlDataReader.GetProviderSpecificValues

## In this Article

Gets an array of objects that are a representation of the underlying provider specific values.

```
public override int GetProviderSpecificValues (object[] values);  
override this.GetProviderSpecificValues : obj[] -> int
```

## Parameters

values [Object\[\]](#)

An array of [Object](#) into which to copy the column values.

## Returns

[Int32](#) [Int32](#)

The array of objects that are a representation of the underlying provider specific values.

## See

[Working with SqlDbTypes](#)

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSchemaTable SqlDataReader.GetSchemaTable

## In this Article

Returns a [DataTable](#) that describes the column metadata of the [SqlDataReader](#).

```
public override System.Data.DataTable GetSchemaTable ();  
override this.GetSchemaTable : unit -> System.Data.DataTable
```

Returns

[DataTable](#) [DataTable](#)

A [DataTable](#) that describes the column metadata.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlDataReader](#) is closed.

## Remarks

For the [GetSchemaTable](#) method returns metadata about each column in the following order:

| DATAREADER COLUMN | DESCRIPTION  |
|-------------------|--|
| AllowDBNull       | Set if the consumer can set the column to a null value or if the provider cannot determine whether the consumer can set the column to a null value. Otherwise, not set. A column may contain null values, even if it cannot be set to a null value.  |
| BaseCatalogName   | The name of the catalog in the data store that contains the column. NULL if the base catalog name cannot be determined. The default of this column is a null value.  |
| BaseColumnName    | The name of the column in the data store. This might be different than the column name returned in the ColumnName column if an alias was used. A null value if the base column name cannot be determined or if the rowset column is derived, but not identical to, a column in the data store. The default of this column is a null value. |
| BaseSchemaName    | The name of the schema in the data store that contains the column. A null value if the base schema name cannot be determined. The default of this column is a null value.  |
| BaseServerName    | The name of the instance of Microsoft SQL Server used by the <a href="#">SqlDataReader</a> .   |
| BaseTableName     | The name of the table or view in the data store that contains the column. A null value if the base table name cannot be determined. The default of this column is a null value.  |

| DATAREADER COLUMN | DESCRIPTION   |
|-------------------|---|
| ColumnName        | The name of the column; this might not be unique. If this cannot be determined, a null value is returned. This name always reflects the most recent renaming of the column in the current view or command text.   |
| ColumnOrdinal     | The zero-based ordinal of the column. This column cannot contain a null value.  |
| ColumnSize        | The maximum possible length of a value in the column. For columns that use a fixed-length data type, this is the size of the data type. For <code>nvarchar(MAX)</code> , <code>varchar(MAX)</code> , and <code>varbinary(MAX)</code> columns stored in a SQL Server database, the maximum size is 2GB. If these columns are stored and accessed as files, the limit on maximum size is imposed by the file system. This value changes when using the <code>Type System Version</code> keyword in the connection string. For new types they are represented as downlevel types. The MAX data types return the normal 4k for <code>nvarchar</code> and 8000 for <code>varchar</code> . For more information, see the <a href="#">Transact-SQL reference</a> . |
| DataTypeName      | Returns a string representing the data type of the specified column.  |
| IsAliased         | <p><code>true</code>: The column name is an alias.</p> <p><code>false</code>: The column name is not an alias.</p>  |
| IsAutoIncrement   | <p><code>true</code>: The column assigns values to new rows in fixed increments.</p> <p><code>false</code>: The column does not assign values to new rows in fixed increments. The default of this column is <code>false</code>.</p>  |
| IsColumnSet       | <p><code>true</code>: The column is a sparse column that is a member of a column set.</p>   |
| IsExpression      | <p><code>true</code>: The column is an expression.</p> <p><code>false</code>: The column is not an expression.</p>  |
| IsHidden          | <p><code>true</code>: The column is hidden.</p> <p><code>false</code>: The column is not hidden.</p>  |
| IsIdentity        | <p><code>true</code>: The column is an identity column.</p> <p><code>false</code>: The column is not an identity column.</p>  |

| DATAREADER COLUMN        | DESCRIPTION   |
|--------------------------|---|
| IsKey                    | <p><code>true</code>: The column is one of a set of columns in the rowset that, taken together, uniquely identify the row. The set of columns with <code>IsKey</code> set to <code>true</code> must uniquely identify a row in the rowset. There is no requirement that this set of columns is a minimal set of columns. This set of columns may be generated from a base table primary key, a unique constraint or a unique index.</p> <p><code>false</code>: The column is not required to uniquely identify the row.</p> |
| IsLong                   | <p><code>true</code>: The column contains a Binary Long Object (BLOB) that contains very long data. The definition of very long data is provider-specific.</p> <p><code>false</code>: The column does not contain a Binary Long Object (BLOB) that contains very long data.</p>   |
| IsReadOnly               | <p><code>true</code>: The column cannot be modified.</p> <p><code>false</code>: The column can be modified.</p>   |
| IsRowVersion             | <p><code>true</code>: The column contains a persistent row identifier that cannot be written to, and has no meaningful value except to identify the row.</p> <p><code>false</code>: The column does not contain a persistent row identifier that cannot be written to, and has no meaningful value except to identify the row.</p>  |
| IsUnicode                | <p><code>true</code>: Column is of type <code>timestamp</code>.</p> <p><code>false</code>: Column is not of type <code>timestamp</code>.</p>  |
| NonVersionedProviderType | The type of the column irrespective of the current <code>Type System Version</code> specified in the connection string. The returned value is from the <code>SqlDbType</code> enumeration.  |
| NumericPrecision         | If <code>ProviderType</code> is a numeric data type, this is the maximum precision of the column. The precision depends on the definition of the column. If <code>ProviderType</code> is not a numeric data type, this is 255.  |
| NumericScale             | If <code>ProviderType</code> is <code>DBTYPE_DECIMAL</code> or <code>DBTYPE_NUMERIC</code> , the number of digits to the right of the decimal point. Otherwise, this is 255.  |
| ProviderSpecificDataType | Returns the provider-specific data type of the column based on the <code>Type System Version</code> keyword in the connection string.   |
| ProviderType             | The indicator of the column's data type. If the data type of the column varies from row to row, this must be <code>Object</code> . This column cannot contain a null value.   |

| DataReader Column               | Description  |
|---------------------------------|--|
| UdtAssemblyQualifiedName        | If the column is a user-defined type (UDT), this is the qualified name of the UDT's assembly as per <a href="#">AssemblyQualifiedName</a> . If the column is not a UDT, this is null.  |
| XmlSchemaCollectionDatabase     | The name of the database where the schema collection for this XML instance is located, if the row contains information about an XML column. This value is <code>null</code> ( <code>Nothing</code> in Visual Basic) if the collection is defined within the current database. It is also null if there is no schema collection, in which case the <code>XmlSchemaCollectionName</code> and <code>XmlSchemaCollectionOwningSchema</code> columns are also null. |
| XmlSchemaCollectionName         | The name of the schema collection for this XML instance, if the row contains information about an XML column. This value is <code>null</code> ( <code>Nothing</code> in Visual Basic) if there is no associated schema collection. If the value is null, the <code>XmlSchemaCollectionDatabase</code> and <code>XmlSchemaCollectionOwningSchema</code> columns are also null.  |
| XmlSchemaCollectionOwningSchema | The owning relational schema where the schema collection for this XML instance is located, if the row contains information about an XML column. This value is <code>null</code> ( <code>Nothing</code> in Visual Basic) if the collection is defined within the current database. It is also null if there is no schema collection, in which case the <code>XmlSchemaCollectionDatabase</code> and <code>XmlSchemaCollectionName</code> columns are also null. |

**Note**

To make sure that metadata columns return the correct information, you must call [ExecuteReader](#) with the `behavior` parameter set to `KeyInfo`. Otherwise, some of the columns in the schema table may return default, null, or incorrect data.

See

[ADO.NET Overview](#)

Also

# SqlDataReader.GetSqlBinary SqlDataReader.GetSqlBinary

## In this Article

Gets the value of the specified column as a [SqlBinary](#).

```
public virtual System.Data.SqlTypes.SqlBinary GetSqlBinary (int i);  
abstract member GetSqlBinary : int -> System.Data.SqlTypes.SqlBinary  
override this.GetSqlBinary : int -> System.Data.SqlTypes.SqlBinary
```

## Parameters

|   |   |
|---|---|
| i | <a href="#">Int32</a> <a href="#">Int32</a> |
|---|---|

The zero-based column ordinal.

## Returns

[SqlBinary](#) [SqlBinary](#)

The value of the column expressed as a [SqlBinary](#).

## Remarks

No conversions are performed; therefore the data retrieved must already be a binary structure or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlBoolean SqlDataReader.GetSqlBoolean

## In this Article

Gets the value of the specified column as a [SqlBoolean](#).

```
public virtual System.Data.SqlTypes.SqlBoolean GetSqlBoolean (int i);  
  
abstract member GetSqlBoolean : int -> System.Data.SqlTypes.SqlBoolean  
override this.GetSqlBoolean : int -> System.Data.SqlTypes.SqlBoolean
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlBoolean](#)

The value of the column.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a Boolean or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlByte SqlDataReader.GetSqlByte

## In this Article

Gets the value of the specified column as a [SqlByte](#).

```
public virtual System.Data.SqlTypes.SqlByte GetSqlByte (int i);  
  
abstract member GetSqlByte : int -> System.Data.SqlTypes.SqlByte  
override this.GetSqlByte : int -> System.Data.SqlTypes.SqlByte
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlByte](#) [SqlByte](#)

The value of the column expressed as a [SqlByte](#).

## Remarks

No conversions are performed; therefore the data retrieved must already be a byte, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlBytes SqlDataReader.GetSqlBytes

## In this Article

Gets the value of the specified column as [SqlBytes](#).

```
public virtual System.Data.SqlTypes.SqlBytes GetSqlBytes (int i);  
  
abstract member GetSqlBytes : int -> System.Data.SqlTypes.SqlBytes  
override this.GetSqlBytes : int -> System.Data.SqlTypes.SqlBytes
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlBytes](#) [SqlBytes](#)

The value of the column expressed as a [SqlBytes](#).

## See

[Working with SqlDbTypes](#)

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlChars SqlDataReader.GetSqlChars

## In this Article

Gets the value of the specified column as [SqlChars](#).

```
public virtual System.Data.SqlTypes.SqlChars GetSqlChars (int i);  
  
abstract member GetSqlChars : int -> System.Data.SqlTypes.SqlChars  
override this.GetSqlChars : int -> System.Data.SqlTypes.SqlChars
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlChars](#) [SqlChars](#)

The value of the column expressed as a [SqlChars](#).

## See

## Also

[Working with SqlDbTypes](#)

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlDateTime SqlDataReader.GetSqlDateTime

## In this Article

Gets the value of the specified column as a [SqlDateTime](#).

```
public virtual System.Data.SqlTypes.SqlDateTime GetSqlDateTime (int i);  
abstract member GetSqlDateTime : int -> System.Data.SqlTypes.SqlDateTime  
override this.GetSqlDateTime : int -> System.Data.SqlTypes.SqlDateTime
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlDateTime](#)

The value of the column expressed as a [SqlDateTime](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a date/time value, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlDecimal SqlDataReader.GetSqlDecimal

## In this Article

Gets the value of the specified column as a [SqlDecimal](#).

```
public virtual System.Data.SqlTypes.SqlDecimal GetSqlDecimal (int i);  
abstract member GetSqlDecimal : int -> System.Data.SqlTypes.SqlDecimal  
override this.GetSqlDecimal : int -> System.Data.SqlTypes.SqlDecimal
```

## Parameters

|   |   |
|---|---|
| i | <a href="#">Int32</a> <a href="#">Int32</a> |
|---|---|

The zero-based column ordinal.

## Returns

[SqlDecimal](#) [SqlDecimal](#)

The value of the column expressed as a [SqlDecimal](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a decimal value, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlDouble SqlDataReader.GetSqlDouble

## In this Article

Gets the value of the specified column as a [SqlDouble](#).

```
public virtual System.Data.SqlTypes.SqlDouble GetSqlDouble (int i);  
abstract member GetSqlDouble : int -> System.Data.SqlTypes.SqlDouble  
override this.GetSqlDouble : int -> System.Data.SqlTypes.SqlDouble
```

## Parameters

|   |   |
|---|---|
| i | <a href="#">Int32</a> <a href="#">Int32</a> |
|---|---|

The zero-based column ordinal.

## Returns

[SqlDouble](#) [SqlDouble](#)

The value of the column expressed as a [SqlDouble](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a double-precision floating-point number, or an exception is generated.

### See

[Working with SqlDbType](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlGuid SqlDataReader.GetSqlGuid

## In this Article

Gets the value of the specified column as a [SqlGuid](#).

```
public virtual System.Data.SqlTypes.SqlGuid GetSqlGuid (int i);  
  
abstract member GetSqlGuid : int -> System.Data.SqlTypes.SqlGuid  
override this.GetSqlGuid : int -> System.Data.SqlTypes.SqlGuid
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlGuid](#) [SqlGuid](#)

The value of the column expressed as a [SqlGuid](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a GUID, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetInt16 SqlDataReader.GetInt16

## In this Article

Gets the value of the specified column as a [SqlInt16](#).

```
public virtual System.Data.SqlTypes.SqlInt16 GetSqlInt16 (int i);  
  
abstract member GetSqlInt16 : int -> System.Data.SqlTypes.SqlInt16  
override this.GetSqlInt16 : int -> System.Data.SqlTypes.SqlInt16
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlInt16](#)

The value of the column expressed as a [SqlInt16](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a 16-bit signed integer, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetInt32 SqlDataReader.GetInt32

## In this Article

Gets the value of the specified column as a [SqlInt32](#).

```
public virtual System.Data.SqlTypes.SqlInt32 GetSqlInt32 (int i);  
  
abstract member GetSqlInt32 : int -> System.Data.SqlTypes.SqlInt32  
override this.GetSqlInt32 : int -> System.Data.SqlTypes.SqlInt32
```

## Parameters

i [Int32](#) [Int32](#)

The zero-based column ordinal.

## Returns

[SqlInt32](#) [SqlInt32](#)

The value of the column expressed as a [SqlInt32](#).

## Remarks

No conversions are performed; therefore the data retrieved must already be a 32-bit signed integer, or an exception is generated.

See

[Working with SqlDbTypes](#)

Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetInt64 SqlDataReader.GetInt64

## In this Article

Gets the value of the specified column as a [SqlInt64](#).

```
public virtual System.Data.SqlTypes.SqlInt64 GetSqlInt64 (int i);  
  
abstract member GetSqlInt64 : int -> System.Data.SqlTypes.SqlInt64  
override this.GetSqlInt64 : int -> System.Data.SqlTypes.SqlInt64
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlInt64](#)

The value of the column expressed as a [SqlInt64](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a 64-bit signed integer, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlMoney SqlDataReader.GetSqlMoney

## In this Article

Gets the value of the specified column as a [SqlMoney](#).

```
public virtual System.Data.SqlTypes.SqlMoney GetSqlMoney (int i);  
  
abstract member GetSqlMoney : int -> System.Data.SqlTypes.SqlMoney  
override this.GetSqlMoney : int -> System.Data.SqlTypes.SqlMoney
```

## Parameters

|   |   |
|---|---|
| i | <a href="#">Int32</a> <a href="#">Int32</a> |
|---|---|

The zero-based column ordinal.

## Returns

[SqlMoney](#) [SqlMoney](#)

The value of the column expressed as a [SqlMoney](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a decimal value, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlSingle SqlDataReader.GetSqlSingle

## In this Article

Gets the value of the specified column as a [SqlSingle](#).

```
public virtual System.Data.SqlTypes.SqlSingle GetSqlSingle (int i);  
abstract member GetSqlSingle : int -> System.Data.SqlTypes.SqlSingle  
override this.GetSqlSingle : int -> System.Data.SqlTypes.SqlSingle
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlSingle](#)

The value of the column expressed as a [SqlSingle](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a single precision floating point number, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetString SqlDataReader.GetString

## In this Article

Gets the value of the specified column as a [SqlString](#).

```
public virtual System.Data.SqlTypes.SqlString GetString (int i);  
  
abstract member GetString : int -> System.Data.SqlTypes.SqlString  
override this.GetString : int -> System.Data.SqlTypes.SqlString
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[SqlString](#)

The value of the column expressed as a [SqlString](#).

## Remarks

No conversions are performed; therefore, the data retrieved must already be a string, or an exception is generated.

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetValue SqlDataReader.GetValue

## In this Article

Returns the data value in the specified column as a SQL Server type.

```
public virtual object GetValue (int i);  
  
abstract member GetValue : int -> obj  
override this.GetValue : int -> obj
```

## Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

## Returns

[Object Object](#)

The value of the column expressed as a [SqlDbType SqlDbType](#).

## Remarks

[GetValue](#) returns data using the native SQL Server types. To retrieve data using the .NET Framework types, see [GetValue](#).

### See

[Working with SqlDbTypes](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlValues SqlDataReader.GetSqlValues

## In this Article

Fills an array of [Object](#) that contains the values for all the columns in the record, expressed as SQL Server types.

```
public virtual int GetSqlValues (object[] values);  
  
abstract member GetSqlValues : obj[] -> int  
override this.GetSqlValues : obj[] -> int
```

## Parameters

values [Object](#)[]

An array of [Object](#) into which to copy the values. The column values are expressed as SQL Server types.

## Returns

[Int32](#) [Int32](#)

An integer indicating the number of columns copied.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`values` is null.

## Remarks

Returns the values for all the columns in the record in a single call, using the SQL type system instead of the CLR type system. The length of the [Object](#) array does not need to match the number of columns in the record. You can pass an [Object](#) array that contains fewer than the number of columns contained in the record. Only the amount of data the [Object](#) array holds is copied to the array, starting at the column with ordinal 0. You can also pass an [Object](#) array whose length is more than the number of columns contained in the resulting row. Any remaining columns are untouched.

## See

[Working with SqlDbTypes](#)

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetSqlXml SqlDataReader.GetSqlXml

## In this Article

Gets the value of the specified column as an XML value.

```
public virtual System.Data.SqlTypes.SqlXml GetSqlXml (int i);  
  
abstract member GetSqlXml : int -> System.Data.SqlTypes.SqlXml  
override this.GetSqlXml : int -> System.Data.SqlTypes.SqlXml
```

## Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

## Returns

[SqlXml](#) [SqlXml](#)

A [SqlXml](#) value that contains the XML stored within the corresponding field.

## Exceptions

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

The index passed was outside the range of 0 to [FieldCount](#) - 1

[InvalidOperationException](#) [InvalidOperationException](#)

An attempt was made to read or access columns in a closed [SqlDataReader](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The retrieved data is not compatible with the [SqlXml](#) type.

## Remarks

No conversions are performed; therefore, the data retrieved must already be an XML value.

Call [IsDBNull](#) to check for null values before calling this method.

### See

### Also

[Manipulating Data \(ADO.NET\)](#)

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetStream SqlDataReader.GetStream

## In this Article

Retrieves binary, image, varbinary, UDT, and variant data types as a [Stream](#).

```
public override System.IO.Stream GetStream (int i);  
override this.GetStream : int -> System.IO.Stream
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Stream Stream](#)

A stream object.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

The connection drops or is closed during the data retrieval.

The [SqlDataReader](#) is closed during the data retrieval.

There is no data ready to be read (for example, the first [Read\(\)](#) hasn't been called, or returned false).

Tried to read a previously-read column in sequential mode.

There was an asynchronous operation in progress. This applies to all Get\* methods when running in sequential mode, as they could be called while reading a stream.

[IndexOutOfRangeException IndexOutOfRangeException](#)

Trying to read a column that does not exist.

[InvalidOperationException InvalidOperationException](#)

The returned type was not one of the types below:

- binary
- image
- varbinary
- udt

## Remarks

[ReadTimeout](#) defaults to the value of [CommandTimeout](#); but you can modify [ReadTimeout](#) via [GetStream](#).

Null values will be returned as an empty (zero bytes) [Stream](#).

[GetBytes](#) will raise an [InvalidOperationException](#) exception when used on an object returned by [GetStream](#) when [SequentialAccess](#) is in effect.

[SqlException](#) exceptions raised from [Stream](#) are thrown as [IOException](#) exceptions; check the inner exception for the

## [SqlException](#).

The following [Stream](#) members are not available for objects returned by [GetStream](#):

- [BeginWrite](#)
- [EndWrite](#)
- [Length](#)
- [Position](#)
- [Seek](#)
- [SetLength](#)
- [Write](#)
- [WriteByte](#)
- [WriteTimeout](#)

When the connection property `ContextConnection=true`, [GetStream](#) only supports synchronous data retrieval for both sequential ([SequentialAccess](#)) and non-sequential ([Default](#)) access.

For more information, see [SqlClient Streaming Support](#).

# SqlDataReader.GetString SqlDataReader.GetString

## In this Article

Gets the value of the specified column as a string.

```
public override string GetString (int i);  
override this.GetString : int -> string
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[String String](#)

The value of the specified column.

### Exceptions

[InvalidOperationException InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a string.

Call [IsDBNull IsDBNull](#) to check for null values before calling this method.

### See

### Also

[Working with SqlDbTypes Working with SqlDbTypes](#)

[DataAdapters and DataReaders \(ADO.NET\) DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET SQL Server and ADO.NET](#)

[ADO.NET Overview ADO.NET Overview](#)

# SqlDataReader.GetTextReader SqlDataReader.GetTextReader

## In this Article

Retrieves Char, NChar, NText, NVarChar, text, varChar, and Variant data types as a [TextReader](#).

```
public override System.IO.TextReader GetTextReader (int i);  
override this.GetTextReader : int -> System.IO.TextReader
```

## Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The column to be retrieved.

## Returns

[TextReader](#) [TextReader](#)

The returned object.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The connection drops or is closed during the data retrieval.

The [SqlDataReader](#) is closed during the data retrieval.

There is no data ready to be read (for example, the first [Read\(\)](#) hasn't been called, or returned false).

Tried to read a previously-read column in sequential mode.

There was an asynchronous operation in progress. This applies to all Get\* methods when running in sequential mode, as they could be called while reading a stream.

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

Trying to read a column that does not exist.

[InvalidCastException](#) [InvalidCastException](#)

The returned type was not one of the types below:

- char
- nchar
- ntext
- nvarchar
- text
- varchar

## Remarks

[SqlException](#) exceptions raised from [TextReader](#) are thrown as [IOException](#) exceptions; check the inner exception for the [SqlException](#).

Null values will be returned as an empty (zero bytes) [TextReader](#).

[GetChars](#) will raise an [InvalidOperationException](#) exception when used on an object returned by [GetTextReader](#) when [SequentialAccess](#) is in effect.

When the connection property `ContextConnection=true`, [GetTextReader](#) only supports synchronous data retrieval for both sequential ([SequentialAccess](#)) and non-sequential ([Default](#)) access.

For more information, see [SqlClient Streaming Support](#).

# SqlDataReader.GetTimeSpan SqlDataReader.GetTimeSpan

## In this Article

Retrieves the value of the specified column as a [TimeSpan](#) object.

```
public virtual TimeSpan GetTimeSpan (int i);  
  
abstract member GetTimeSpan : int -> TimeSpan  
override this.GetTimeSpan : int -> TimeSpan
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The zero-based column ordinal.

## Returns

[TimeSpan](#)

The value of the specified column.

## Exceptions

[InvalidOperationException](#)

The specified cast is not valid.

## Remarks

No conversions are performed; therefore, the data retrieved must already be a [TimeSpan](#) object.

Call [IsDBNull](#) to check for null values before calling this method.

### See

[Date and Time Data Types \(Katmai\)](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetValue SqlDataReader.GetValue

## In this Article

Gets the value of the specified column in its native format.

```
public override object GetValue (int i);  
override this.GetValue : int -> obj
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Object Object](#)

This method returns [DBNull DBNull](#) for null database columns.

## Remarks

[GetValue](#) returns data using the .NET Framework types.

### See

### Also

[Working with SqlDbTypes](#)

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetValues SqlDataReader.GetValues

## In this Article

Populates an array of objects with the column values of the current row.

```
public override int GetValues (object[] values);  
override this.GetValues : obj[] -> int
```

### Parameters

values [Object\[\]](#)

An array of [Object](#) into which to copy the attribute columns.

### Returns

[Int32](#) [Int32](#)

The number of instances of [Object](#) in the array.

## Examples

The following example demonstrates using a correctly sized array to read all values from the current row in the supplied [SqlDataReader](#). In addition, the sample demonstrates using a fixed-sized array that could be either smaller or larger than the number of available columns.

```
private static void TestGetValues(SqlDataReader reader)  
{  
    // Given a SqlDataReader, use the GetValues  
    // method to retrieve a full row of data.  
    // Test the GetValues method, passing in an array large  
    // enough for all the columns.  
    Object[] values = new Object[reader.FieldCount];  
    int fieldCount = reader.GetValues(values);  
  
    Console.WriteLine("reader.GetValues retrieved {0} columns.",  
        fieldCount);  
    for (int i = 0; i < fieldCount; i++)  
        Console.WriteLine(values[i]);  
  
    Console.WriteLine();  
  
    // Now repeat, using an array that may contain a different  
    // number of columns than the original data. This should work correctly,  
    // whether the size of the array is larger or smaller than  
    // the number of columns.  
  
    // Attempt to retrieve three columns of data.  
    values = new Object[3];  
    fieldCount = reader.GetValues(values);  
    Console.WriteLine("reader.GetValues retrieved {0} columns.",  
        fieldCount);  
    for (int i = 0; i < fieldCount; i++)  
        Console.WriteLine(values[i]);  
}
```

## Remarks

For most applications, this method provides an efficient means for retrieving all columns, instead of retrieving each column individually.

You can pass an [Object](#) array that contains fewer than the number of columns contained in the resulting row. Only the amount of data the [Object](#) array holds is copied to the array. You can also pass an [Object](#) array whose length is more than the number of columns contained in the resulting row.

This method returns [DBNull](#) for null database columns.

See

Also

[Working with SqlDbTypes](#)

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.GetXmlReader SqlDataReader.GetXmlReader

## In this Article

Retrieves data of type XML as an [XmlReader](#).

```
public virtual System.Xml.XmlReader GetXmlReader (int i);  
  
abstract member GetXmlReader : int -> System.Xml.XmlReader  
override this.GetXmlReader : int -> System.Xml.XmlReader
```

## Parameters

|   |                       |
|---|-----------------------|
| i | <a href="#">Int32</a> |
|---|-----------------------|

The value of the specified column.

## Returns

[XmlReader](#) [XmlReader](#)

The returned object.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The connection drops or is closed during the data retrieval.

The [SqlDataReader](#) is closed during the data retrieval.

There is no data ready to be read (for example, the first [Read\(\)](#) hasn't been called, or returned false).

Trying to read a previously read column in sequential mode.

There was an asynchronous operation in progress. This applies to all Get\* methods when running in sequential mode, as they could be called while reading a stream.

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

Trying to read a column that does not exist.

[InvalidOperationException](#) [InvalidOperationException](#)

The returned type was not xml.

## Remarks

The [XmlReader](#) object returned by [GetXmlReader](#) does not support asynchronous operations. If you require asynchronous operations on an [XmlReader](#), cast the XML column to an NVARCHAR(MAX) on the server and use [GetTextReader](#) with [Create](#).

[SqlException](#) exceptions raised from [XmlReader](#) are thrown as [XmlException](#) exceptions; check the inner exception for the [SqlException](#).

[GetChars](#) will raise an [InvalidOperationException](#) exception when used on an object returned by [GetXmlReader](#) when [SequentialAccess](#) is in effect.

For more information, see [SqlClient Streaming Support](#).

# SqlDataReader.HasRows SqlDataReader.HasRows

## In this Article

Gets a value that indicates whether the [SqlDataReader](#) contains one or more rows.

```
public override bool HasRows { get; }  
member this.HasRows : bool
```

Returns

[Boolean](#)

`true` if the [SqlDataReader](#) contains one or more rows; otherwise `false`.

See

Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.IDataRecord.GetData

## In this Article

Returns an [IDataReader](#) for the specified column ordinal.

```
System.Data.IDataReader IDataReader.GetData (int i);
```

### Parameters

|   |       |
|---|-------|
| i | Int32 |
|---|-------|

A column ordinal.

### Returns

[IDataReader](#)

The [IDataReader](#) instance for the specified column ordinal.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlDataReader](#) instance is cast to an [IDataRecord](#) interface.

### See

[ADO.NET Overview](#)

### Also

# SqlDataReader.IDisposable.Dispose

## In this Article

```
void IDisposable.Dispose();
```

# SqlDataReader.IEnumerable.GetEnumerator

## In this Article

```
System.Collections.IEnumerator IEnumerable.GetEnumerator();
```

## Returns

[IEnumerator](#)

# SqlDataReader.IsClosed SqlDataReader.IsClosed

## In this Article

Retrieves a Boolean value that indicates whether the specified [SqlDataReader](#) instance has been closed.

```
public override bool IsClosed { get; }  
member this.IsClosed : bool
```

Returns

[Boolean](#)

`true` if the specified [SqlDataReader](#) instance is closed; otherwise `false`.

## Remarks

It is not possible to read from a [SqlDataReader](#) instance that is closed.

See

Also

[SQL Server Connection Pooling \(ADO.NET\)](#)

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.IsCommandBehavior SqlDataReader.IsCommandBehavior

## In this Article

Determines whether the specified [CommandBehavior](#) matches that of the [SqlDataReader](#).

```
protected bool IsCommandBehavior (System.Data.CommandBehavior condition);  
member this.IsCommandBehavior : System.Data.CommandBehavior -> bool
```

### Parameters

condition [CommandBehavior](#) [CommandBehavior](#)

A [CommandBehavior](#) enumeration.

### Returns

[Boolean](#) [Boolean](#)

`true` if the specified [CommandBehavior](#) is true, `false` otherwise.

## Remarks

This member supports the .NET Framework infrastructure and is not intended to be used directly from your code.

### See

[ADO.NET Overview](#)

### Also

# SqlDataReader.IsDBNull SqlDataReader.IsDBNull

## In this Article

Gets a value that indicates whether the column contains non-existent or missing values.

```
public override bool IsDBNull (int i);  
override this.IsDBNull : int -> bool
```

### Parameters

|   |             |
|---|-------------|
| i | Int32 Int32 |
|---|-------------|

The zero-based column ordinal.

### Returns

[Boolean Boolean](#)

`true` if the specified column value is equivalent to [DBNull](#); otherwise `false`.

## Remarks

Call this method to check for null column values before calling the typed get methods (for example, [GetByte](#), [GetChar](#), and so on) to avoid raising an error.

```
using System;  
using System.Data;  
using System.Data.SqlClient;  
  
class Program {  
    static void Main(string[] args) {  
  
        using (var connection = new SqlConnection(@"Data Source=(local);Initial  
Catalog=AdventureWorks2012;Integrated Security=SSPI")) {  
            var command = new SqlCommand("SELECT p.FirstName, p.MiddleName, p.LastName FROM  
HumanResources.Employee AS e" +  
                " JOIN Person.Person AS p ON e.BusinessEntityID =  
p.BusinessEntityID;", connection);  
            connection.Open();  
            var reader = command.ExecuteReader();  
            while (reader.Read()) {  
                Console.WriteLine(reader.GetString(reader.GetOrdinal("FirstName")));  
                // display middle name only if not null  
                if (!reader.IsDBNull(reader.GetOrdinal("MiddleName")))  
                    Console.WriteLine(" {0}", reader.GetString(reader.GetOrdinal("MiddleName")));  
                Console.WriteLine(" {0}", reader.GetString(reader.GetOrdinal("LastName")));  
            }  
            connection.Close();  
        }  
    }  
}
```

See

Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.IsDBNullAsync SqlDataReader.IsDBNullAsync

## In this Article

An asynchronous version of [IsDBNull\(Int32\)](#), which gets a value that indicates whether the column contains non-existent or missing values.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public override System.Threading.Tasks.Task<bool> IsDBNullAsync (int i,
System.Threading.CancellationToken cancellationToken);

override this.IsDBNullAsync : int * System.Threading.CancellationToken ->
System.Threading.Tasks.Task<bool>
```

## Parameters

i [Int32 Int32](#)

The zero-based column to be retrieved.

cancellationToken [CancellationToken CancellationToken](#)

The cancellation instruction, which propagates a notification that operations should be canceled. This does not guarantee the cancellation. A setting of [CancellationToken.None](#) makes this method equivalent to [IsDBNull\(Int32\)](#). The returned task must be marked as cancelled.

## Returns

[Task<Boolean>](#)

[true](#) if the specified column value is equivalent to [DBNull](#) otherwise [false](#).

## Exceptions

[InvalidOperationException InvalidOperationException](#)

The connection drops or is closed during the data retrieval.

The [SqlDataReader](#) is closed during the data retrieval.

There is no data ready to be read (for example, the first [Read\(\)](#) hasn't been called, or returned false).

Trying to read a previously read column in sequential mode.

There was an asynchronous operation in progress. This applies to all Get\* methods when running in sequential mode, as they could be called while reading a stream.

[Context Connection=true](#) is specified in the connection string.

[IndexOutOfRangeException IndexOutOfRangeException](#)

Trying to read a column that does not exist.

## Remarks

For more information, see [SqlClient Streaming Support](#).

# SqlDataReader.Item[Int32] SqlDataReader.Item[Int32]

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">Item[String] Item[String]</a> | Gets the value of the specified column in its native format given the column name.    |
| <a href="#">Item[Int32] Item[Int32]</a>   | Gets the value of the specified column in its native format given the column ordinal. |

## Item[String] Item[String]

Gets the value of the specified column in its native format given the column name.

```
public override object this[string name] { get; }  
member this.Item(string) : obj
```

### Parameters

name String String

The column name.

### Returns

[Object Object](#)

The value of the specified column in its native format.

### Exceptions

[IndexOutOfRangeException IndexOutOfRangeException](#)

No column with the specified name was found.

### Remarks

A case-sensitive lookup is performed first. If it fails, a second case-insensitive search is made (a case-insensitive comparison is done using the database collation). Unexpected results can occur when comparisons are affected by culture-specific casing rules. For example, in Turkish, the following example yields the wrong results because the file system in Turkish does not use linguistic casing rules for the letter 'i' in "file".

This method is kana-width insensitive.

### See

[DataAdapters and DataReaders \(ADO.NET\)](#)

### Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

## Item[Int32] Item[Int32]

Gets the value of the specified column in its native format given the column ordinal.

```
public override object this[int i] { get; }  
member this.Item(int) : obj
```

## Parameters

|   |                             |
|---|-----------------------------|
| i | <a href="#">Int32 Int32</a> |
|---|-----------------------------|

The zero-based column ordinal.

## Returns

[Object Object](#)

The value of the specified column in its native format.

## Exceptions

[IndexOutOfRangeException IndexOutOfRangeException](#)

The index passed was outside the range of 0 through [FieldCount](#).

## See

## Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.NextResult SqlDataReader.NextResult

## In this Article

Advances the data reader to the next result, when reading the results of batch Transact-SQL statements.

```
public override bool NextResult ();  
override this.NextResult : unit -> bool
```

Returns

[Boolean Boolean](#)

`true` if there are more result sets; otherwise `false`.

## Remarks

Used to process multiple results, which can be generated by executing batch Transact-SQL statements.

By default, the data reader is positioned on the first result.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.NextResultAsync SqlDataReader.NextResultAsync

## In this Article

An asynchronous version of [NextResult\(\)](#), which advances the data reader to the next result, when reading the results of batch Transact-SQL statements.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public override System.Threading.Tasks.Task<bool> NextResultAsync  
(System.Threading.CancellationToken cancellationToken);  
  
override this.NextResultAsync : System.Threading.CancellationToken ->  
System.Threading.Tasks.Task<bool>
```

## Parameters

cancellationToken

[CancellationToken](#) [CancellationToken](#)

The cancellation instruction.

## Returns

[Task<Boolean>](#)

A task representing the asynchronous operation.

## Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

Calling [NextResultAsync\(CancellationToken\)](#) more than once for the same instance before task completion.

`Context Connection=true` is specified in the connection string.

[SQLException](#) [SQLException](#)

SQL Server returned an error while executing the command text.

## Remarks

For more information about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

# SqlDataReader.Read SqlDataReader.Read

## In this Article

Advances the [SqlDataReader](#) to the next record.

```
public override bool Read ();  
override this.Read : unit -> bool
```

Returns

[Boolean](#) Boolean

`true` if there are more rows; otherwise `false`.

Exceptions

[SqlException](#) [SqlException](#)

SQL Server returned an error while executing the command text.

## Examples

The following example creates a [SqlConnection](#), a [SqlCommand](#), and a [SqlDataReader](#). The example reads through the data, writing it out to the console window. The code then closes the [SqlDataReader](#). The [SqlConnection](#) is closed automatically at the end of the `using` code block.

```

using System;
using System.Data;
using System.Data.SqlClient;

class Program
{
    static void Main()
    {
        string str = "Data Source=(local);Initial Catalog=Northwind;" +
            + "Integrated Security=SSPI";
        ReadOrderData(str);
    }

    private static void ReadOrderData(string connectionString)
    {
        string queryString =
            "SELECT OrderID, CustomerID FROM dbo.Orders;";

        using (SqlConnection connection =
            new SqlConnection(connectionString))
        {
            SqlCommand command =
                new SqlCommand(queryString, connection);
            connection.Open();

            SqlDataReader reader = command.ExecuteReader();

            // Call Read before accessing data.
            while (reader.Read())
            {
                ReadSingleRow((IDataRecord)reader);
            }

            // Call Close when done reading.
            reader.Close();
        }
    }

    private static void ReadSingleRow(IDataRecord record)
    {
        Console.WriteLine(String.Format("{0}, {1}", record[0], record[1]));
    }
}

```

## Remarks

The default position of the [SqlDataReader](#) is before the first record. Therefore, you must call [Read](#) to begin accessing any data.

Only one [SqlDataReader](#) per associated [SqlConnection](#) may be open at a time, and any attempt to open another will fail until the first one is closed. Similarly, while the [SqlDataReader](#) is being used, the associated [SqlConnection](#) is busy serving it until you call [Close](#).

See

Also

[SQL Server Connection Pooling \(ADO.NET\)](#)  
[DataAdapters and DataReaders](#)  
[SQL Server and ADO.NET](#)  
[ADO.NET Overview](#)

# SqlDataReader.ReadAsync SqlDataReader.ReadAsync

## In this Article

An asynchronous version of [Read\(\)](#), which advances the [SqlDataReader](#) to the next record.

The cancellation token can be used to request that the operation be abandoned before the command timeout elapses. Exceptions will be reported via the returned Task object.

```
public override System.Threading.Tasks.Task<bool> ReadAsync (System.Threading.CancellationToken cancellationToken);  
override this.ReadAsync : System.Threading.CancellationToken -> System.Threading.Tasks.Task<bool>
```

### Parameters

cancellationToken

CancellationToken CancellationToken

The cancellation instruction.

### Returns

[Task<Boolean>](#)

A task representing the asynchronous operation.

### Exceptions

[InvalidOperationException](#) InvalidOperationException

Calling [ReadAsync\(CancellationToken\)](#) more than once for the same instance before task completion.

`Context Connection=true` is specified in the connection string.

[SqlException](#) [SqlException](#)

SQL Server returned an error while executing the command text.

## Remarks

If the `behavior` parameter of [ExecuteReaderAsync](#) is set to `Default`, [ReadAsync](#) reads the entire row before returning the Task.

For more information, including code samples, about asynchronous programming in the .NET Framework Data Provider for SQL Server, see [Asynchronous Programming](#).

See

[ADO.NET Overview](#)

Also

# SqlDataReader.RecordsAffected SqlDataReader.RecordsAffected

## In this Article

Gets the number of rows changed, inserted, or deleted by execution of the Transact-SQL statement.

```
public override int RecordsAffected { get; }  
member this.RecordsAffected : int
```

## Returns

[Int32](#) [Int32](#)

The number of rows changed, inserted, or deleted; 0 if no rows were affected or the statement failed; and -1 for SELECT statements.

## Remarks

The value of this property is cumulative. For example, if two records are inserted in batch mode, the value of [RecordsAffected](#) will be two.

[IsClosed](#) and [RecordsAffected](#) are the only properties that you can call after the [SqlDataReader](#) is closed.

### See

[SQL Server Connection Pooling \(ADO.NET\)](#)

### Also

[DataAdapters and DataReaders \(ADO.NET\)](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlDataReader.VisibleFieldCount SqlDataReader.VisibleFieldCount

## In this Article

Gets the number of fields in the [SqlDataReader](#) that are not hidden.

```
public override int VisibleFieldCount { get; }  
member this.VisibleFieldCount : int
```

Returns

[Int32](#) [Int32](#)

The number of fields that are not hidden.

## Remarks

This value is used to determine how many fields in the [SqlDataReader](#) are visible. For example, a SELECT on a partial primary key returns the remaining parts of the key as hidden fields. The hidden fields are always appended behind the visible fields.

See

[DataAdapters and DataReaders \(ADO.NET\)](#)

Also

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SQLDebugging SQLDebugging Class

Included to support debugging applications. Not intended for direct use.

## Declaration

```
[System.Runtime.InteropServices.ClassInterface]
[System.Runtime.InteropServices.ComVisible(true)]
[System.Runtime.InteropServices.Guid("afef65ad-4577-447a-a148-83acadd3d4b9")]
[System.Runtime.InteropServices.ClassInterface(System.Runtime.InteropServices.ClassInterfaceType
.None)]
public sealed class SQLDebugging

type SQLDebugging = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

`SQLDebugging()`

`SQLDebugging()`

Included to support debugging applications. Not intended for direct use.

## See Also

# SQLDebugging

## In this Article

Included to support debugging applications. Not intended for direct use.

```
public SQLDebugging ();
```

See

[ADO.NET Overview](#)

Also

# SqlDependency SqlDependency Class

The [SqlDependency](#) object represents a query notification dependency between an application and an instance of SQL Server. An application can create a [SqlDependency](#) object and register to receive notifications via the [OnChangeEventHandler](#) event handler.

## Declaration

```
public sealed class SqlDependency  
type SqlDependency = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

[SqlDependency](#) is ideal for caching scenarios, where your ASP.NET application or middle-tier service needs to keep certain information cached in memory. [SqlDependency](#) allows you to receive notifications when the original data in the database changes so that the cache can be refreshed.

To set up a dependency, you need to associate a [SqlDependency](#) object to one or more [SqlCommand](#) objects. To receive notifications, you need to subscribe to the [OnChange](#) event. For more information about the requirements for creating queries for notifications, see [Working with Query Notifications](#).

**Note**

[SqlDependency](#) was designed to be used in ASP.NET or middle-tier services where there is a relatively small number of servers having dependencies active against the database. It was not designed for use in client applications, where hundreds or thousands of client computers would have [SqlDependency](#) objects set up for a single database server. If you are developing an application where you need reliable sub-second notifications when data changes, review the sections [Planning an Efficient Query Notifications Strategy](#) and [Alternatives to Query Notifications](#) in the [Planning for Notifications](#) article.

For more information, see [Query Notifications in SQL Server](#) and [Building Notification Solutions](#).

**Note**

The [OnChange](#) event may be generated on a different thread from the thread that initiated command execution.

Query notifications are supported only for SELECT statements that meet a list of specific requirements.

## Constructors

[SqlDependency\(\)](#)

[SqlDependency\(SqlCommand\)](#)

Creates a new instance of the [SqlDependency](#) class with the default settings.

[SqlDependency\(SqlCommand\)](#)

[SqlDependency\(SqlCommand\)](#)

Creates a new instance of the [SqlDependency](#) class and associates it with the [SqlCommand](#) parameter.

```
SqlDependency(SqlCommand, String, Int32)
SqlDependency(SqlCommand, String, Int32)
```

Creates a new instance of the [SqlDependency](#) class, associates it with the [SqlCommand](#) parameter, and specifies notification options and a time-out value.

## Properties

```
HasChanges
```

```
HasChanges
```

Gets a value that indicates whether one of the result sets associated with the dependency has changed.

```
Id
```

```
Id
```

Gets a value that uniquely identifies this instance of the [SqlDependency](#) class.

## Methods

```
AddCommandDependency(SqlCommand)
```

```
AddCommandDependency(SqlCommand)
```

Associates a [SqlCommand](#) object with this [SqlDependency](#) instance.

```
Start(String)
```

```
Start(String)
```

Starts the listener for receiving dependency change notifications from the instance of SQL Server specified by the connection string.

```
Start(String, String)
```

```
Start(String, String)
```

Starts the listener for receiving dependency change notifications from the instance of SQL Server specified by the connection string using the specified SQL Server Service Broker queue.

```
Stop(String)
```

```
Stop(String)
```

Stops a listener for a connection specified in a previous [Start](#) call.

```
Stop(String, String)
```

```
Stop(String, String)
```

Stops a listener for a connection specified in a previous [Start](#) call.

## Events

OnChange

OnChange

Occurs when a notification is received for any of the commands associated with this [SqlDependency](#) object.

## See Also

# SqlDependency.AddCommandDependency SqlDependency.AddCommandDependency

## In this Article

Associates a [SqlCommand](#) object with this [SqlDependency](#) instance.

```
public void AddCommandDependency (System.Data.SqlClient.SqlCommand command);  
member this.AddCommandDependency : System.Data.SqlClient.SqlCommand -> unit
```

### Parameters

command [SqlCommand](#) [SqlCommand](#)

A [SqlCommand](#) object containing a statement that is valid for notifications.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The `command` parameter is null.

[InvalidOperationException](#) [InvalidOperationException](#)

The [SqlCommand](#) object already has a [SqlNotificationRequest](#) object assigned to its [Notification](#) property, and that [SqlNotificationRequest](#) is not associated with this dependency.

## Remarks

Query notifications are supported only for SELECT statements that meet a list of specific requirements. For more information, see [SQL Server Service Broker](#) and [Working with Query Notifications](#).

### See

[Using Query Notifications](#)

### Also

[ADO.NET Overview](#)

# SqlDependency.HasChanges SqlDependency.HasChanges

## In this Article

Gets a value that indicates whether one of the result sets associated with the dependency has changed.

```
public bool HasChanges { get; }  
member this.HasChanges : bool
```

Returns

**Boolean Boolean**

A Boolean value indicating whether one of the result sets has changed.

## Remarks

If you are not using the [OnChange](#) event, you can check the [HasChanges](#) property to determine if the query results have changed.

The [HasChanges](#) property does not necessarily imply a change in the data. Other circumstances, such as time-out expired and failure to set the notification request, also generate a change event.

See

[Using Query Notifications](#)

Also

[ADO.NET Overview](#)

# SqlDependency.Id SqlDependency.Id

## In this Article

Gets a value that uniquely identifies this instance of the [SqlDependency](#) class.

```
public string Id { get; }  
member this.Id : string
```

Returns

[String String](#)

A string representation of a GUID that is generated for each instance of the [SqlDependency](#) class.

## Remarks

The [Id](#) property is used to uniquely identify a given [SqlDependency](#) instance.

See

Also

[Using Query Notifications](#)

[ADO.NET Overview](#)

# SqlDependency.OnChange SqlDependency.OnChange

## In this Article

Occurs when a notification is received for any of the commands associated with this [SqlDependency](#) object.

```
public event System.Data.SqlClient.OnChangeEventHandler OnChange;  
member this.OnChange : System.Data.SqlClient.OnChangeEventHandler
```

## Remarks

[OnChange](#) occurs when the results for the associated command change. If you are not using [OnChange](#), you can check the [HasChanges](#) property to determine whether the query results have changed.

The [OnChange](#) event does not necessarily imply a change in the data. Other circumstances, such as time-out expired and failure to set the notification request, also generate [OnChange](#).

See

[Using Query Notifications](#)

Also

[ADO.NET Overview](#)

# SqlDependency SqlDependency

In this Article

## Overloads

|   |  |
|---|--|
| <a href="#">SqlDependency()</a>   | Creates a new instance of the <a href="#">SqlDependency</a> class with the default settings.   |
| <a href="#">SqlDependency(SqlCommand)</a> <a href="#">SqlDependency(SqlCommand)</a>                               | Creates a new instance of the <a href="#">SqlDependency</a> class and associates it with the <a href="#">SqlCommand</a> parameter.   |
| <a href="#">SqlDependency(SqlCommand, String, Int32)</a> <a href="#">SqlDependency(SqlCommand, String, Int32)</a> | Creates a new instance of the <a href="#">SqlDependency</a> class, associates it with the <a href="#">SqlCommand</a> parameter, and specifies notification options and a time-out value. |

## SqlDependency()

Creates a new instance of the [SqlDependency](#) class with the default settings.

```
public SqlDependency();
```

### Remarks

The constructor initializes the [SqlDependency](#) object using the default Service Broker service name and time-out. At some point after construction, you must use the [AddCommandDependency](#) method to associate one or more commands to this [SqlDependency](#) object.

Query notifications are supported only for SELECT statements that meet a list of specific requirements. For more information, see [SQL Server Service Broker](#) and [Working with Query Notifications](#).

See

[Using Query Notifications](#)

Also

[ADO.NET Overview](#)

## SqlDependency(SqlCommand) SqlDependency(SqlCommand)

Creates a new instance of the [SqlDependency](#) class and associates it with the [SqlCommand](#) parameter.

```
public SqlDependency (System.Data.SqlClient.SqlCommand command);  
new System.Data.SqlClient.SqlDependency : System.Data.SqlClient.SqlCommand ->  
System.Data.SqlClient.SqlDependency
```

### Parameters

command

[SqlCommand](#) [SqlCommand](#)

The [SqlCommand](#) object to associate with this [SqlDependency](#) object. The constructor will set up a [SqlNotificationRequest](#) object and bind it to the command.

### Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The `command` parameter is NULL.

#### [InvalidOperationException](#) InvalidOperationException

The `SqlCommand` object already has a `SqlNotificationRequest` object assigned to its `Notification` property, and that `SqlNotificationRequest` is not associated with this dependency.

#### Remarks

Internally, this constructor creates an instance of the `SqlNotificationRequest` class, and binds it to a `SqlCommand` object.

Query notifications are supported only for SELECT statements that meet a list of specific requirements. For more information, see [SQL Server Service Broker](#) and [Working with Query Notifications](#).

#### See

[Using Query Notifications](#)

#### Also

[ADO.NET Overview](#)

## **SqlDependency(SqlCommand, String, Int32)**

## **SqlDependency(SqlCommand, String, Int32)**

Creates a new instance of the `SqlDependency` class, associates it with the `SqlCommand` parameter, and specifies notification options and a time-out value.

```
public SqlDependency (System.Data.SqlClient.SqlCommand command, string options, int timeout);
new System.Data.SqlClient.SqlDependency : System.Data.SqlClient.SqlCommand * string * int ->
System.Data.SqlClient.SqlDependency
```

#### Parameters

##### command

`SqlCommand` `SqlCommand`

The `SqlCommand` object to associate with this `SqlDependency` object. The constructor sets up a `SqlNotificationRequest` object and bind it to the command.

##### options

`String` `String`

The notification request options to be used by this dependency. `null` to use the default service.

##### timeout

`Int32` `Int32`

The time-out for this notification in seconds. The default is 0, indicating that the server's time-out should be used.

#### Exceptions

##### [ArgumentNullException](#) ArgumentNullException

The `command` parameter is NULL.

#### [ArgumentOutOfRangeException](#) ArgumentOutOfRangeException

The time-out value is less than zero.

#### [InvalidOperationException](#) InvalidOperationException

The `SqlCommand` object already has a `SqlNotificationRequest` object assigned to its `Notification` property and that `SqlNotificationRequest` is not associated with this dependency.

An attempt was made to create a `SqlDependency` instance from within SQLCLR.

## Remarks

Query notifications are supported only for SELECT statements that meet a list of specific requirements. For more information, see [SQL Server Service Broker](#) and [Working with Query Notifications](#).

See

[Using Query Notifications](#)

Also

[ADO.NET Overview](#)

# SqlDependency.Start SqlDependency.Start

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">Start(String) Start(String)</a>                 | Starts the listener for receiving dependency change notifications from the instance of SQL Server specified by the connection string.   |
| <a href="#">Start(String, String) Start(String, String)</a> | Starts the listener for receiving dependency change notifications from the instance of SQL Server specified by the connection string using the specified SQL Server Service Broker queue. |

## Remarks

The [SqlDependency](#) listener will restart when an error occurs in the SQL Server connection.

Multiple calls to the [Start](#) method can be made, subject to the following restrictions:

- Multiple calls with identical parameters (the same connection string and Windows credentials in the calling thread) are valid.
- Multiple calls with different connection strings are valid as long as:
  - Each connection string specifies a different database, or
  - Each connection string specifies a different user, or
  - The calls come from different application domains.

You can make the [SqlDependency](#) work correctly for applications that use multiple threads to represent different user credentials without giving the dbo role to the group, because different users can subscribe and listen (using [SqlCacheDependency](#) or [SqlCommand](#)) to a notification queue created by an administrator. When the relevant application domain starts, call Start with the (Windows) credentials of a user that has permission to initialize a service/queue (the CREATE QUEUE and CREATE SERVICE permissions for the database). Ensure that Start is only called once per AppDomain, otherwise an ambiguity exception is raised. The user thread must have permission to subscribe to the notification (the SUBSCRIBE QUERY NOTIFICATIONS permission for the database). [SqlDependency](#) will associate the subscription request of a non-administrator user to the service/queue created by the administrator.

## Start(String) Start(String)

Starts the listener for receiving dependency change notifications from the instance of SQL Server specified by the connection string.

```
public static bool Start (string connectionString);  
static member Start : string -> bool
```

Parameters

connectionString

[String](#) [String](#)

The connection string for the instance of SQL Server from which to obtain change notifications.

Returns

[Boolean Boolean](#)

**true** if the listener initialized successfully; **false** if a compatible listener already exists.

Exceptions

[ArgumentNullException ArgumentNullException](#)

The `connectionString` parameter is NULL.

[InvalidOperationException InvalidOperationException](#)

The `connectionString` parameter is the same as a previous call to this method, but the parameters are different.

The method was called from within the CLR.

[SecurityException SecurityException](#)

The caller does not have the required `SqlClientPermission` code access security (CAS) permission.

[SqlException SqlException](#)

A subsequent call to the method has been made with an equivalent `connectionString` parameter with a different user, or a user that does not default to the same schema.

Also, any underlying **SqlClient** exceptions.

Remarks

This method starts the listener for the `AppDomain` for receiving dependency notifications from the instance of SQL Server specified by the `connectionString` parameter. This method may be called more than once with different connection strings for multiple servers.

For additional remarks, see [Start](#).

See

[Using Query Notifications](#)

Also

[ADO.NET Overview](#)

## Start(String, String) Start(String, String)

Starts the listener for receiving dependency change notifications from the instance of SQL Server specified by the connection string using the specified SQL Server Service Broker queue.

```
public static bool Start (string connectionString, string queue);  
static member Start : string * string -> bool
```

Parameters

connectionString

[String String](#)

The connection string for the instance of SQL Server from which to obtain change notifications.

queue

[String String](#)

An existing SQL Server Service Broker queue to be used. If `null`, the default queue is used.

Returns

[Boolean Boolean](#)

**true** if the listener initialized successfully; **false** if a compatible listener already exists.

Exceptions

#### [ArgumentNullException](#) [ArgumentNullException](#)

The `connectionString` parameter is NULL.

#### [InvalidOperationException](#) [InvalidOperationException](#)

The `connectionString` parameter is the same as a previous call to this method, but the parameters are different.

The method was called from within the CLR.

#### [SecurityException](#) [SecurityException](#)

The caller does not have the required [SqlClientPermission](#) code access security (CAS) permission.

#### [SqlException](#) [SqlException](#)

A subsequent call to the method has been made with an equivalent `connectionString` parameter but a different user, or a user that does not default to the same schema.

Also, any underlying **SqlClient** exceptions.

### Remarks

This method starts the listener for the [AppDomain](#) for receiving dependency notifications from the instance of SQL Server specified by the `connectionString` parameter. This method may be called more than once with different connection strings for multiple servers.

If no queue name is specified, [SqlDependency](#) creates a temporary queue and service in the server that is used for the entire process, even if the process involves more than one [AppDomain](#). The queue and service are automatically removed upon application shutdown.

For additional remarks, see [Start](#).

See

Also

[Using Query Notifications](#)

[ADO.NET Overview](#)

# SqlDependency.Stop SqlDependency.Stop

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">Stop(String) Stop(String)</a>                 | Stops a listener for a connection specified in a previous <a href="#">Start</a> call. |
| <a href="#">Stop(String, String) Stop(String, String)</a> | Stops a listener for a connection specified in a previous <a href="#">Start</a> call. |

## Remarks

The [SqlDependency](#) listener will restart when an error occurs in the SQL Server connection.

## Stop(String) Stop(String)

Stops a listener for a connection specified in a previous [Start](#) call.

```
public static bool Stop (string connectionString);  
static member Stop : string -> bool
```

Parameters

**connectionString** [String](#) [String](#)

Connection string for the instance of SQL Server that was used in a previous [Start\(String\)](#) call.

Returns

[Boolean](#) [Boolean](#)

**true** if the listener was completely stopped; **false** if the [AppDomain](#) was unbound from the listener, but there are at least one other [AppDomain](#) using the same listener.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The `connectionString` parameter is NULL.

[InvalidOperationException](#) [InvalidOperationException](#)

The method was called from within SQLCLR.

[SecurityException](#) [SecurityException](#)

The caller does not have the required [SqlClientPermission](#) code access security (CAS) permission.

[SqlException](#) [SqlException](#)

An underlying [SqlClient](#) exception occurred.

Remarks

The [Stop](#) method must be called for each [Start](#) call. A given listener only shuts down fully when it receives the same

number of [Stop](#) requests as [Start](#) requests.

See

Also

[Using Query Notifications](#)

[ADO.NET Overview](#)

## Stop(String, String) Stop(String, String)

Stops a listener for a connection specified in a previous [Start](#) call.

```
public static bool Stop (string connectionString, string queue);  
static member Stop : string * string -> bool
```

Parameters

connectionString [String](#) [String](#)

Connection string for the instance of SQL Server that was used in a previous [Start\(String, String\)](#) call.

queue [String](#) [String](#)

The SQL Server Service Broker queue that was used in a previous [Start\(String, String\)](#) call.

Returns

[Boolean](#) [Boolean](#)

**true** if the listener was completely stopped; **false** if the [AppDomain](#) was unbound from the listener, but there is at least one other [AppDomain](#) using the same listener.

Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The `connectionString` parameter is NULL.

[InvalidOperationException](#) [InvalidOperationException](#)

The method was called from within SQLCLR.

[SecurityException](#) [SecurityException](#)

The caller does not have the required [SqlClientPermission](#) code access security (CAS) permission.

[SQLException](#) [SQLException](#)

And underlying **SqlClient** exception occurred.

Remarks

The [Stop](#) method must be called for each [Start](#) call. A given listener only shuts down fully when it receives the same number of [Stop](#) requests as [Start](#) requests.

See

Also

[Using Query Notifications](#)

[ADO.NET Overview](#)

# SqlEnclaveAttestationParameters Class

Encapsulates the information SqlClient sends to SQL Server to initiate the process of attesting and creating a secure session with the enclave, SQL Server uses for computations on columns protected using Always Encrypted.

## Declaration

```
public class SqlEnclaveAttestationParameters  
type SqlEnclaveAttestationParameters = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

[SqlEnclaveAttestationParameters\(Int32, Byte\[\], ECDiffieHellmanCng\)](#)  
[SqlEnclaveAttestationParameters\(Int32, Byte\[\], ECDiffieHellmanCng\)](#)

Initializes a new instance of the [SqlEnclaveAttestationParameters](#) class.

## Properties

[ClientDiffieHellmanKey](#)

[ClientDiffieHellmanKey](#)

Gets a Diffie-Hellman algorithm that encapsulates a key pair that SqlClient uses to establish a secure session with the enclave.

[Protocol](#)

[Protocol](#)

Gets the enclave attestation protocol identifier.

## Methods

[GetInput\(\)](#)

[GetInput\(\)](#)

Gets the information used to initiate the process of attesting the enclave. The format and the content of this information is specific to the attestation protocol.

# SqlEnclaveAttestationParameters.ClientDiffieHellmanKey

## SqlEnclaveAttestationParameters.ClientDiffieHellmanKey

### In this Article

Gets a Diffie-Hellman algorithm that encapsulates a key pair that `SqlClient` uses to establish a secure session with the enclave.

```
public System.Security.Cryptography.ECDiffieHellmanCng ClientDiffieHellmanKey { get; }  
member this.ClientDiffieHellmanKey : System.Security.Cryptography.ECDiffieHellmanCng
```

### Returns

[ECDiffieHellmanCng](#) [ECDiffieHellmanCng](#)

The Diffie-Hellman algorithm.

# SqlEnclaveAttestationParameters.GetInput SqlEnclaveAttestationParameters.GetInput

## In this Article

Gets the information used to initiate the process of attesting the enclave. The format and the content of this information is specific to the attestation protocol.

```
public byte[] GetInput ();  
member this.GetInput : unit -> byte[]
```

## Returns

[Byte\[\]](#)

The information required by SQL Server to execute attestation protocol identified by [EnclaveAttestationProtocols](#).

# SqlEnclaveAttestationParameters.Protocol SqlEnclaveAttestationParameters.Protocol

## In this Article

Gets the enclave attestation protocol identifier.

```
public int Protocol { get; }  
member this.Protocol : int
```

Returns

[Int32 Int32](#)

The enclave attestation protocol identifier.

# SqlEnclaveAttestationParameters SqlEnclaveAttestationParameters

## In this Article

Initializes a new instance of the [SqlEnclaveAttestationParameters](#) class.

```
public SqlEnclaveAttestationParameters (int protocol, byte[] input,
System.Security.Cryptography.ECDiffieHellmanCng clientDiffieHellmanKey);

new System.Data.SqlClient.SqlEnclaveAttestationParameters : int * byte[] *
System.Security.Cryptography.ECDiffieHellmanCng ->
System.Data.SqlClient.SqlEnclaveAttestationParameters
```

## Parameters

protocol [Int32](#)

The enclave attestation protocol.

input [Byte\[\]](#)

The input of the enclave attestation protocol.

clientDiffieHellmanKey [ECDiffieHellmanCng](#)

A Diffie-Hellman algorithm that encapsulates a client-side key pair.

## Exceptions

[ArgumentNullException](#) [ArgumentException](#)

`clientDiffieHellmanKey` is `null`.

# SqlEnclaveSession SqlEnclaveSession Class

Encapsulates the state of a secure session between SqlClient and an enclave inside SQL Server, which can be used for computations on encrypted columns protected with Always Encrypted.

## Declaration

```
public class SqlEnclaveSession  
type SqlEnclaveSession = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Constructors

```
SqlEnclaveSession(Byte[], Int64)  
SqlEnclaveSession(Byte[], Int64)
```

Instantiates a new instance of the [SqlEnclaveSession](#) class.

## Properties

[SessionId](#)

[SessionId](#)

Gets the session ID.

## Methods

[GetSessionKey\(\)](#)

[GetSessionKey\(\)](#)

Gets the symmetric key that SqlClient uses to encrypt all the information it sends to the enclave using the session.

# SqlEnclaveSession.GetSessionKey SqlEnclaveSession.GetSessionKey

## In this Article

Gets the symmetric key that SqlClient uses to encrypt all the information it sends to the enclave using the session.

```
public byte[] GetSessionKey ();  
member this.GetSessionKey : unit -> byte[]
```

Returns

[Byte\[\]](#)

The symmetric key.

# SqlEnclaveSession.SessionId SqlEnclaveSession.SessionId

## In this Article

Gets the session ID.

```
public long SessionId { get; }  
member this.SessionId : int64
```

Returns

[Int64 Int64](#)

The session ID.

# SqlEnclaveSession SqlEnclaveSession

## In this Article

Instantiates a new instance of the [SqlEnclaveSession](#) class.

```
public SqlEnclaveSession (byte[] sessionKey, long sessionId);  
new System.Data.SqlClient.SqlEnclaveSession : byte[] * int64 ->  
System.Data.SqlClient.SqlEnclaveSession
```

## Parameters

sessionKey [Byte](#)[]

The symmetric key used to encrypt all the information sent using the session.

sessionId [Int64](#) [Int64](#)

The session ID.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

`sessionKey` is `null`.

[ArgumentException](#) [ArgumentException](#)

`sessionKey` has zero length.

# SqlError SqlError Class

Collects information relevant to a warning or error returned by SQL Server.

## Declaration

```
[Serializable]
public sealed class SqlError

type SqlError = class
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

This class is created by the .NET Framework Data Provider for SQL Server when an error occurs. An instance of [SqlError](#) is created and managed by the [SqlErrorCollection](#), which in turn is created by the [SQLException](#) class.

Messages with a severity level of 10 or less are informational and indicate problems caused by mistakes in information that a user has entered. Severity levels from 11 through 16 are generated by the user, and can be corrected by the user. Severity levels from 17 through 25 indicate software or hardware errors. When a level 17, 18, or 19 error occurs, you can continue working, although you might not be able to execute a particular statement.

The [SqlConnection](#) remains open when the severity level is 19 or less. When the severity level is 20 or greater, the server usually closes the [SqlConnection](#). However, the user can reopen the connection and continue. In both cases, a [SQLException](#) is generated by the method executing the command.

For more information on errors generated by SQL Server, see [Cause and Resolution of Database Engine Errors](#). For more information about severity levels, see [Database Engine Error Severities](#).

## Properties

[Class](#)

[Class](#)

Gets the severity level of the error returned from SQL Server.

[LineNumber](#)

[LineNumber](#)

Gets the line number within the Transact-SQL command batch or stored procedure that contains the error.

[Message](#)

[Message](#)

Gets the text describing the error.

[Number](#)

[Number](#)

Gets a number that identifies the type of error.

Procedure

Procedure

Gets the name of the stored procedure or remote procedure call (RPC) that generated the error.

Server

Server

Gets the name of the instance of SQL Server that generated the error.

Source

Source

Gets the name of the provider that generated the error.

State

State

Some error messages can be raised at multiple points in the code for the Database Engine. For example, an 1105 error can be raised for several different conditions. Each specific condition that raises an error assigns a unique state code.

## Methods

`ToString()`

`ToString()`

Gets the complete text of the error message.

## See Also

# SqlError Class

## In this Article

Gets the severity level of the error returned from SQL Server.

```
public byte Class { get; }  
member this.Class : byte
```

Returns

[Byte](#)

A value from 1 to 25 that indicates the severity level of the error. The default is 0.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

## Remarks

Messages with a severity level of 10 or less are informational and indicate problems caused by mistakes in information that a user has entered. Severity levels from 11 through 16 are generated by the user, and can be corrected by the user. Severity levels from 17 through 25 indicate software or hardware errors. When a level 17, 18, or 19 error occurs, you can continue working, although you might not be able to execute a particular statement.

The [SqlConnection](#) remains open when the severity level is 19 or less. When the severity level is 20 or greater, the server usually closes the [SqlConnection](#). However, the user can reopen the connection and continue. In both cases, a [SqlException](#) is generated by the method executing the command.

For more information on errors generated by SQL Server, see [Database Engine Events and Errors](#).

See

[ADO.NET Overview](#)

Also

# SqlError.LineNumber SqlError.LineNumber

## In this Article

Gets the line number within the Transact-SQL command batch or stored procedure that contains the error.

```
public int LineNumber { get; }  
member this.LineNumber : int
```

Returns

[Int32](#) [Int32](#)

The line number within the Transact-SQL command batch or stored procedure that contains the error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

## Remarks

Line numbering starts at 1. If the value is 0, the line number is not applicable.

For more information on errors generated by SQL Server, see [Database Engine Events and Errors](#).

See

[ADO.NET Overview](#)

Also

# SqlError.Message SqlError.Message

## In this Article

Gets the text describing the error.

```
public string Message { get; }  
member this.Message : string
```

Returns

[String String](#)

The text describing the error. For more information on errors generated by SQL Server, see [Database Engine Events and Errors](#).

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

See

[ADO.NET Overview](#)

Also

# SqlError.Number SqlError.Number

## In this Article

Gets a number that identifies the type of error.

```
public int Number { get; }  
member this.Number : int
```

Returns

[Int32](#) [Int32](#)

The number that identifies the type of error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

## Remarks

The following table describes the possible values for this property:

| SOURCE OF ERROR   | SQLERROR.NUMBER   | SQLERROR.STATE           | SQLEXCEPTION HAS INNER WIN32EXCEPTION (BEGINNING WITH .NET FRAMEWORK 4.5) |
|-------------------|---|--------------------------|---|
| Error from server | Server error code<br><br>This number corresponds to an entry in the <a href="#">master.dbo.sysmessages</a> table. | Typically greater than 0 | No  |

| Source of Error                              | SQLERROR.Number  | SQLERROR.State | SQLEXCEPTION HAS INNER WIN32EXCEPTION (BEGINNING WITH .NET FRAMEWORK 4.5) |
|--|------------------|----------------|---|
| Connection timeout                           | -2               | 0              | Yes (Number = 258)  |
| Communication error (non-LocalDB)            | Win32 error code | 0              | Yes (Number = Win32 error code)   |
| Communication error (LocalDB)                | Win32 error code | 0              | No  |
| Encryption capability mismatch               | 20               | 0              | No  |
| Failed to start LocalDB                      | Win32 error code | 0              | No  |
| Read-only routing failure                    | 0                | 0              | No  |
| Server had severe error processing query     | 0                | 0              | No  |
| Processed cancellation while parsing results | 0                | 0              | No  |
| Failed to create user instance               | 0                | 0              | No  |

For more information on errors generated by SQL Server, see [Database Engine Events and Errors](#).

See

[ADO.NET Overview](#)

Also

# SqlError.Procedure SqlError.Procedure

## In this Article

Gets the name of the stored procedure or remote procedure call (RPC) that generated the error.

```
public string Procedure { get; }  
member this.Procedure : string
```

Returns

[String String](#)

The name of the stored procedure or RPC. For more information on errors generated by SQL Server, see [Database Engine Events and Errors](#).

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

See

[ADO.NET Overview](#)

Also

# SqlError.Server SqlError.Server

## In this Article

Gets the name of the instance of SQL Server that generated the error.

```
public string Server { get; }  
member this.Server : string
```

Returns

[String](#) [String](#)

The name of the instance of SQL Server.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

See

[ADO.NET Overview](#)

Also

# SqlError.Source SqlError.Source

## In this Article

Gets the name of the provider that generated the error.

```
public string Source { get; }  
member this.Source : string
```

Returns

[String String](#)

The name of the provider that generated the error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

See

[ADO.NET Overview](#)

Also

# SqlError.State SqlError.State

## In this Article

Some error messages can be raised at multiple points in the code for the Database Engine. For example, an 1105 error can be raised for several different conditions. Each specific condition that raises an error assigns a unique state code.

```
public byte State { get; }  
member this.State : byte
```

Returns

[Byte](#)

The state code.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Source: " + exception.Errors[i].Source + "  
" +  
            "Number: " + exception.Errors[i].Number.ToString() + "  
" +  
            "State: " + exception.Errors[i].State.ToString() + "  
" +  
            "Class: " + exception.Errors[i].Class.ToString() + "  
" +  
            "Server: " + exception.Errors[i].Server + "  
" +  
            "Message: " + exception.Errors[i].Message + "  
" +  
            "Procedure: " + exception.Errors[i].Procedure + "  
" +  
            "LineNumber: " + exception.Errors[i].LineNumber.ToString());  
    }  
    Console.ReadLine();  
}
```

## Remarks

State is only set for errors that are received from the server.

For more information on errors generated by SQL Server, see [Understanding Database Engine Errors](#).

See

[ADO.NET Overview](#)

Also

# SqlError.ToString SqlError.ToString

## In this Article

Gets the complete text of the error message.

```
public override string ToString ();
override this.ToString : unit -> string
```

Returns

[String](#) [String](#)

The complete text of the error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public static void ShowSqlException(string connectionString)
{
    string queryString = "EXECUTE NonExistantStoredProcedure";

    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        SqlCommand command = new SqlCommand(queryString, connection);
        try
        {
            command.Connection.Open();
            command.ExecuteNonQuery();
        }
        catch (SqlException ex)
        {
            DisplaySqlErrors(ex);
        }
    }
}

private static void DisplaySqlErrors(SqlException exception)
{
    for (int i = 0; i < exception.Errors.Count; i++)
    {
        Console.WriteLine("Index #" + i + "
" +
                    "Error: " + exception.Errors[i].ToString() +
    });
    Console.ReadLine();
}
```

## Remarks

The string is in the form "SqlError:", followed by the [Message](#), and the stack trace. For example:

SqlError:User Id or Password not valid. <stack trace>

See

[ADO.NET Overview](#)

Also

# SqlErrorCollection SqlErrorCollection Class

Collects all errors generated by the .NET Framework Data Provider for SQL Server. This class cannot be inherited.

## Declaration

```
[System.ComponentModel.ListBindable(false)]
[Serializable]
public sealed class SqlErrorCollection : System.Collections.ICollection

type SqlErrorCollection = class
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

[Object](#) [Object](#)

## Remarks

This class is created by [SQLException](#) to collect instances of the [SqlError](#) class. [SqlErrorCollection](#) always contains at least one instance of the [SqlError](#) class.

## Properties

Count

Count

Gets the number of errors in the collection.

Item[Int32]

Item[Int32]

Gets the error at the specified index.

## Methods

CopyTo(Array, Int32)

CopyTo(Array, Int32)

Copies the elements of the [SqlErrorCollection](#) collection into an [Array](#), starting at the specified index.

CopyTo(SqlError[], Int32)

CopyTo(SqlError[], Int32)

Copies the elements of the [SqlErrorCollection](#) collection into a [SqlErrorCollection](#), starting at the specified index.

GetEnumerator()

GetEnumerator()

Returns an enumerator that iterates through the [SqlErrorCollection](#).

[ICollection.IsSynchronized](#)

[ICollection.IsSynchronized](#)

For a description of this member, see [IsSynchronized](#).

[ICollection.SyncRoot](#)

[ICollection.SyncRoot](#)

For a description of this member, see [SyncRoot](#).

## See Also

[SqlError](#) [SqlError](#)

[SqlError](#) [SqlError](#)

# SqlErrorCollection.CopyTo SqlErrorCollection.CopyTo

In this Article

## Overloads

[CopyTo\(Array, Int32\)](#) [CopyTo\(Array, Int32\)](#)

Copies the elements of the [SqlErrorCollection](#) collection into an [Array](#), starting at the specified index.

[CopyTo\(SqlError\[\], Int32\)](#) [CopyTo\(SqlError\[\], Int32\)](#)

Copies the elements of the [SqlErrorCollection](#) collection into a [SqlErrorCollection](#), starting at the specified index.

## CopyTo(Array, Int32) CopyTo(Array, Int32)

Copies the elements of the [SqlErrorCollection](#) collection into an [Array](#), starting at the specified index.

```
public void CopyTo (Array array, int index);  
abstract member CopyTo : Array * int -> unit  
override this.CopyTo : Array * int -> unit
```

### Parameters

array [Array](#) [Array](#)

The [Array](#) to copy elements into.

index [Int32](#) [Int32](#)

The index from which to start copying into the `array` parameter.

### Exceptions

[ArgumentException](#) [ArgumentException](#)

The sum of `index` and the number of elements in the [SqlErrorCollection](#) collection is greater than the [Length](#) of the [Array](#).

[ArgumentNullException](#) [ArgumentNullException](#)

The `array` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

The `index` is not valid for `array`.

[ADO.NET Overview](#)

See

Also

## CopyTo(SqlError[], Int32) CopyTo(SqlError[], Int32)

Copies the elements of the [SqlErrorCollection](#) collection into a [SqlErrorCollection](#), starting at the specified index.

```
public void CopyTo (System.Data.SqlClient.SqlError[] array, int index);  
member this.CopyTo : System.Data.SqlClient.SqlError[] * int -> unit
```

## Parameters

array [SqlError\[\]](#)

The [SqlErrorCollection](#) to copy the elements into.

index [Int32 Int32](#)

The index from which to start copying into the `array` parameter.

## Exceptions

[ArgumentException](#) [ArgumentException](#)

The sum of `index` and the number of elements in the [SqlErrorCollection](#) collection is greater than the length of the [SqlErrorCollection](#).

[ArgumentNullException](#) [ArgumentNullException](#)

The `array` is `null`.

[ArgumentOutOfRangeException](#) [ArgumentOutOfRangeException](#)

The `index` is not valid for `array`.

## See

[ADO.NET Overview](#)

## Also

# SqlErrorCollection.Count SqlErrorCollection.Count

## In this Article

Gets the number of errors in the collection.

```
public int Count { get; }  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The total number of errors in the collection.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public static void ShowSqlException(string connectionString)  
{  
    string queryString = "EXECUTE NonExistantStoredProcedure";  
  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        SqlCommand command = new SqlCommand(queryString, connection);  
        try  
        {  
            command.Connection.Open();  
            command.ExecuteNonQuery();  
        }  
        catch (SqlException ex)  
        {  
            DisplaySqlErrors(ex);  
        }  
    }  
}  
  
private static void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
                           "Error: " + exception.Errors[i].ToString() + "  
");  
    }  
    Console.ReadLine();  
}
```

See

[ADO.NET Overview](#)

Also

# SqlErrorCollection.GetEnumerator SqlErrorCollection.Get Enumerator

## In this Article

Returns an enumerator that iterates through the [SqlErrorCollection](#).

```
public System.Collections.IEnumerator GetEnumerator ();  
  
abstract member GetEnumerator : unit -> System.Collections.IEnumerator  
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [SqlErrorCollection](#).

## Remarks

Enumerators can be used to read the data in a collection, but they cannot be used to modify the underlying collection.

See

[ADO.NET Overview](#)

Also

# SqlErrorCollection.ICollection.IsSynchronized

## In this Article

For a description of this member, see [IsSynchronized](#).

```
bool System.Collections.ICollection.IsSynchronized { get; }
```

Returns

[Boolean](#)

`true` if access to the [ICollection](#) is synchronized (thread safe); otherwise, `false`.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlErrorCollection](#) instance is cast to an [ICollection](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlErrorCollection.ICollection.SyncRoot

## In this Article

For a description of this member, see [SyncRoot](#).

```
object System.Collections.ICollection.SyncRoot { get; }
```

Returns

[Object](#)

An object that can be used to synchronize access to the [ICollection](#).

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlErrorCollection](#) instance is cast to an [ICollection](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlErrorCollection.Item[Int32] SqlErrorCollection.Item[Int32]

## In this Article

Gets the error at the specified index.

```
public System.Data.SqlClient.SqlError this[int index] { get; }  
member this.Item(int) : System.Data.SqlClient.SqlError
```

## Parameters

index Int32 Int32

The zero-based index of the error to retrieve.

## Returns

[SqlError](#) [SqlError](#)

A [SqlError](#) that contains the error at the specified index.

## Exceptions

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

Index parameter is outside array bounds.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public static void ShowSqlException(string connectionString)  
{  
    string queryString = "EXECUTE NonExistantStoredProcedure";  
  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        SqlCommand command = new SqlCommand(queryString, connection);  
        try  
        {  
            command.Connection.Open();  
            command.ExecuteNonQuery();  
        }  
        catch (SqlException ex)  
        {  
            DisplaySqlErrors(ex);  
        }  
    }  
}  
  
private static void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Error: " + exception.Errors[i].ToString() + "  
");  
    }  
    Console.ReadLine();  
}
```

See

Also

CountCount

ADO.NET Overview

# SqlException SqlException Class

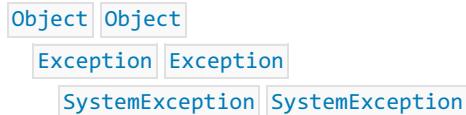
The exception that is thrown when SQL Server returns a warning or error. This class cannot be inherited.

## Declaration

```
[Serializable]
public sealed class SqlException : System.Data.Common.DbException

type SqlException = class
    inherit DbException
```

## Inheritance Hierarchy



## Remarks

This class is created whenever the .NET Framework Data Provider for SQL Server encounters an error generated from the server. (Client side errors are thrown as standard common language runtime exceptions.) `SqlException` always contains at least one instance of `SqlError`.

Messages that have a severity level of 10 or less are informational and indicate problems caused by mistakes in information that a user has entered. Severity levels from 11 through 16 are generated by the user, and can be corrected by the user. Severity levels from 17 through 25 indicate software or hardware errors. When a level 17, 18, or 19 error occurs, you can continue working, although you might not be able to execute a particular statement.

The `SqlConnection` remains open when the severity level is 19 or less. When the severity level is 20 or greater, the server ordinarily closes the `SqlConnection`. However, the user can reopen the connection and continue. In both cases, a `SqlException` is generated by the method executing the command.

For information about the warning and informational messages sent by SQL Server, see [Database Engine Events and Errors](#). The `SqlException` class maps to SQL Server severity.

The following is general information on handling exceptions. Your code should catch exceptions to prevent the application from crashing and to allow displaying a relevant error message to the user. You can use database transactions to ensure that the data is consistent regardless of what happens in the client application (including a crash). Features like `System.Transaction.TransactionScope` or the `BeginTransaction` method (in `System.Data.OleDb.OleDbConnection`, `System.Data.ODBC.ODBCConnection`, and `System.Data.SqlClient.SqlConnection`) ensure consistent data regardless of exceptions raised by a provider. Transactions can fail, so catch failures and retry the transaction.

Note that beginning with .NET Framework 4.5, `SqlException` can return an inner `Win32Exception`.

The exception class of a .Net Framework data provider reports provider-specific errors. For example `System.Data.Odbc` has `OdbcException`, `System.Data.OleDb` has `OleDbException`, and `System.Data.SqlClient` has `SqlException`. For the best level of error detail, catch these exceptions and use the members of these exception classes to get details of the error.

In addition to the provider-specific errors, .NET Framework data provider types can raise .NET Framework exceptions such as `System.OutOfMemoryException` and `System.Threading.ThreadAbortException`. Recovery from these exceptions may not be possible.

Bad input can cause a .NET Framework data provider type to raise an exception such as `System.ArgumentException` or `System.IndexOutOfRangeException`. Calling a method at the wrong time can raise `System.InvalidOperationException`.

So, in general, write an exception handler that catches any provider specific exceptions as well as exceptions from the common language runtime. These can be layered as follows:

```
try {
    // code here
}
catch (SqlException odbcEx) {
    // Handle more specific SqlException exception here.
}
catch (Exception ex) {
    // Handle generic ones here.
}
```

Or:

```
try {
    // code here
}
catch (Exception ex) {
    if (ex is SqlException) {
        // Handle more specific SqlException exception here.
    }
    else {
        // Handle generic ones here.
    }
}
```

It is also possible for a .NET Framework data provider method call to fail on a thread pool thread with no user code on the stack. In this case, and when using asynchronous method calls, you must register the [UnhandledException](#) event to handle those exceptions and avoid application crash.

## Properties

Class

Class

Gets the severity level of the error returned from the .NET Framework Data Provider for SQL Server.

ClientConnectionId

ClientConnectionId

Represents the client connection ID. For more information, see [Data Tracing in ADO.NET](#).

Errors

Errors

Gets a collection of one or more [SqlError](#) objects that give detailed information about exceptions generated by the .NET Framework Data Provider for SQL Server.

LineNumber

LineNumber

Gets the line number within the Transact-SQL command batch or stored procedure that generated the error.

**Message**

**Message**

**Number**

**Number**

Gets a number that identifies the type of error.

**Procedure**

**Procedure**

Gets the name of the stored procedure or remote procedure call (RPC) that generated the error.

**Server**

**Server**

Gets the name of the computer that is running an instance of SQL Server that generated the error.

**Source**

**Source**

Gets the name of the provider that generated the error.

**State**

**State**

Gets a numeric error code from SQL Server that represents an error, warning or "no data found" message. For more information about how to decode these values, see [Database Engine Events and Errors](#).

## Methods

`GetObjectData(SerializationInfo, StreamingContext)`

`GetObjectData(SerializationInfo, StreamingContext)`

Sets the `SerializationInfo` with information about the exception.

`ToString()`

`ToString()`

Returns a string that represents the current `SqlException` object, and includes the client connection ID (for more information, see [ClientConnectionId](#)).

## See Also

`SqlErrorCollection`

`SqlErrorCollection`

# SqlException.Class SqlException.Class

## In this Article

Gets the severity level of the error returned from the .NET Framework Data Provider for SQL Server.

```
public byte Class { get; }  
member this.Class : byte
```

Returns

[Byte](#) [Byte](#)

A value from 1 to 25 that indicates the severity level of the error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
               "Integrated Security=SSPI";
    }
}

```

## Remarks

Messages that have a severity level of 10 or less are informational and indicate problems caused by mistakes in information that a user has entered. Severity levels from 11 through 16 are generated by the user, and can be corrected by the user. Severity levels from 17 through 25 indicate software or hardware errors. When a level 17, 18, or 19 error

occurs, you can continue working, although you might not be able to execute a particular statement.

The [SqlConnection](#) remains open when the severity level is 19 or less. When the severity level is 20 or greater, the server ordinarily closes the [SqlConnection](#). However, the user can reopen the connection and continue. In both cases, a [SqlException](#) is generated by the method executing the command.

For information about the warning and informational messages sent by SQL Server, see the Troubleshooting section of the SQL Server documentation.

This is a wrapper for the [Class](#) property of the first [SqlError](#) in the [Errors](#) property.

See

[Number](#)

Also

[Source](#)

[State](#)

[Server](#)

[Procedure](#)

[LineNumber](#)

[ADO.NET Overview](#)

# SqlException.ClientConnectionId SqlException.ClientConnectionId

## In this Article

Represents the client connection ID. For more information, see [Data Tracing in ADO.NET](#).

```
public Guid ClientConnectionId { get; }  
member this.ClientConnectionId : Guid
```

Returns

[Guid](#) [Guid](#)

Returns the client connection ID.

## Remarks

For a code sample, see [ToString](#).

# SqlException.Errors SqlException.Errors

## In this Article

Gets a collection of one or more [SqlError](#) objects that give detailed information about exceptions generated by the .NET Framework Data Provider for SQL Server.

```
public System.Data.SqlClient.SqlErrorCollection Errors { get; }  
member this.Errors : System.Data.SqlClient.SqlErrorCollection
```

Returns

[SqlErrorCollection](#) [SqlErrorCollection](#)

The collected instances of the [SqlError](#) class.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```
public static void ShowSqlException(string connectionString)  
{  
    string queryString = "EXECUTE NonExistantStoredProcedure";  
  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        SqlCommand command = new SqlCommand(queryString, connection);  
        try  
        {  
            command.Connection.Open();  
            command.ExecuteNonQuery();  
        }  
        catch (SqlException ex)  
        {  
            DisplaySqlErrors(ex);  
        }  
    }  
}  
  
private static void DisplaySqlErrors(SqlException exception)  
{  
    for (int i = 0; i < exception.Errors.Count; i++)  
    {  
        Console.WriteLine("Index #" + i + "  
" +  
            "Error: " + exception.Errors[i].ToString() + "  
");  
    }  
    Console.ReadLine();  
}
```

## Remarks

The [SqlErrorCollection](#) class always contains at least one instance of the [SqlError](#) class.

This is a wrapper for [SqlErrorCollection](#). For more information on SQL Server engine errors, see [Database Engine Events and Errors](#).

See

[SqlErrorCollection](#)  
[SqlErrorCollection](#)

Also

[ADO.NET Overview](#)

# SqlException.GetObjectData SqlException.GetObjectData

## In this Article

Sets the [SerializationInfo](#) with information about the exception.

```
public override void GetObjectData (System.Runtime.Serialization.SerializationInfo si,
System.Runtime.Serialization.StreamingContext context);

override this.GetObjectData : System.Runtime.Serialization.SerializationInfo *
System.Runtime.Serialization.StreamingContext -> unit
```

## Parameters

si [SerializationInfo](#) [SerializationInfo](#)

The [SerializationInfo](#) that holds the serialized object data about the exception being thrown.

context [StreamingContext](#) [StreamingContext](#)

The [StreamingContext](#) that contains contextual information about the source or destination.

## Exceptions

[ArgumentNullException](#) [ArgumentNullException](#)

The `si` parameter is a null reference (`Nothing` in Visual Basic).

## Remarks

`GetObjectData` sets a `SerializationInfo` with all the exception object data targeted for serialization. During deserialization, the exception is reconstituted from the `SerializationInfo` transmitted over the stream.

See

[ADO.NET Overview](#)

Also

# SqlException.LineNumber SqlException.LineNumber

## In this Article

Gets the line number within the Transact-SQL command batch or stored procedure that generated the error.

```
public int LineNumber { get; }  
member this.LineNumber : int
```

Returns

[Int32 Int32](#)

The line number within the Transact-SQL command batch or stored procedure that generated the error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
            "Integrated Security=SSPI";
    }
}

```

## Remarks

The line numbering starts at 1; if 0 is returned, the line number is not applicable.

This is a wrapper for the [LineNumber](#) property of the first [SqlError](#) in the [Errors](#) property.

See

Also

[NumberNumber](#)

[SourceSource](#)

[StateState](#)

[ClassClass](#)

[ServerServer](#)

[ProcedureProcedure](#)

[ADO.NET Overview](#)

# SqlException.Message SqlException.Message

## In this Article

```
public override string Message { get; }  
member this.Message : string
```

## Returns

[String](#) [String](#)

# SqlException.Number SqlException.Number

## In this Article

Gets a number that identifies the type of error.

```
public int Number { get; }  
member this.Number : int
```

Returns

[Int32](#) [Int32](#)

The number that identifies the type of error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
            "Integrated Security=SSPI";
    }
}

```

## Remarks

This is a wrapper for the [Number](#) property of the first [SqlError](#) in the [Errors](#) property. For more information on SQL Server engine errors, see [Database Engine Events and Errors](#).

See

Also

[State](#)[State](#)

[Class](#)[Class](#)

[Source](#)[Source](#)

[Server](#)[Server](#)

[Procedure](#)[Procedure](#)

[LineNumber](#)[LineNumber](#)

[ADO.NET Overview](#)

# SqlException.Procedure SqlException.Procedure

## In this Article

Gets the name of the stored procedure or remote procedure call (RPC) that generated the error.

```
public string Procedure { get; }  
member this.Procedure : string
```

Returns

[String String](#)

The name of the stored procedure or RPC.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
            "Integrated Security=SSPI";
    }
}

```

## Remarks

This is a wrapper for the [Procedure](#) property of the first [SqlError](#) in the [Errors](#) property.

See

[NumberNumber](#)

Also

[State](#)  
[Class](#)  
[Server](#)  
[Source](#)  
[LineNumber](#)  
[ADO.NET Overview](#)

# SqlException.Server SqlException.Server

## In this Article

Gets the name of the computer that is running an instance of SQL Server that generated the error.

```
public string Server { get; }  
member this.Server : string
```

Returns

[String String](#)

The name of the computer running an instance of SQL Server.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
               "Integrated Security=SSPI";
    }
}

```

## Remarks

This is a wrapper for the [Server](#) property of the first [SqlError](#) in the [Errors](#) property.

See

[NumberNumber](#)

Also

[State](#)  
[Source](#)  
[Class](#)  
[Procedure](#)  
[LineNumber](#)  
[ADO.NET Overview](#)

# SqlException.Source SqlException.Source

## In this Article

Gets the name of the provider that generated the error.

```
public override string Source { get; }  
member this.Source : string
```

Returns

[String String](#)

The name of the provider that generated the error.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
               "Integrated Security=SSPI";
    }
}

```

## Remarks

This is a wrapper for the [Source](#) property of the first [SqlError](#) in the [Errors](#) property.

See

[NumberNumber](#)

Also

[State](#)  
[Class](#)  
[Server](#)  
[Procedure](#)  
[LineNumber](#)  
[ADO.NET Overview](#)

# SqlException.State SqlException.State

## In this Article

Gets a numeric error code from SQL Server that represents an error, warning or "no data found" message. For more information about how to decode these values, see [Database Engine Events and Errors](#).

```
public byte State { get; }  
member this.State : byte
```

Returns

[Byte Byte](#)

The number representing the error code.

## Examples

The following example displays each [SqlError](#) within the [SqlErrorCollection](#) collection.

```

using System;
using System.Data;
using System.Data.SqlClient;
using System.Text;

class Program
{
    static void Main()
    {
        string s = GetConnectionString();
        ShowSqlException(s);
        Console.ReadLine();
    }

    public static void ShowSqlException(string connectionString)
    {
        string queryString = "EXECUTE NonExistantStoredProcedure";
        StringBuilder errorMessages = new StringBuilder();

        using (SqlConnection connection = new SqlConnection(connectionString))
        {
            SqlCommand command = new SqlCommand(queryString, connection);
            try
            {
                command.Connection.Open();
                command.ExecuteNonQuery();
            }
            catch (SqlException ex)
            {
                for (int i = 0; i < ex.Errors.Count; i++)
                {
                    errorMessages.Append("Index #" + i + "
" +
                        "Message: " + ex.Errors[i].Message + "
" +
                        "Error Number: " + ex.Errors[i].Number + "
" +
                        "LineNumber: " + ex.Errors[i].LineNumber + "
" +
                        "Source: " + ex.Errors[i].Source + "
" +
                        "Procedure: " + ex.Errors[i].Procedure + "
");
                }
                Console.WriteLine(errorMessages.ToString());
            }
        }
    }

    static private string GetConnectionString()
    {
        // To avoid storing the connection string in your code,
        // you can retrieve it from a configuration file.
        return "Data Source=(local);Initial Catalog=AdventureWorks;" +
               "Integrated Security=SSPI";
    }
}

```

## Remarks

This is a wrapper for the [State](#) property of the first [SqlError](#) in the [Errors](#) property.

See

[NumberNumber](#)

Also

[Class Class](#)  
[Source Source](#)  
[Server Server](#)  
[Procedure Procedure](#)  
[Line Number Line Number](#)  
[ADO.NET Overview](#)

# SqlException.ToString SqlException.ToString

## In this Article

Returns a string that represents the current [SqlException](#) object, and includes the client connection ID (for more information, see [ClientConnectionId](#)).

```
public override string ToString ();
override this.ToString : unit -> string
```

Returns

[String](#) [String](#)

A string that represents the current [SqlException](#) object.[String](#).

## Examples

The following C# example shows how a connection attempt to a valid server but non-existent database causes a [SqlException](#), which includes the client connection ID:

```
using System.Data.SqlClient;
using System;

public class A {
    public static void Main() {
        SqlConnection connection = new SqlConnection();
        connection.ConnectionString = "Data Source=a_valid_server;Initial
Catalog=Northwinda;Integrated Security=true";
        try {
            connection.Open();
        }
        catch (SqlException p) {
            Console.WriteLine("{0}", p.ClientConnectionId);
            Console.WriteLine("{0}", p.ToString());
        }
        connection.Close();
    }
}
```

The following Visual Basic sample is functionally equivalent to the previous (C#) sample:

# SqlInfoMessageEventArgs Class

Provides data for the [InfoMessage](#) event.

## Declaration

```
public sealed class SqlInfoMessageEventArgs : EventArgs  
type SqlInfoMessageEventArgs = class  
    inherit EventArgs
```

## Inheritance Hierarchy

```
Object | Object  
       | EventArgs | EventArgs
```

## Remarks

The [InfoMessage](#) event contains a [SqlErrorCollection](#) collection which contains the warnings sent from the server.

An [InfoMessage](#) event is generated when a SQL Server message with a severity level of 10 or less occurs.

## Properties

Errors

Errors

Gets the collection of warnings sent from the server.

Message

Message

Gets the full text of the error sent from the database.

Source

Source

Gets the name of the object that generated the error.

## Methods

ToString()

ToString()

Retrieves a string representation of the [InfoMessage](#) event.

## See Also

# SqlInfoMessageEventArgs.Errors SqlInfoMessageEventArgs.Errors

## In this Article

Gets the collection of warnings sent from the server.

```
public System.Data.SqlClient.SqlErrorCollection Errors { get; }  
member this.Errors : System.Data.SqlClient.SqlErrorCollection
```

## Returns

[SqlErrorCollection](#) [SqlErrorCollection](#)

The collection of warnings sent from the server.

See

[ADO.NET Overview](#)

Also

# SqlInfoMessageEventArgs.Message SqlInfoMessageEventArgs.Message

## In this Article

Gets the full text of the error sent from the database.

```
public string Message { get; }  
member this.Message : string
```

Returns

[String](#) [String](#)

The full text of the error.

## Remarks

This is a wrapper for the [Message](#) property of the first [SqlError](#) in the [Errors](#) collection.

See

[ADO.NET Overview](#)

Also

# SqlInfoMessageEventArgs.Source SqlInfoMessageEventArgs.Source

## In this Article

Gets the name of the object that generated the error.

```
public string Source { get; }  
member this.Source : string
```

Returns

[String](#) [String](#)

The name of the object that generated the error.

## Remarks

This is a wrapper for the [Source](#) property of the first [SqlError](#) in the [Errors](#) collection.

See

[ADO.NET Overview](#)

Also

# SqlInfoMessageEventArgs.ToString SqlInfoMessageEventArgs.ToString

## In this Article

Retrieves a string representation of the [InfoMessage](#) event.

```
public override string ToString ();
override this.ToString : unit -> string
```

Returns

[String](#)

A string representing the [InfoMessage](#) event.

See

[ADO.NET Overview](#)

Also

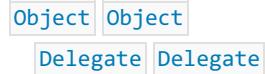
# SqlInfoMessageEventHandler SqlInfoMessageEvent Handler Delegate

Represents the method that will handle the [InfoMessage](#) event of a [SqlConnection](#).

## Declaration

```
public delegate void SqlInfoMessageEventHandler(object sender, SqlInfoMessageEventArgs e);  
type SqlInfoMessageEventHandler = delegate of obj * SqlInfoMessageEventArgs -> unit
```

## Inheritance Hierarchy



## Remarks

When you create a [SqlInfoMessageEventArgs](#) delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, unless you remove the delegate. For more information about event handler delegates, see [Handling and Raising Events](#).

## See Also

# SqlNotificationEventArgs SqlNotificationEventArgs Class

Represents the set of arguments passed to the notification event handler.

## Declaration

```
public class SqlNotificationEventArgs : EventArgs  
type SqlNotificationEventArgs = class  
    inherit EventArgs
```

## Inheritance Hierarchy

```
Object Object  
EventArgs EventArgs
```

## Constructors

`SqlNotificationEventArgs(SqlNotificationType, SqlNotificationInfo, SqlNotificationSource)`

`SqlNotificationEventArgs(SqlNotificationType, SqlNotificationInfo, SqlNotificationSource)`

Creates a new instance of the `SqlNotificationEventArgs` object.

## Properties

Info

Info

Gets a value that indicates the reason for the notification event, such as a row in the database being modified or a table being truncated.

Source

Source

Gets a value that indicates the source that generated the notification, such as a change to the query data or the database's state.

Type

Type

Gets a value that indicates whether this notification is generated because of an actual change, or by the subscription.

## See Also

# SqlNotificationEventArgs.Info SqlNotificationEventArgs.Info

## In this Article

Gets a value that indicates the reason for the notification event, such as a row in the database being modified or a table being truncated.

```
public System.Data.SqlClient.SqlNotificationInfo Info { get; }  
member this.Info : System.Data.SqlClient.SqlNotificationInfo
```

## Returns

[SqlNotificationInfo](#) [SqlNotificationInfo](#)

The notification event reason.

## Remarks

This event may occur because the data in the store actually changed, or the notification is no longer valid (for example, it timed out).

See

[ADO.NET Overview](#)

Also

# SqlNotificationEventArgs.Source SqlNotificationEventArgs.Source

## In this Article

Gets a value that indicates the source that generated the notification, such as a change to the query data or the database's state.

```
public System.Data.SqlClient.SqlNotificationSource Source { get; }  
member this.Source : System.Data.SqlClient.SqlNotificationSource
```

## Returns

[SqlNotificationSource](#) [SqlNotificationSource](#)

The source of the notification.

## See

[ADO.NET Overview](#)

## Also

# SqlNotificationEventArgs

## In this Article

Creates a new instance of the [SqlNotificationEventArgs](#) object.

```
public SqlNotificationEventArgs (System.Data.SqlClient.SqlNotificationType type,
System.Data.SqlClient.SqlNotificationInfo info, System.Data.SqlClient.SqlNotificationSource source);
new System.Data.SqlClient.SqlNotificationEventArgs : System.Data.SqlClient.SqlNotificationType *
System.Data.SqlClient.SqlNotificationInfo * System.Data.SqlClient.SqlNotificationSource ->
System.Data.SqlClient.SqlNotificationEventArgs
```

## Parameters

type [SqlNotificationType](#) [SqlNotificationType](#)

[SqlNotificationType](#) value that indicates whether this notification is generated because of an actual change, or by the subscription.

info [SqlNotificationInfo](#) [SqlNotificationInfo](#)

[SqlNotificationInfo](#) value that indicates the reason for the notification event. This may occur because the data in the store actually changed, or the notification became invalid (for example, it timed out).

source [SqlNotificationSource](#) [SqlNotificationSource](#)

[SqlNotificationSource](#) value that indicates the source that generated the notification.

See [ADO.NET Overview](#)

Also

# SqlNotificationEventArgs.Type SqlNotificationEventArgs.Type

## In this Article

Gets a value that indicates whether this notification is generated because of an actual change, or by the subscription.

```
public System.Data.SqlClient.SqlNotificationType Type { get; }  
member this.Type : System.Data.SqlClient.SqlNotificationType
```

## Returns

[SqlNotificationType](#) [SqlNotificationType](#)

A value indicating whether the notification was generated by a change or a subscription.

See

[ADO.NET Overview](#)

Also

# SqlNotificationInfo SqlNotificationInfo Enum

This enumeration provides additional information about the different notifications that can be received by the dependency event handler.

## Declaration

```
public enum SqlNotificationInfo  
type SqlNotificationInfo =
```

## Inheritance Hierarchy



## Remarks

The [SqlNotificationInfo](#) enumeration is referenced by an instance of the [SqlNotificationEventArgs](#) class.

## Fields

`AlreadyChanged`  
`AlreadyChanged`

The [SqlDependency](#) object already fired, and new commands cannot be added to it.

`Alter`  
`Alter`

An underlying server object related to the query was modified.

`Delete`  
`Delete`

Data was changed by a DELETE statement.

`Drop`  
`Drop`

An underlying object related to the query was dropped.

`Error`  
`Error`

An internal server error occurred.

`Expired`  
`Expired`

The [SqlDependency](#) object has expired.

`Insert`  
`Insert`

Data was changed by an INSERT statement.

`Invalid`  
`Invalid`

A statement was provided that cannot be notified (for example, an UPDATE statement).

`Isolation`  
`Isolation`

The statement was executed under an isolation mode that was not valid (for example, Snapshot).

Merge Merge

Used to distinguish the server-side cause for a query notification firing.

Options Options

The SET options were not set appropriately at subscription time.

PreviousFire  
PreviousFire

A previous statement has caused query notifications to fire under the current transaction.

Query Query

A SELECT statement that cannot be notified or was provided.

Resource Resource

Fires as a result of server resource pressure.

Restart Restart

The server was restarted (notifications are sent during restart.).

TemplateLimit  
TemplateLimit

The subscribing query causes the number of templates on one of the target tables to exceed the maximum allowable limit.

Truncate Truncate

One or more tables were truncated.

Unknown Unknown

Used when the info option sent by the server was not recognized by the client.

Update Update

Data was changed by an UPDATE statement.

## See Also

# SqlNotificationSource SqlNotificationSource Enum

Indicates the source of the notification received by the dependency event handler.

## Declaration

```
public enum SqlNotificationSource  
type SqlNotificationSource =
```

## Inheritance Hierarchy



## Remarks

The `SqlNotificationSource` enumeration is referenced by an instance of the [SqlNotificationEventArgs](#) class.

Query notifications are supported only for SELECT statements that meet a list of specific requirements. For more information, see [SQL Server Service Broker](#) and [Working with Query Notifications](#).

## Fields

|        |        |
|--------|--------|
| Client | Client |
|--------|--------|

A client-initiated notification occurred, such as a client-side time-out or as a result of attempting to add a command to a dependency that has already fired.

|      |      |
|------|------|
| Data | Data |
|------|------|

Data has changed; for example, an insert, update, delete, or truncate operation occurred.

|          |          |
|----------|----------|
| Database | Database |
|----------|----------|

The database state changed; for example, the database related to the query was dropped or detached.

|             |             |
|-------------|-------------|
| Environment | Environment |
|-------------|-------------|

The run-time environment was not compatible with notifications; for example, the isolation level was set to snapshot, or one or more SET options are not compatible.

|           |           |
|-----------|-----------|
| Execution | Execution |
|-----------|-----------|

A run-time error occurred during execution.

|        |        |
|--------|--------|
| Object | Object |
|--------|--------|

A database object changed; for example, an underlying object related to the query was dropped or modified.

|       |       |
|-------|-------|
| Owner | Owner |
|-------|-------|

Internal only; not intended to be used in your code.

|           |           |
|-----------|-----------|
| Statement | Statement |
|-----------|-----------|

The Transact-SQL statement is not valid for notifications; for example, a SELECT statement that could not be notified or a non-SELECT statement was executed.

**System** **System**

A system-related event occurred. For example, there was an internal error, the server was restarted, or resource pressure caused the invalidation.

**Timeout**  
**Timeout**

The subscription time-out expired.

**Unknown**  
**Unknown**

Used when the source option sent by the server was not recognized by the client.

## See Also

# SqlNotificationType SqlNotificationType Enum

Describes the different notification types that can be received by an [OnChangeEventHandler](#) event handler through the [SqlNotificationEventArgs](#) parameter.

## Declaration

```
public enum SqlNotificationType  
type SqlNotificationType =
```

## Inheritance Hierarchy



## Remarks

The [SqlNotificationType](#) enumeration is referenced by an instance of the [SqlNotificationEventArgs](#) class. This information is provided when a notification event is fired with the [SqlDependency](#) class.

## Fields

Change  
Change

Data on the server being monitored changed. Use the [SqlNotificationInfo](#) item to determine the details of the change.

Subscribe  
Subscribe

There was a failure to create a notification subscription. Use the [SqlNotificationEventArgs](#) object's [SqlNotificationInfo](#) item to determine the cause of the failure.

Unknown  
Unknown

Used when the type option sent by the server was not recognized by the client.

## See Also

# SqlParameter SqlParameter Class

Represents a parameter to a [SqlCommand](#) and optionally its mapping to [DataSet](#) columns. This class cannot be inherited. For more information on parameters, see [Configuring Parameters and Parameter Data Types](#).

## Declaration

```
[System.ComponentModel.TypeConverter("System.Data.SqlClient.SqlParameter+SqlParameterConverter,
System.Data, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")]
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlParameterConverter))]
[System.ComponentModel.TypeConverter(typeof(System.Data.SqlClient.SqlParameter/SqlParameterConve
rter))]
[System.ComponentModel.TypeConverter("System.Data.SqlClient.SqlParameter+SqlParameterConverter,
System.Data, Version=2.0.5.0, Culture=neutral, PublicKeyToken=b77a5c561934e089")]
public sealed class SqlParameter : System.Data.Common.DbParameter, ICloneable

type SqlParameter = class
    inherit DbParameter
    interface IDbDataParameter
    interface IDataParameter
    interface ICloneable
```

## Inheritance Hierarchy

```
Object Object
MarshalByRefObject MarshalByRefObject
```

## Remarks

Parameter names are not case sensitive.

### Note

Nameless, also called ordinal, parameters are not supported by the .NET Framework Data Provider for SQL Server.

For more information, along with additional sample code demonstrating how to use parameters, see [Commands and Parameters](#).

## Constructors

```
SqlParameter()
```

```
SqlParameter()
```

Initializes a new instance of the [SqlParameter](#) class.

```
SqlParameter(String, SqlDbType)
```

```
SqlParameter(String, SqlDbType)
```

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name and the data type.

```
SqlParameter(String, Object)
```

```
SqlParameter(String, Object)
```

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name and a value of the new [SqlParameter](#).

```
SqlParameter(String, SqlDbType, Int32)
```

```
SqlParameter(String, SqlDbType, Int32)
```

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the [SqlDbType](#), and the size.

```
SqlParameter(String, SqlDbType, Int32, String)
```

```
SqlParameter(String, SqlDbType, Int32, String)
```

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the [SqlDbType](#), the size, and the source column name.

```
SqlParameter(String, SqlDbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object)
```

```
SqlParameter(String, SqlDbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object)
```

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the type of the parameter, the size of the parameter, a [ParameterDirection](#), the precision of the parameter, the scale of the parameter, the source column, a [DataRowVersion](#) to use, and the value of the parameter.

```
SqlParameter(String, SqlDbType, Int32, ParameterDirection, Byte, Byte, String, DataRowVersion, Boolean, Object, String, String, String)
```

```
SqlParameter(String, SqlDbType, Int32, ParameterDirection, Byte, Byte, String, DataRowVersion, Boolean, Object, String, String, String)
```

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the type of the parameter, the length of the parameter the direction, the precision, the scale, the name of the source column, one of the [DataRowVersion](#) values, a Boolean for source column mapping, the value of the [SqlParameter](#), the name of the database where the schema collection for this XML instance is located, the owning relational schema where the schema collection for this XML instance is located, and the name of the schema collection for this parameter.

## Properties

CompareInfo

```
CompareInfo
```

Gets or sets the [CompareInfo](#) object that defines how string comparisons should be performed for this parameter.

DbType

```
DbType
```

Gets or sets the [SqlDbType](#) of the parameter.

Direction

```
Direction
```

Gets or sets a value that indicates whether the parameter is input-only, output-only, bidirectional, or a stored procedure return value parameter.

**ForceColumnEncryption**

**ForceColumnEncryption**

Enforces encryption of a parameter when using [Always Encrypted](#). If SQL Server informs the driver that the parameter does not need to be encrypted, the query using the parameter will fail. This property provides additional protection against security attacks that involve a compromised SQL Server providing incorrect encryption metadata to the client, which may lead to data disclosure.

**IsNullable**

**IsNullable**

Gets or sets a value that indicates whether the parameter accepts null values. [IsNullable](#) is not used to validate the parameter's value and will not prevent sending or receiving a null value when executing a command.

**LocaleId**

**LocaleId**

Gets or sets the locale identifier that determines conventions and language for a particular region.

**Offset**

**Offset**

Gets or sets the offset to the [Value](#) property.

**ParameterName**

**ParameterName**

Gets or sets the name of the [SqlParameter](#).

**Precision**

**Precision**

Gets or sets the maximum number of digits used to represent the [Value](#) property.

**Scale**

**Scale**

Gets or sets the number of decimal places to which [Value](#) is resolved.

**Size**

**Size**

Gets or sets the maximum size, in bytes, of the data within the column.

**SourceColumn**

#### **SourceColumn**

Gets or sets the name of the source column mapped to the [DataSet](#) and used for loading or returning the [Value](#)

#### **SourceColumnNullMapping**

#### **SourceColumnNullMapping**

Sets or gets a value which indicates whether the source column is nullable. This allows [SqlCommandBuilder](#) to correctly generate Update statements for nullable columns.

#### **SourceVersion**

#### **SourceVersion**

Gets or sets the [DataRowVersion](#) to use when you load [Value](#)

#### **SqlDbType**

#### **SqlDbType**

Gets or sets the [SqlDbType](#) of the parameter.

#### **SqlValue**

#### **SqlValue**

Gets or sets the value of the parameter as an SQL type.

#### **TypeName**

#### **TypeName**

Gets or sets the type name for a table-valued parameter.

#### **UdtTypeName**

#### **UdtTypeName**

Gets or sets a `string` that represents a user-defined type as a parameter.

#### **Value**

#### **Value**

Gets or sets the value of the parameter.

#### **XmlSchemaCollectionDatabase**

#### **XmlSchemaCollectionDatabase**

Gets the name of the database where the schema collection for this XML instance is located.

`XmlSchemaCollectionName`

`XmlSchemaCollectionName`

Gets the name of the schema collection for this XML instance.

`XmlSchemaCollectionOwningSchema`

`XmlSchemaCollectionOwningSchema`

The owning relational schema where the schema collection for this XML instance is located.

## Methods

`ResetDbType()`

`ResetDbType()`

Resets the type associated with this [SqlParameter](#).

`ResetSqlDbType()`

`ResetSqlDbType()`

Resets the type associated with this [SqlParameter](#).

`ToString()`

`ToString()`

Gets a string that contains the [ParameterName](#).

`ICloneable.Clone()`

`ICloneable.Clone()`

For a description of this member, see [Clone\(\)](#).

## See Also

# SqlParameter.CompareInfo SqlParameter.CompareInfo

## In this Article

Gets or sets the [CompareInfo](#) object that defines how string comparisons should be performed for this parameter.

```
[System.ComponentModel.Browsable(false)]
public System.Data.SqlTypes.SqlCompareOptions CompareInfo { get; set; }

member this.CompareInfo : System.Data.SqlTypes.SqlCompareOptions with get, set
```

Returns

[SqlCompareOptions](#) [SqlCompareOptions](#)

A [CompareInfo](#) object that defines string comparison for this parameter.

Attributes

[BrowsableAttribute](#)

See

[ADO.NET Overview](#)

Also

# SqlParameter.DbType SqlParameter.DbType

## In this Article

Gets or sets the [SqlDbType](#) of the parameter.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("DataParameter_DbType")]
public override System.Data.DbType DbType { get; set; }

member this.DbType : System.Data.DbType with get, set
```

Returns

[DbType](#) [DbType](#)

One of the [SqlDbType](#) values. The default is [NVarChar](#).

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
private static void AddSqlParameter(SqlCommand command,
    string paramValue)
{
    SqlParameter parameter = new SqlParameter(
        "@Description", SqlDbType.VarChar);
    parameter.Value = paramValue;
    parameter.IsNullable = true;
    command.Parameters.Add(parameter);
}

private static void SetParameterToNull(IDataParameter parameter)
{
    if (parameter.IsNullable)
    {
        parameter.Value = DBNull.Value;
    }
    else
    {
        throw new ArgumentException("Parameter provided is not nullable", "parameter");
    }
}
```

## Remarks

The [SqlDbType](#) and [DbType](#) are linked. Therefore, setting the [DbType](#) changes the [SqlDbType](#) to a supporting [SqlDbType](#).

For a list of the supported data types, see the appropriate [SqlDbType](#) member. For more information, see [DataAdapter Parameters](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters](#)

[DataAdapter Parameters](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlParameter.Direction SqlParameter.Direction

## In this Article

Gets or sets a value that indicates whether the parameter is input-only, output-only, bidirectional, or a stored procedure return value parameter.

```
[System.Data.DataSysDescription("DataParameter_Direction")]
public override System.Data.ParameterDirection Direction { get; set; }

member this.Direction : System.Data.ParameterDirection with get, set
```

Returns

[ParameterDirection](#) [ParameterDirection](#)

One of the [ParameterDirection](#) values. The default is [Input](#).

Attributes

[DataSysDescriptionAttribute](#)

Exceptions

[ArgumentException](#) [ArgumentNullException](#)

The property was not set to one of the valid [ParameterDirection](#) values.

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

[Commands and Parameters](#)

[DataAdapter Parameters](#)

[SQL Server and ADO.NET](#)

## Remarks

If the [ParameterDirection](#) is output, and execution of the associated [SqlCommand](#) does not return a value, the [SqlParameter](#) contains a null value.

[Output](#), [InputOut](#), and [ReturnValue](#) parameters returned by calling [ExecuteReader](#) cannot be accessed until you close the [SqlDataReader](#).

See

[ADO.NET Overview](#)

Also

# SqlParameter.ForceColumnEncryption SqlParameter.ForceColumnEncryption

## In this Article

Enforces encryption of a parameter when using [Always Encrypted](#). If SQL Server informs the driver that the parameter does not need to be encrypted, the query using the parameter will fail. This property provides additional protection against security attacks that involve a compromised SQL Server providing incorrect encryption metadata to the client, which may lead to data disclosure.

```
public bool ForceColumnEncryption { get; set; }  
member this.ForceColumnEncryption : bool with get, set
```

## Returns

**Boolean Boolean**

`true` if the parameter has a force column encryption; otherwise, `false`.

# SqlParameter.ICollectionable.Clone

## In this Article

For a description of this member, see [Clone\(\)](#).

```
object ICollectionable.Clone();
```

Returns

[Object](#)

A new [Object](#) that is a copy of this instance.

## Remarks

This member is an explicit interface member implementation. It can be used only when the [SqlParameter](#) instance is cast to an [ICloneable](#) interface.

See

[ADO.NET Overview](#)

Also

# SqlParameter.IsDBNull

## In this Article

Gets or sets a value that indicates whether the parameter accepts null values. [IsNullable](#) is not used to validate the parameter's value and will not prevent sending or receiving a null value when executing a command.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("DataParameter_IsNullable")]
public override bool IsNullable { get; set; }

member this.IsNullable : bool with get, set
```

Returns

[Boolean](#)

`true` if null values are accepted; otherwise `false`. The default is `false`.

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterNullable()
{
    SqlParameter parameter = new SqlParameter("Description", SqlDbType.VarChar, 88);
    parameter.Nullable = true;
    parameter.Direction = ParameterDirection.Output;
}
```

## Remarks

Null values are handled using the [DBNull](#) class.

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlParameter.LocaleId SqlParameter.LocaleId

## In this Article

Gets or sets the locale identifier that determines conventions and language for a particular region.

```
[System.ComponentModel.Browsable(false)]
public int LocaleId { get; set; }

member this.LocaleId : int with get, set
```

Returns

[Int32](#) [Int32](#)

Returns the locale identifier associated with the parameter.

Attributes

[BrowsableAttribute](#)

## Remarks

The locale identifies conventions and language for a particular geographical region. The codepage used to encode a specific string (the character set) is based on the locale used by that string or the environment that produced it. This property sets (for input parameters) or gets (for output parameters) the locale to be attached to a string when exchanging data with the server. This property is typically used together with the [CompareInfo](#) property.

```
static void CreateSqlParameterLocaleId(){
    SqlParameter parameter = new SqlParameter("pName", SqlDbType.VarChar);
    parameter.LocaleId = 1033; // English - United States
}
```

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.Offset SqlParameter.Offset

## In this Article

Gets or sets the offset to the [Value](#) property.

```
[System.ComponentModel.Browsable(false)]
[System.Data.DataSysDescription("SqlParameter_Offset")]
public int Offset { get; set; }

member this.Offset : int with get, set
```

Returns

[Int32](#) [Int32](#)

The offset to the [Value](#). The default is 0.

Attributes

[BrowsableAttribute](#) [DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterOffset()
{
    SqlParameter parameter = new SqlParameter("pDName", SqlDbType.VarChar);
    parameter.IsNullable = true;
    parameter.Offset = 3;
}
```

## Remarks

The `Offset` property is used for client-side chunking of binary and string data. For example, in order to insert 10MB of text into a column on a server, a user might execute 10 parameterized inserts of 1MB chunks, shifting the value of `Offset` on each iteration by 1MB.

`Offset` specifies the number of bytes for binary types, and the number of characters for strings. The count for strings does not include the terminating character.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.ParameterName SqlParameter.ParameterName

## In this Article

Gets or sets the name of the [SqlParameter](#).

```
[System.Data.DataSysDescription("SqlParameter_ParameterName")]
public override string ParameterName { get; set; }

member this.ParameterName : string with get, set
```

Returns

[String](#)

The name of the [SqlParameter](#). The default is an empty string.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
private static void AddSqlParameter(SqlCommand command)
{
    SqlParameter parameter = new SqlParameter();
    parameter.ParameterName = "@Description";
    parameter.IsNullable = true;
    parameter.DbType = DbType.String;
    parameter.Direction = ParameterDirection.Output;

    command.Parameters.Add(parameter);
}
```

## Remarks

The [ParameterName](#) is specified in the form @paramname. You must set [ParameterName](#) before executing a [SqlCommand](#) that relies on parameters.

See

[Commands and Parameters](#)

Also

[DataAdapter Parameters](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

# SqlParameter.Precision SqlParameter.Precision

## In this Article

Gets or sets the maximum number of digits used to represent the [Value](#) property.

```
[System.Data.DataSysDescription("DbDataParameter_Precision")]
public byte Precision { get; set; }

member this.Precision : byte with get, set
```

## Returns

[Byte](#) [Byte](#)

The maximum number of digits used to represent the [Value](#) property. The default value is 0. This indicates that the data provider sets the precision for [Value](#).

## Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
private static void AddSqlParameter(SqlCommand command)
{
    SqlParameter parameter = new SqlParameter("@Price", SqlDbType.Decimal);
    parameter.Value = 3.1416;
    parameter.Precision = 8;
    parameter.Scale = 4;

    command.Parameters.Add(parameter);
}
```

## Remarks

The [Precision](#) property is used by parameters that have a [SqlDbType](#) of [Decimal](#).

You do not need to specify values for the [Precision](#) and [Scale](#) properties for input parameters, as they can be inferred from the parameter value. [Precision](#) and [Scale](#) are required for output parameters and for scenarios where you need to specify complete metadata for a parameter without indicating a value, such as specifying a null value with a specific precision and scale.

### Note

Use of this property to coerce data passed to the database is not supported. To round, truncate, or otherwise coerce data before passing it to the database, use the [Math](#) class that is part of the [System](#) namespace prior to assigning a value to the parameter's [Value](#) property.

### Note

Microsoft .NET Framework data providers that are included with the .NET Framework version 1.0 do not verify the [Precision](#) or [Scale](#) of [Decimal](#) parameter values. This can cause truncated data being inserted at the data source. If you are using .NET Framework version 1.0, validate the [Precision](#) and [Scale](#) of [Decimal](#) values before setting the parameter value. When you use .NET Framework version 1.1 or a later version, an exception is thrown if a [Decimal](#) parameter value is set with an invalid [Precision](#). [Scale](#) values that exceed the [Decimal](#) parameter scale are still truncated.

## See

[Commands and Parameters](#)

## Also

[DataAdapter Parameters](#)

[SQL Server and ADO.NET](#)



# SqlParameter.ResetDbType SqlParameter.ResetDbType

## In this Article

Resets the type associated with this [SqlParameter](#).

```
public override void ResetDbType ();
override this.ResetDbType : unit -> unit
```

## Remarks

When executing a command that includes parameter values, code can either set the type of a parameter explicitly, or the parameter can infer its type from its value. Calling this method resets the parameter so that it can again infer its type from the value passed in the parameter. Calling this method affects both the [DbType](#) and [SqlDbType](#) properties of the [SqlParameter](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.ResetSqlDbType SqlParameter.ResetSqlDbType

## In this Article

Resets the type associated with this [SqlParameter](#).

```
public void ResetSqlDbType ();  
member this.ResetSqlDbType : unit -> unit
```

## Remarks

When executing a command that includes parameter values, code can either set the type of a parameter explicitly, or the parameter can infer its type from its value. Calling this method resets the parameter so that it can again infer its type from the value passed in the parameter. Calling this method affects both the [DbType](#) and [SqlDbType](#) properties of the [SqlParameter](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.Scale SqlParameter.Scale

## In this Article

Gets or sets the number of decimal places to which [Value](#) is resolved.

```
[System.Data.DataSysDescription("DbDataParameter_Scale")]
public byte Scale { get; set; }

member this.Scale : byte with get, set
```

Returns

[Byte](#) [Byte](#)

The number of decimal places to which [Value](#) is resolved. The default is 0.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterPrecisionScale()
{
    SqlParameter parameter = new SqlParameter("Price", SqlDbType.Decimal);
    parameter.Value = 3.1416;
    parameter.Precision = 8;
    parameter.Scale = 4;
}
```

## Remarks

The [Scale](#) property is used by parameters that have a [SqlDbType](#) of [Decimal](#).

### Warning

Data may be truncated if the [Scale](#) property is not explicitly specified and the data on the server does not fit in scale 0 (the default).

You do not need to specify values for the [Precision](#) and [Scale](#) properties for input parameters, as they can be inferred from the parameter value. [Precision](#) and [Scale](#) are required for output parameters and for scenarios where you need to specify complete metadata for a parameter without indicating a value, such as specifying a null value with a specific precision and scale.

### Note

Use of this property to coerce data passed to the database is not supported. To round, truncate, or otherwise coerce data before passing it to the database, use the [Math](#) class that is part of the [System](#) namespace prior to assigning a value to the parameter's [Value](#) property.

### Note

.NET Framework data providers that are included with the .NET Framework version 1.0 do not verify the [Precision](#) or [Scale](#) of [Decimal](#) parameter values. This can cause truncated data to be inserted at the data source. If you are using .NET Framework version 1.0, validate the [Precision](#) and [SqlParameter](#) of [Decimal](#) values before setting the parameter value. [Scale](#) values that exceed the [Decimal](#) parameter scale are still truncated.

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)



# SqlParameter.Size SqlParameter.Size

## In this Article

Gets or sets the maximum size, in bytes, of the data within the column.

```
[System.Data.DataSysDescription("DbDataParameter_Size")]
public override int Size { get; set; }

member this.Size : int with get, set
```

Returns

[Int32](#) [Int32](#)

The maximum size, in bytes, of the data within the column. The default value is inferred from the parameter value.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterSize()
{
    string description = "12 foot scarf - multiple colors, one previous owner";
    SqlParameter parameter = new SqlParameter("Description", SqlDbType.VarChar);
    parameter.Direction = ParameterDirection.InputOutput;
    parameter.Size = description.Length;
    parameter.Value = description;
}
```

## Remarks

Return values are not affected by this property; return parameters from stored procedures are always fixed-size integers.

For output parameters with a variable length type (nvarchar, for example), the size of the parameter defines the size of the buffer holding the output parameter. The output parameter can be truncated to a size specified with [Size](#). For character types, the size specified with [Size](#) is in characters.

The [Size](#) property is used for binary and string types. For parameters of type [SqlType.String](#), [Size](#) means length in Unicode characters. For parameters of type [SqlType.Xml](#), [Size](#) is ignored.

For nonstring data types and ANSI string data, the [Size](#) property refers to the number of bytes. For Unicode string data, [Size](#) refers to the number of characters. The count for strings does not include the terminating character.

For variable-length data types, [Size](#) describes the maximum amount of data to transmit to the server. For example, for a Unicode string value, [Size](#) could be used to limit the amount of data sent to the server to the first one hundred characters.

If not explicitly set, the size is inferred from the actual size of the specified parameter value.

If the fractional part of the parameter value is greater than the size, then the value will be truncated to match the size.

For fixed length data types, the value of [Size](#) is ignored. It can be retrieved for informational purposes, and returns the maximum amount of bytes the provider uses when transmitting the value of the parameter to the server.

For information about streaming, see [SqlClient Streaming Support](#).

See

Also

[SQL Server Data Types and ADO.NET](#)

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.SourceColumn SqlParameter.SourceColumn

## In this Article

Gets or sets the name of the source column mapped to the [DataSet](#) and used for loading or returning the [Value](#)

```
[System.Data.DataSysDescription("DataParameter_SourceColumn")]
public override string SourceColumn { get; set; }

member this.SourceColumn : string with get, set
```

Returns

[String String](#)

The name of the source column mapped to the [DataSet](#). The default is an empty string.

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterSourceColumn()
{
    SqlParameter parameter = new SqlParameter("Description", SqlDbType.VarChar, 88);
    parameter.SourceColumn = "Description";
}
```

## Remarks

When [SourceColumn](#) is set to anything other than an empty string, the value of the parameter is retrieved from the column with the [SourceColumn](#) name. If [Direction](#) is set to [Input](#), the value is taken from the [DataSet](#). If [Direction](#) is set to [Output](#), the value is taken from the data source. A [Direction](#) of [InputOutput](#) is a combination of both.

For more information about how to use the [SourceColumn](#) property, see [DataAdapter Parameters](#) and [Updating Data Sources with DataAdapters](#).

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[Using Parameters with a DataAdapter](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.SourceColumnNullMapping SqlParameter.SourceColumnNullMapping

## In this Article

Sets or gets a value which indicates whether the source column is nullable. This allows [SqlCommandBuilder](#) to correctly generate Update statements for nullable columns.

```
public override bool SourceColumnNullMapping { get; set; }  
member this.SourceColumnNullMapping : bool with get, set
```

## Returns

[Boolean](#)

`true` if the source column is nullable; `false` if it is not.

## Remarks

`SourceColumnNullMapping` is used by the [SqlCommandBuilder](#) to correctly generate update commands when dealing with nullable columns. Generally, use of `SourceColumnNullMapping` is limited to developers inheriting from [SqlCommandBuilder](#).

`DbCommandBuilder` uses this property to determine whether the source column is nullable, and sets this property to `true` if it is nullable, and `false` if it is not. When [SqlCommandBuilder](#) is generating its Update statement, it examines the `SourceColumnNullMapping` for each parameter. If the property is `true`, [SqlCommandBuilder](#) generates a WHERE clauses like the following (in this query expression, "FieldName" represents the name of the field):

```
((@IsNull_FieldName = 1 AND FieldName IS NULL) OR  
(FieldName = @Original_FieldName))
```

If `SourceColumnNullMapping` for the field is false, [SqlCommandBuilder](#) generates the following WHERE clause:

```
FieldName = @OriginalFieldName
```

In addition, `@IsNull_FieldName` contains 1 if the source field contains null, and 0 if it does not. This mechanism allows for a performance optimization in SQL Server, and provides for common code that works across multiple providers.

### See

[Commands and Parameters \(ADO.NET\)](#)

### Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.SourceVersion SqlParameter.SourceVersion

## In this Article

Gets or sets the [DataRowVersion](#) to use when you load [Value](#)

```
[System.Data.DataSysDescription("SqlParameter_SourceVersion")]
public override System.Data.DataRowVersion SourceVersion { get; set; }

member this.SourceVersion : System.Data.DataRowVersion with get, set
```

Returns

[DataRowVersion](#) [DataRowVersion](#)

One of the [DataRowVersion](#) values. The default is [Current](#).

Attributes

[DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterSourceVersion()
{
    SqlParameter parameter = new SqlParameter("Description", SqlDbType.VarChar, 88);
    parameter.SourceColumn = "Description";
    parameter.SourceVersion = DataRowVersion.Current;
}
```

## Remarks

This property is used by the [SqlDataAdapter.UpdateCommand](#) during an update to determine whether the original or current value is used for a parameter value. This lets primary keys be updated. This property is set to the version of the [DataRow](#) used by the [DataRow.Item](#) property, or one of the [DataRow.GetChildRows](#) methods of the [DataRow](#) object.

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlParameter.SqlDbType SqlParameter.SqlDbType

## In this Article

Gets or sets the [SqlDbType](#) of the parameter.

```
[System.Data.Common.DbProviderSpecificTypeProperty(true)]
[System.Data.DataSysDescription("SqlParameter_SqlDbType")]
public System.Data.SqlDbType SqlDbType { get; set; }

member this.S SqlDbType : System.Data.SqlDbType with get, set
```

Returns

[SqlDbType](#) [SqlDbType](#)

One of the [SqlDbType](#) values. The default is [NVarChar](#).

Attributes

[DbProviderSpecificTypePropertyAttribute](#) [DataSysDescriptionAttribute](#)

## Remarks

The [SqlDbType](#) and [DbType](#) are linked. Therefore, setting the [DbType](#) changes the [SqlDbType](#) to a supporting [SqlDbType](#).

For a list of the supported data types, see the appropriate [SqlDbType](#) member. For more information, see [DataAdapter Parameters](#).

For information about streaming, see [SqlClient Streaming Support](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[Using Parameters with a DataAdapter](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlParameter SqlParameter

In this Article

## Overloads

|   |  |
|---|--|
| <code>SqlParameter()</code>   | Initializes a new instance of the <a href="#">SqlParameter</a> class.  |
| <code>SqlParameter(String, SqlDbType)</code> <code>SqlParameter(String, SqlDbType)</code>   | Initializes a new instance of the <a href="#">SqlParameter</a> class that uses the parameter name and the data type.   |
| <code>SqlParameter(String, Object)</code> <code>SqlParameter(String, Object)</code>   | Initializes a new instance of the <a href="#">SqlParameter</a> class that uses the parameter name and a value of the new <a href="#">SqlParameter</a> .  |
| <code>SqlParameter(String, SqlDbType, Int32)</code> <code>SqlParameter(String, SqlDbType, Int32)</code>   | Initializes a new instance of the <a href="#">SqlParameter</a> class that uses the parameter name, the <a href="#">SqlDbType</a> , and the size.   |
| <code>SqlParameter(String, SqlDbType, Int32, String)</code> <code>SqlParameter(String, SqlDbType, Int32, String)</code>   | Initializes a new instance of the <a href="#">SqlParameter</a> class that uses the parameter name, the <a href="#">SqlDbType</a> , the size, and the source column name.   |
| <code>SqlParameter(String, SqlDbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object)</code> <code>SqlParameter(String, SqlDbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object)</code>   | Initializes a new instance of the <a href="#">SqlParameter</a> class that uses the parameter name, the type of the parameter, the size of the parameter, a <a href="#">ParameterDirection</a> , the precision of the parameter, the scale of the parameter, the source column, a <a href="#">DataRowVersion</a> to use, and the value of the parameter.  |
| <code>SqlParameter(String, SqlDbType, Int32, ParameterDirection, Byte, Byte, String, DataRowVersion, Boolean, Object, String, String, String)</code> <code>SqlParameter(String, SqlDbType, Int32, ParameterDirection, Byte, Byte, String, DataRowVersion, Boolean, Object, String, String, String)</code> | Initializes a new instance of the <a href="#">SqlParameter</a> class that uses the parameter name, the type of the parameter, the length of the parameter the direction, the precision, the scale, the name of the source column, one of the <a href="#">DataRowVersion</a> values, a Boolean for source column mapping, the value of the <a href="#">SqlParameter</a> , the name of the database where the schema collection for this XML instance is located, the owning relational schema where the schema collection for this XML instance is located, and the name of the schema collection for this parameter. |

## SqlParameter()

Initializes a new instance of the [SqlParameter](#) class.

```
public SqlParameter () ;
```

### Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```

private static void AddSqlParameter(SqlCommand command)
{
    SqlParameter parameter = new SqlParameter();
    parameter.ParameterName = "@Description";
    parameter.IsNullable = true;
    parameter.SqlDbType = SqlDbType.VarChar;
    parameter.Direction = ParameterDirection.Output;
    parameter.Size = 88;

    command.Parameters.Add(parameter);
}

```

See

Also

[Commands and Parameters](#)

[DataAdapter Parameters](#)

[SQL Server and ADO.NET](#)

[ADO.NET Overview](#)

## SqlParameter(String, SqlDbType) SqlParameter(String, SqlDbType)

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name and the data type.

```

public SqlParameter (string parameterName, System.Data.SqlDbType dbType);

new System.Data.SqlClient.SqlParameter : string * System.Data.SqlDbType ->
System.Data.SqlClient.SqlParameter

```

Parameters

|               |                        |
|---------------|------------------------|
| parameterName | <a href="#">String</a> |
|---------------|------------------------|

The name of the parameter to map.

|        |                           |
|--------|---------------------------|
| dbType | <a href="#">SqlDbType</a> |
|--------|---------------------------|

One of the [SqlDbType](#) values.

Exceptions

[ArgumentException](#) [ArgumentNullException](#)

The value supplied in the `dbType` parameter is an invalid back-end data type.

Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```

private static void AddSqlParameter(SqlCommand command, string paramValue)
{
    SqlParameter parameter = new SqlParameter("@Description", SqlDbType.VarChar);
    parameter.IsNotNullable = true;
    parameter.Direction = ParameterDirection.Output;
    parameter.Size = 88;
    parameter.Value = paramValue;

    command.Parameters.Add(parameter);
}

```

Remarks

The data type and, if appropriate, **Size** and **Precision** are inferred from the value of the `dbType` parameter.

See

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

## SqlParameter(String, Object) SqlParameter(String, Object)

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name and a value of the new [SqlParameter](#).

```
public SqlParameter (string parameterName, object value);  
new System.Data.SqlClient.SqlParameter : string * obj -> System.Data.SqlClient.SqlParameter
```

### Parameters

parameterName String String

The name of the parameter to map.

value Object Object

An [Object](#) that is the value of the [SqlParameter](#).

### Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
private static void AddSqlParameter(SqlCommand command)  
{  
    SqlParameter parameter = new SqlParameter("@Description",  
        SqlDbType.VarChar, 88, "Description");  
    parameter.IsNullable = true;  
    parameter.Direction = ParameterDirection.Output;  
  
    command.Parameters.Add(parameter);  
}
```

### Remarks

When you specify an [Object](#) in the `value` parameter, the [SqlDbType](#) is inferred from the Microsoft .NET Framework type of the [Object](#).

Use caution when you use this overload of the [SqlParameter](#) constructor to specify integer parameter values. Because this overload takes a `value` of type [Object](#), you must convert the integral value to an [Object](#) type when the value is zero, as the following C# example demonstrates.

```
Parameter = new SqlParameter("@pname", (object)0);
```

If you do not perform this conversion, the compiler assumes that you are trying to call the `SqlParameter (string, SqlDbType)` constructor overload.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## SqlParameter(String, SqlDbType, Int32) SqlParameter(String, SqlDbType, Int32)

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the [SqlDbType](#), and the size.

```
public SqlParameter (string parameterName, System.Data.SqlDbType dbType, int size);  
new System.Data.SqlClient.SqlParameter : string * System.Data.SqlDbType * int ->  
System.Data.SqlClient.SqlParameter
```

## Parameters

parameterName [String](#) [String](#)

The name of the parameter to map.

dbType [SqlDbType](#) [SqlDbType](#)

One of the [SqlDbType](#) values.

size [Int32](#) [Int32](#)

The length of the parameter.

## Exceptions

[ArgumentException](#) [ArgumentException](#)

The value supplied in the `DbType` parameter is an invalid back-end data type.

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
private static void AddSqlParameter(SqlCommand command,  
    string paramValue)  
{  
    SqlParameter parameter = new SqlParameter("@Description",  
        SqlDbType.VarChar, 88);  
    parameter.Nullable = true;  
    parameter.Direction = ParameterDirection.Output;  
    parameter.Value = paramValue;  
  
    command.Parameters.Add(parameter);  
}
```

## Remarks

The **Size** is inferred from the value of the `DbType` parameter if it is not explicitly set in the `size` parameter.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **SqlParameter(String, SqlDbType, Int32, String)** **SqlParameter(String, SqlDbType, Int32, String)**

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the [SqlDbType](#), the size, and the source column name.

```
public SqlParameter (string parameterName, System.Data.SqlDbType dbType, int size, string  
sourceColumn);  
new System.Data.SqlClient.SqlParameter : string * System.Data.SqlDbType * int * string ->  
System.Data.SqlClient.SqlParameter
```

## Parameters

|  |                     |
|--|---------------------|
| parameterName                                | String String       |
| The name of the parameter to map.            |                     |
| dbType                                       | SqIDbType SqIDbType |
| One of the <a href="#">SqIDbType</a> values. |                     |
| size   | Int32 Int32         |
| The length of the parameter.                 |                     |
| sourceColumn                                 | String String       |

The name of the source column ([SourceColumn](#)) if this [SqlParameter](#) is used in a call to [Update](#).

## Exceptions

[ArgumentException](#) [ArgumentNullException](#)

The value supplied in the `DbType` parameter is an invalid back-end data type.

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
private static void AddSqlParameter(SqlCommand command)
{
    SqlParameter parameter = new SqlParameter("@Description",
        SqlDbType.VarChar, 88, "Description");
    parameter.Nullable = true;
    parameter.Direction = ParameterDirection.Output;

    command.Parameters.Add(parameter);
}
```

## Remarks

The **Size** is inferred from the value of the `DbType` parameter if it is not explicitly set in the `size` parameter.

## See

[Commands and Parameters \(ADO.NET\)](#)

## Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **SqlParameter(String, SqIDbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object)** **SqlParameter(String, SqIDbType, Int32, ParameterDirection, Boolean, Byte, Byte, String, DataRowVersion, Object)**

Initializes a new instance of the [SqlParameter](#) class that uses the parameter name, the type of the parameter, the size of the parameter, a [ParameterDirection](#), the precision of the parameter, the scale of the parameter, the source column, a [DataRowVersion](#) to use, and the value of the parameter.

```
public SqlParameter (string parameterName, System.Data.SqlDbType dbType, int size,
System.Data.ParameterDirection direction, bool isNullable, byte precision, byte scale, string
sourceColumn, System.Data.DataRowVersion sourceVersion, object value);

new System.Data.SqlClient.SqlParameter : string * System.Data.SqlDbType * int *
System.Data.ParameterDirection * bool * byte * byte * string * System.Data.DataRowVersion * obj ->
System.Data.SqlClient.SqlParameter
```

## Parameters

parameterName String String

The name of the parameter to map.

dbType SqlDbType SqlDbType

One of the [SqlDbType](#) values.

size Int32 Int32

The length of the parameter.

direction  ParameterDirection ParameterDirection

One of the  [ParameterDirection](#) values.

isNullable  Boolean Boolean

`true` if the value of the field can be null; otherwise `false`.

precision  Byte Byte

The total number of digits to the left and right of the decimal point to which [Value](#) is resolved.

scale  Byte Byte

The total number of decimal places to which [Value](#) is resolved.

sourceColumn  String String

The name of the source column ([SourceColumn](#)) if this [SqlParameter](#) is used in a call to [Update](#).

sourceVersion  DataRowVersion DataRowVersion

One of the  [DataRowVersion](#) values.

value  Object Object

An [Object](#) that is the value of the [SqlParameter](#).

## Exceptions

[ArgumentException](#) [ArgumentNullException](#)

The value supplied in the `dbType` parameter is an invalid back-end data type.

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```

private static void AddSqlParameter(SqlCommand command)
{
    SqlParameter parameter = new SqlParameter("@Description",
        SqlDbType.VarChar, 11, ParameterDirection.Input,
        true, 0, 0, "Description", DataRowVersion.Current,
        "garden hose");
    parameter.IsNullable = true;

    command.Parameters.Add(parameter);
}

```

## Remarks

The **Size** and **Precision** are inferred from the value of the `dbType` parameter if they are not explicitly set in the `size` and `precision` parameters.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## **SqlParameter(String, SqlDbType, Int32, ParameterDirection, Byte, Byte, String, DataRowVersion, Boolean, Object, String, String, String) SqlParameter(String, SqlDbType, Int32, ParameterDirection, Byte, Byte, String, DataRowVersion, Boolean, Object, String, String, String)**

Initializes a new instance of the `SqlParameter` class that uses the parameter name, the type of the parameter, the length of the parameter the direction, the precision, the scale, the name of the source column, one of the `DataRowVersion` values, a Boolean for source column mapping, the value of the `SqlParameter`, the name of the database where the schema collection for this XML instance is located, the owning relational schema where the schema collection for this XML instance is located, and the name of the schema collection for this parameter.

```

public SqlParameter (string parameterName, System.Data.SqlDbType dbType, int size,
System.Data.ParameterDirection direction, byte precision, byte scale, string sourceColumn,
System.Data.DataRowVersion sourceVersion, bool sourceColumnNullMapping, object value, string
xmlSchemaCollectionDatabase, string xmlSchemaCollectionOwningSchema, string
xmlSchemaCollectionName);

new System.Data.SqlClient.SqlParameter : string * System.Data.SqlDbType * int *
System.Data.ParameterDirection * byte * byte * string * System.Data.DataRowVersion * bool * obj *
string * string * string -> System.Data.SqlClient.SqlParameter

```

## Parameters

parameterName

[String](#) [String](#)

The name of the parameter to map.

dbType

[SqlDbType](#) [SqlDbType](#)

One of the `SqlDbType` values.

size

[Int32](#) [Int32](#)

The length of the parameter.

direction

[ParameterDirection](#) [ParameterDirection](#)

|  |  |
|--|--|
| One of the <a href="#">ParameterDirection</a> values.  |  |
| precision  | Byte Byte  |
| The total number of digits to the left and right of the decimal point to which <a href="#">Value</a> is resolved.  |  |
| scale  | Byte Byte  |
| The total number of decimal places to which <a href="#">Value</a> is resolved.   |  |
| sourceColumn   | String String  |
| The name of the source column ( <a href="#">SourceColumn</a> ) if this <a href="#">SqlParameter</a> is used in a call to <a href="#">Update</a> .  |  |
| sourceVersion  | DataRowVersion DataRowVersion  |
| One of the <a href="#">DataRowVersion</a> values.  |  |
| sourceColumnNullMapping  | Boolean Boolean  |
| <code>true</code> if the source column is nullable; <code>false</code> if it is not.   |  |
| value  | Object Object  |
| An <a href="#">Object</a> that is the value of the <a href="#">SqlParameter</a> .  |  |
| xmlSchemaCollectionDatabase  | String String  |
| The name of the database where the schema collection for this XML instance is located.   |  |
| xmlSchemaCollectionOwningSchema  | String String  |
| The owning relational schema where the schema collection for this XML instance is located.   |  |
| xmlSchemaCollectionName  | String String  |
| The name of the schema collection for this parameter.  |  |
| <b>Remarks</b>   |  |
| The <a href="#">Size</a> and <a href="#">Precision</a> are inferred from the value of the <code>dbType</code> parameter if they are not explicitly set in the <code>size</code> and <code>precision</code> parameters. |  |
| See  | <a href="#">SQL Server Data Types and ADO.NET</a>  |
| Also   | <a href="#">Commands and Parameters (ADO.NET)</a><br><a href="#">DataAdapter Parameters (ADO.NET)</a><br><a href="#">Using the .NET Framework Data Provider for SQL Server</a><br><a href="#">ADO.NET Overview</a> |

# SqlParameter.SqlValue SqlParameter.SqlValue

## In this Article

Gets or sets the value of the parameter as an SQL type.

```
[System.ComponentModel.Browsable(false)]
public object SqlValue { get; set; }

member this.SqlValue : obj with get, set
```

Returns

[Object Object](#)

An [Object](#) that is the value of the parameter, using SQL types. The default value is null.

Attributes

[BrowsableAttribute](#)

## Remarks

For input parameters, the value is bound to the [SqlCommand](#) that is sent to the server. For output and return value parameters, the value is set on completion of the [SqlCommand](#) and after the [SqlDataReader](#) is closed.

This property can be set to null or [DBNull.Value](#). Use [DBNull.Value](#) to send a NULL value as the value of the parameter. Use null or do not set [SqlParameter.SqlValue](#) to use the default value for the parameter.

If the application specifies the database type, the bound value is converted to that type when the provider sends the data to the server. The provider tries to convert any type of value if it supports the [IConvertible](#) interface. Conversion errors may result if the specified type is not compatible with the value.

Both the [DbType](#) and [SqlDbType](#) properties can be inferred by setting the [SqlParameter.Value](#).

The [SqlParameter.Value](#) property is overwritten by [SqlDataAdapter.UpdateCommand](#).

Use the [SqlParameter.Value](#) property to return parameter values as common language runtime (CLR) types.

For information about streaming, see [SqlClient Streaming Support](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.ToString SqlParameter.ToString

## In this Article

Gets a string that contains the [ParameterName](#).

```
public override string ToString ();
override this.ToString : unit -> string
```

Returns

[String String](#)

A string that contains the [ParameterName](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.TypeName SqlParameter.TypeName

## In this Article

Gets or sets the type name for a table-valued parameter.

```
[System.ComponentModel.Browsable(false)]  
public string TypeName { get; set; }  
  
member this.TypeName : string with get, set
```

## Returns

[String String](#)

The type name of the specified table-valued parameter.

## Attributes

[BrowsableAttribute](#)

## See

[Table-Valued Parameters \(Katmai\)](#)

## Also

[SQL Server Data Types and ADO.NET](#)

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.UdtTypeName SqlParameter.UdtTypeName

## In this Article

Gets or sets a `string` that represents a user-defined type as a parameter.

```
[System.ComponentModel.Browsable(false)]  
public string UdtTypeName { get; set; }  
  
member this.UdtTypeName : string with get, set
```

## Returns

`String`

A `string` that represents the fully qualified name of a user-defined type in the database.

## Attributes

[BrowsableAttribute](#)

## Remarks

For a sample demonstrating [UdtTypeName](#), see [Retrieving UDT Data](#).

### See

[Commands and Parameters \(ADO.NET\)](#)

### Also

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.Value SqlParameter.Value

## In this Article

Gets or sets the value of the parameter.

```
[System.ComponentModel.TypeConverter(typeof(System.ComponentModel.StringConverter))]
[System.Data.DataSysDescription("DataParameter_Value")]
public override object Value { get; set; }

member this.Value : obj with get, set
```

Returns

[Object](#)

An [Object](#) that is the value of the parameter. The default value is null.

Attributes

[TypeConverterAttribute](#) [DataSysDescriptionAttribute](#)

## Examples

The following example creates a [SqlParameter](#) and sets some of its properties.

```
static void CreateSqlParameterVersion()
{
    SqlParameter parameter = new SqlParameter("Description", SqlDbType.VarChar, 88);
    parameter.Value = "garden hose";
}
```

## Remarks

For input parameters, the value is bound to the [SqlCommand](#) that is sent to the server. For output and return value parameters, the value is set on completion of the [SqlCommand](#) and after the [SqlDataReader](#) is closed.

This property can be set to null or [Value](#). Use [Value](#) to send a NULL value as the value of the parameter. Use null or do not set [Value](#) to use the default value for the parameter.

An exception is thrown if non-Unicode XML data is passed as a string.

If the application specifies the database type, the bound value is converted to that type when the provider sends the data to the server. The provider tries to convert any type of value if it supports the [IConvertible](#) interface. Conversion errors may result if the specified type is not compatible with the value.

Both the [DbType](#) and [SqlDbType](#) properties can be inferred by setting the [Value](#).

The [Value](#) property is overwritten by [SqlDataAdapter.UpdateCommand](#).

For information about streaming, see [SqlClient Streaming Support](#).

See

[SQL Server Data Types and ADO.NET](#)

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.XmlSchemaCollectionDatabase SqlParameter.XmlSchemaCollectionDatabase

## In this Article

Gets the name of the database where the schema collection for this XML instance is located.

```
public string XmlSchemaCollectionDatabase { get; set; }  
member this.XmlSchemaCollectionDatabase : string with get, set
```

Returns

[String](#)

The name of the database where the schema collection for this XML instance is located.

## Remarks

This value is null (`Nothing` in Microsoft Visual Basic) if the collection is defined within the current database. It is also null if there is no schema collection, in which case [XmlSchemaCollectionName](#) and [XmlSchemaCollectionOwningSchema](#) are also null.

See

[ADO.NET Overview](#)

Also

# SqlParameter.XmlSchemaCollectionName SqlParameter.XmlSchemaCollectionName

## In this Article

Gets the name of the schema collection for this XML instance.

```
public string XmlSchemaCollectionName { get; set; }  
member this.XmlSchemaCollectionName : string with get, set
```

Returns

[String](#)

The name of the schema collection for this XML instance.

## Remarks

This value is null (`Nothing` in Microsoft Visual Basic) if there is no associated schema collection. If the value is null, then [XmlSchemaCollectionDatabase](#) and [XmlSchemaCollectionOwningSchema](#) are also null.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameter.XmlSchemaCollectionOwningSchema

## SqlParameter.XmlSchemaCollectionOwningSchema

### In this Article

The owning relational schema where the schema collection for this XML instance is located.

```
public string XmlSchemaCollectionOwningSchema { get; set; }  
member this.XmlSchemaCollectionOwningSchema : string with get, set
```

Returns

[String](#)

The owning relational schema for this XML instance.

## Remarks

This value is null (`Nothing` in Microsoft Visual Basic) if the collection is defined within the current database. It is also null if there is no schema collection, in which case [XmlSchemaCollectionDatabase](#) and [XmlSchemaCollectionName](#) are also null.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection SqlParameterCollection Class

Represents a collection of parameters associated with a [SqlCommand](#) and their respective mappings to columns in a [DataSet](#). This class cannot be inherited.

## Declaration

```
[System.ComponentModel.ListBindable(false)]
public sealed class SqlParameterCollection : System.Data.Common.DbParameterCollection,
System.Collections.IList

type SqlParameterCollection = class
    inherit DbParameterCollection
    interface IDataParameterCollection
    interface IList
    interface ICollection
    interface IEnumerable
```

## Inheritance Hierarchy

```
Object Object
MarshalByRefObject MarshalByRefObject
```

## Remarks

If the command contains an ad hoc SQL statement, as opposed to a stored-procedure name, the number of the parameters in the collection must be equal to the number of parameter placeholders within the command text, or SQL Server raises an error. With a stored procedure, all the parameters declared in the stored procedure without a default value must be provided. Parameters declared with a default value are optional. This lets you specify a value other than the default.

For more information with additional sample code demonstrating how to use parameters, see [Commands and Parameters](#).

## Properties

Count

Count

Returns an Integer that contains the number of elements in the [SqlParameterCollection](#). Read-only.

IsFixedSize

IsFixedSize

Gets a value that indicates whether the [SqlParameterCollection](#) has a fixed size.

IsReadOnly

IsReadOnly

Gets a value that indicates whether the [SqlParameterCollection](#) is read-only.

IsSynchronized

**IsSynchronized**

Gets a value that indicates whether the [SqlParameterCollection](#) is synchronized.

**Item[String]**

**Item[String]**

Gets the [SqlParameter](#) with the specified name.

**Item[Int32]**

**Item[Int32]**

Gets the [SqlParameter](#) at the specified index.

**SyncRoot**

**SyncRoot**

Gets an object that can be used to synchronize access to the [SqlParameterCollection](#).

## Methods

**Add(SqlParameter)**

**Add(SqlParameter)**

Adds the specified [SqlParameter](#) object to the [SqlParameterCollection](#).

**Add(Object)**

**Add(Object)**

Adds the specified [SqlParameter](#) object to the [SqlParameterCollection](#).

**Add(String, SqlDbType)**

**Add(String, SqlDbType)**

Adds a [SqlParameter](#) to the [SqlParameterCollection](#) given the parameter name and the data type.

**Add(String, Object)**

**Add(String, Object)**

Adds the specified [SqlParameter](#) object to the [SqlParameterCollection](#).

**Add(String, SqlDbType, Int32)**

**Add(String, SqlDbType, Int32)**

Adds a [SqlParameter](#) to the [SqlParameterCollection](#), given the specified parameter name, [SqlDbType](#) and size.

```
Add(String, SqlDbType, Int32, String)  
Add(String, SqlDbType, Int32, String)
```

Adds a [SqlParameter](#) to the [SqlParameterCollection](#) with the parameter name, the data type, and the column length.

```
AddRange(Array)  
AddRange(Array)
```

Adds an array of values to the end of the [SqlParameterCollection](#).

```
AddRange(SqlParameter[])  
AddRange(SqlParameter[])
```

Adds an array of [SqlParameter](#) values to the end of the [SqlParameterCollection](#).

```
AddWithValue(String, Object)  
AddWithValue(String, Object)
```

Adds a value to the end of the [SqlParameterCollection](#).

```
Clear()  
Clear()
```

Removes all the [SqlParameter](#) objects from the [SqlParameterCollection](#).

```
Contains(SqlParameter)  
Contains(SqlParameter)
```

Determines whether the specified [SqlParameter](#) is in this [SqlParameterCollection](#).

```
Contains(Object)  
Contains(Object)
```

Determines whether the specified [Object](#) is in this [SqlParameterCollection](#).

```
Contains(String)  
Contains(String)
```

Determines whether the specified parameter name is in this [SqlParameterCollection](#).

```
CopyTo(Array, Int32)  
CopyTo(Array, Int32)
```

Copies all the elements of the current [SqlParameterCollection](#) to the specified one-dimensional [Array](#) starting at the specified destination [Array](#) index.

```
CopyTo(SqlParameter[], Int32)
```

```
CopyTo(SqlParameter[], Int32)
```

Copies all the elements of the current [SqlParameterCollection](#) to the specified [SqlParameterCollection](#) starting at the specified destination index.

```
GetEnumerator()
```

```
GetEnumerator()
```

Returns an enumerator that iterates through the [SqlParameterCollection](#).

```
IndexOf(SqlParameter)
```

```
IndexOf(SqlParameter)
```

Gets the location of the specified [SqlParameter](#) within the collection.

```
IndexOf(Object)
```

```
IndexOf(Object)
```

Gets the location of the specified [Object](#) within the collection.

```
IndexOf(String)
```

```
IndexOf(String)
```

Gets the location of the specified [SqlParameter](#) with the specified name.

```
Insert(Int32, SqlParameter)
```

```
Insert(Int32, SqlParameter)
```

Inserts a [SqlParameter](#) object into the [SqlParameterCollection](#) at the specified index.

```
Insert(Int32, Object)
```

```
Insert(Int32, Object)
```

Inserts an [Object](#) into the [SqlParameterCollection](#) at the specified index.

```
Remove(SqlParameter)
```

```
Remove(SqlParameter)
```

Removes the specified [SqlParameter](#) from the collection.

```
Remove(Object)
```

`Remove(Object)`

Removes the specified [SqlParameter](#) from the collection.

`RemoveAt(Int32)`

`RemoveAt(Int32)`

Removes the [SqlParameter](#) from the [SqlParameterCollection](#) at the specified index.

`RemoveAt(String)`

`RemoveAt(String)`

Removes the [SqlParameter](#) from the [SqlParameterCollection](#) at the specified parameter name.

## See Also

# SqlParameterCollection.Add SqlParameterCollection.Add

In this Article

## Overloads

|                                       |                                       |
|---------------------------------------|---------------------------------------|
| Add(SqlParameter)                     | Add(SqlParameter)                     |
| Add(Object)                           | Add(Object)                           |
| Add(String, SqlDbType)                | Add(String, SqlDbType)                |
| Add(String, Object)                   | Add(String, Object)                   |
| Add(String, SqlDbType, Int32)         | Add(String, SqlDbType, Int32)         |
| Add(String, SqlDbType, Int32, String) | Add(String, SqlDbType, Int32, String) |

## Add(SqlParameter) Add(SqlParameter)

Adds the specified [SqlParameter](#) object to the [SqlParameterCollection](#).

```
public System.Data.SqlClient.SqlParameter Add (System.Data.SqlClient.SqlParameter value);  
override this.Add : System.Data.SqlClient.SqlParameter -> System.Data.SqlClient.SqlParameter
```

Parameters

**value** [SqlParameter](#) [SqlParameter](#)

The [SqlParameter](#) to add to the collection.

Returns

[SqlParameter](#) [SqlParameter](#)

A new [SqlParameter](#) object.

Exceptions

[ArgumentException](#) [ArgumentException](#)

The [SqlParameter](#) specified in the **value** parameter is already added to this or another [SqlParameterCollection](#).

[InvalidOperationException](#) [InvalidOperationException](#)

The parameter passed was not a [SqlParameter](#).

## ArgumentNullException ArgumentNullException

The `value` parameter is null.

### Examples

```
public void AddSqlParameter(SqlCommand command)
{
    command.Parameters.Add(new SqlParameter("Description", "Beverages"));
}
```

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## Add(Object) Add(Object)

Adds the specified [SqlParameter](#) object to the [SqlParameterCollection](#).

```
public override int Add (object value);
override this.Add : obj -> int
```

Parameters

value

[Object](#) [Object](#)

An [Object](#).

Returns

[Int32](#) [Int32](#)

The index of the new [SqlParameter](#) object.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## Add(String, SqlDbType) Add(String, SqlDbType)

Adds a [SqlParameter](#) to the [SqlParameterCollection](#) given the parameter name and the data type.

```
public System.Data.SqlClient.SqlParameter Add (string parameterName, System.Data.SqlDbType
sqlDbType);
override this.Add : string * System.Data.SqlDbType -> System.Data.SqlClient.SqlParameter
```

Parameters

parameterName

[String](#) [String](#)

The name of the parameter.

sqlDbType

[SqlDbType](#) [SqlDbType](#)

One of the [SqlDbType](#) values.

Returns

## SqlParameter SqlParameter

A new [SqlParameter](#) object.

### Examples

```
public void AddSqlParameter(SqlCommand command)
{
    SqlParameter param = command.Parameters.Add(
        "@Description", SqlDbType.NVarChar);
    param.Size = 16;
    param.Value = "Beverages";
}
```

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## Add(String, Object) Add(String, Object)

Adds the specified [SqlParameter](#) object to the [SqlParameterCollection](#).

```
[System.Obsolete("Do not call this method.")]
[System.Obsolete("Add(String parameterName, Object value) has been deprecated. Use
AddWithValue(String parameterName, Object value). http://go.microsoft.com/fwlink/?LinkId=14202",
false)]
public System.Data.SqlClient.SqlParameter Add (string parameterName, object value);

override this.Add : string * obj -> System.Data.SqlClient.SqlParameter
```

Parameters

parameterName

[String](#)

The name of the [SqlParameter](#) to add to the collection.

value

[Object](#)

A [Object](#).

Returns

## SqlParameter SqlParameter

A new [SqlParameter](#) object.

Use caution when you are using this overload of the `SqlParameterCollection.Add` method to specify integer parameter values. Because this overload takes a `value` of type [Object](#), you must convert the integral value to an [Object](#) type when the value is zero, as the following C# example demonstrates.

```
parameters.Add("@pname", Convert.ToInt32(0));
```

If you do not perform this conversion, the compiler assumes that you are trying to call the

`SqlParameterCollection.Add (string, SqlDbType)` overload.

Attributes

[ObsoleteAttribute](#)

Exceptions

## ArgumentException ArgumentException

The [SqlParameter](#) specified in the `value` parameter is already added to this or another [SqlParameterCollection](#).

## ArgumentNullException ArgumentNullException

The `value` parameter is null.

## Examples

```
public void AddSqlParameter(SqlCommand command)
{
    SqlParameter param = new SqlParameter(
        "@Description", SqlDbType.NVarChar, 16);
    param.Value = "Beverages";
    command.Parameters.Add(param);
}
```

See

[ADO.NET Overview](#)

Also

## Add(String, SqlDbType, Int32) Add(String, SqlDbType, Int32)

Adds a [SqlParameter](#) to the [SqlParameterCollection](#), given the specified parameter name, [SqlDbType](#) and size.

```
public System.Data.SqlClient.SqlParameter Add (string parameterName, System.Data.SqlDbType
sqlDbType, int size);

override this.Add : string * System.Data.SqlDbType * int -> System.Data.SqlClient.SqlParameter
```

Parameters

parameterName [String](#) [String](#)

The name of the parameter.

sqlDbType [SqlDbType](#) [SqlDbType](#)

The [SqlDbType](#) of the [SqlParameter](#) to add to the collection.

size [Int32](#) [Int32](#)

The size as an [Int32](#).

Returns

[SqlParameter](#) [SqlParameter](#)

A new [SqlParameter](#) object.

## Examples

```
public void AddSqlParameter(SqlCommand command)
{
    SqlParameter param = new SqlParameter(
        "@Description", SqlDbType.NVarChar, 16);
    param.Value = "Beverages";
    command.Parameters.Add(param);
}
```

## Remarks

This overload is useful when you are adding a parameter of a variable-length data type such as `varchar` or `binary`.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## Add(String, SqlDbType, Int32, String) Add(String, SqlDbType, Int32, String)

Adds a [SqlParameter](#) to the [SqlParameterCollection](#) with the parameter name, the data type, and the column length.

```
public System.Data.SqlClient.SqlParameter Add (string parameterName, System.Data.SqlDbType  
sqlDbType, int size, string sourceColumn);  
  
override this.Add : string * System.Data.SqlDbType * int * string ->  
System.Data.SqlClient.SqlParameter
```

Parameters

parameterName [String](#) [String](#)

The name of the parameter.

sqlDbType [SqlDbType](#) [SqlDbType](#)

One of the [SqlDbType](#) values.

size [Int32](#) [Int32](#)

The column length.

sourceColumn [String](#) [String](#)

The name of the source column ([SourceColumn](#)) if this [SqlParameter](#) is used in a call to [Update](#).

Returns

[SqlParameter](#) [SqlParameter](#)

A new [SqlParameter](#) object.

Examples

```
public void AddSqlParameter(SqlCommand cmd)  
{  
    SqlParameter p1 = cmd.Parameters.Add("@Description", SqlDbType.NVarChar, 16, "Description");  
}
```

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.AddRange SqlParameter Collection.AddRange

In this Article

## Overloads

|   |   |
|---|---|
| AddRange(Array) AddRange(Array)                   | Adds an array of values to the end of the <a href="#">SqlParameterCollection</a> .                              |
| AddRange(SqlParameter[]) AddRange(SqlParameter[]) | Adds an array of <a href="#">SqlParameter</a> values to the end of the <a href="#">SqlParameterCollection</a> . |

## AddRange(Array) AddRange(Array)

Adds an array of values to the end of the [SqlParameterCollection](#).

```
public override void AddRange (Array values);  
override this.AddRange : Array -> unit
```

Parameters

values Array Array

The [Array](#) values to add.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## AddRange(SqlParameter[]) AddRange(SqlParameter[])

Adds an array of [SqlParameter](#) values to the end of the [SqlParameterCollection](#).

```
public void AddRange (System.Data.SqlClient.SqlParameter[] values);  
override this.AddRange : System.Data.SqlClient.SqlParameter[] -> unit
```

Parameters

values [SqlParameter](#)[]

The [SqlParameter](#) values to add.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.AddWithValue SqlParameterCollection.AddWithValue

## In this Article

Adds a value to the end of the [SqlParameterCollection](#).

```
public System.Data.SqlClient.SqlParameter AddWithValue (string parameterName, object value);  
member this.AddWithValue : string * obj -> System.Data.SqlClient.SqlParameter
```

## Parameters

parameterName [String](#) [String](#)

The name of the parameter.

value [Object](#) [Object](#)

The value to be added. Use [Value](#) instead of null, to indicate a null value.

## Returns

[SqlParameter](#) [SqlParameter](#)

A [SqlParameter](#) object.

## Examples

The following example demonstrates how to use the `AddWithValue` method.

```
private static void UpdateDemographics(Int32 customerID,  
    string demoXml, string connectionString)  
{  
    // Update the demographics for a store, which is stored  
    // in an xml column.  
    string commandText = "UPDATE Sales.Store SET Demographics = @demographics "  
        + "WHERE CustomerID = @ID;";  
  
    using (SqlConnection connection = new SqlConnection(connectionString))  
    {  
        SqlCommand command = new SqlCommand(commandText, connection);  
        command.Parameters.Add("@ID", SqlDbType.Int);  
        command.Parameters["@ID"].Value = customerID;  
  
        // Use AddWithValue to assign Demographics.  
        // SQL Server will implicitly convert strings into XML.  
        command.Parameters.AddWithValue("@demographics", demoXml);  
  
        try  
        {  
            connection.Open();  
            Int32 rowsAffected = command.ExecuteNonQuery();  
            Console.WriteLine("RowsAffected: {0}", rowsAffected);  
        }  
        catch (Exception ex)  
        {  
            Console.WriteLine(ex.Message);  
        }  
    }  
}
```

## Remarks

[AddWithValue](#) replaces the `SqlParameterCollection.Add` method that takes a `String` and an `Object`. The overload of `Add` that takes a string and an object was deprecated because of possible ambiguity with the `SqlParameterCollection.Add` overload that takes a `String` and a `SqlDbType` enumeration value where passing an integer with the string could be interpreted as being either the parameter value or the corresponding `SqlDbType` value. Use [AddWithValue](#) whenever you want to add a parameter by specifying its name and value.

For `SqlDbType.Xml` enumeration values, you can use a string, an XML value, an `XmlReader` derived type instance, or a `SqlXml` object.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.Clear SqlParameterCollection.Clear

## In this Article

Removes all the [SqlParameter](#) objects from the [SqlParameterCollection](#).

```
public override void Clear ();  
override this.Clear : unit -> unit
```

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

# SqlParameterCollection.Contains SqlParameter Collection.Contains

In this Article

## Overloads

|   |   |
|---|---|
| Contains(SqlParameter) Contains(SqlParameter) | Determines whether the specified <a href="#">SqlParameter</a> is in this <a href="#">SqlParameterCollection</a> . |
| Contains(Object) Contains(Object)             | Determines whether the specified <a href="#">Object</a> is in this <a href="#">SqlParameterCollection</a> .       |
| Contains(String) Contains(String)             | Determines whether the specified parameter name is in this <a href="#">SqlParameterCollection</a> .               |

## Contains(SqlParameter) Contains(SqlParameter)

Determines whether the specified [SqlParameter](#) is in this [SqlParameterCollection](#).

```
public bool Contains (System.Data.SqlClient.SqlParameter value);  
override this.Contains : System.Data.SqlClient.SqlParameter -> bool
```

Parameters

value [SqlParameter](#) [SqlParameter](#)

The [SqlParameter](#) value.

Returns

[Boolean](#) [Boolean](#)

[true](#) if the [SqlParameterCollection](#) contains the value; otherwise [false](#).

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## Contains(Object) Contains(Object)

Determines whether the specified [Object](#) is in this [SqlParameterCollection](#).

```
public override bool Contains (object value);  
override this.Contains : obj -> bool
```

Parameters

value [Object](#) [Object](#)

The [Object](#) value.

Returns

[Boolean Boolean](#)

`true` if the [SqlParameterCollection](#) contains the value; otherwise `false`.

See

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## Contains(String) Contains(String)

Determines whether the specified parameter name is in this [SqlParameterCollection](#).

```
public override bool Contains (string value);  
override this.Contains : string -> bool
```

Parameters

value

[String String](#)

The [String](#) value.

Returns

[Boolean Boolean](#)

`true` if the [SqlParameterCollection](#) contains the value; otherwise `false`.

See

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.CopyTo SqlParameterCollection.CopyTo

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">CopyTo(Array, Int32)</a> <a href="#">CopyTo(Array, Int32)</a>                   | Copies all the elements of the current <a href="#">SqlParameterCollection</a> to the specified one-dimensional <a href="#">Array</a> starting at the specified destination <a href="#">Array</a> index. |
| <a href="#">CopyTo(SqlParameter[], Int32)</a> <a href="#">CopyTo(SqlParameter[], Int32)</a> | Copies all the elements of the current <a href="#">SqlParameterCollection</a> to the specified <a href="#">SqlParameterCollection</a> starting at the specified destination index.                      |

### CopyTo(Array, Int32) CopyTo(Array, Int32)

Copies all the elements of the current [SqlParameterCollection](#) to the specified one-dimensional [Array](#) starting at the specified destination [Array](#) index.

```
public override void CopyTo (Array array, int index);  
override this.CopyTo : Array * int -> unit
```

#### Parameters

array [Array](#)

The one-dimensional [Array](#) that is the destination of the elements copied from the current [SqlParameterCollection](#).

index [Int32](#)

A 32-bit integer that represents the index in the [Array](#) at which copying starts.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

### CopyTo(SqlParameter[], Int32) CopyTo(SqlParameter[], Int32)

Copies all the elements of the current [SqlParameterCollection](#) to the specified [SqlParameterCollection](#) starting at the specified destination index.

```
public void CopyTo (System.Data.SqlClient.SqlParameter[] array, int index);  
override this.CopyTo : System.Data.SqlClient.SqlParameter[] * int -> unit
```

#### Parameters

array [SqlParameter](#)[]

The [SqlParameterCollection](#) that is the destination of the elements copied from the current [SqlParameterCollection](#).

|       |             |
|-------|-------------|
| index | Int32 Int32 |
|-------|-------------|

A 32-bit integer that represents the index in the [SqlParameterCollection](#) at which copying starts.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.Count SqlParameterCollection.Count

## In this Article

Returns an Integer that contains the number of elements in the [SqlParameterCollection](#). Read-only.

```
[System.ComponentModel.Browsable(false)]  
public override int Count { get; }  
  
member this.Count : int
```

Returns

[Int32](#) [Int32](#)

The number of elements in the [SqlParameterCollection](#) as an Integer.

Attributes

[BrowsableAttribute](#)

See

[ADO.NET Overview](#)

Also

# SqlParameterCollection.GetEnumerator SqlParameterCollection.GetEnumerator

## In this Article

Returns an enumerator that iterates through the [SqlParameterCollection](#).

```
public override System.Collections.IEnumerator GetEnumerator ();
override this.GetEnumerator : unit -> System.Collections.IEnumerator
```

Returns

[IEnumerator](#) [IEnumerator](#)

An [IEnumerator](#) for the [SqlParameterCollection](#).

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.IndexOf SqlParameterCollection.IndexOf

In this Article

## Overloads

|   |  |
|---|--|
| <a href="#">IndexOf(SqlParameter) IndexOf(SqlParameter)</a> | Gets the location of the specified <a href="#">SqlParameter</a> within the collection.   |
| <a href="#">IndexOf(Object) IndexOf(Object)</a>             | Gets the location of the specified <a href="#">Object</a> within the collection.         |
| <a href="#">IndexOf(String) IndexOf(String)</a>             | Gets the location of the specified <a href="#">SqlParameter</a> with the specified name. |

### IndexOf(SqlParameter) IndexOf(SqlParameter)

Gets the location of the specified [SqlParameter](#) within the collection.

```
public int IndexOf (System.Data.SqlClient.SqlParameter value);  
override this.IndexOf : System.Data.SqlClient.SqlParameter -> int
```

Parameters

value [SqlParameter SqlParameter](#)

The [SqlParameter](#) to find.

Returns

[Int32 Int32](#)

The zero-based location of the specified [SqlParameter](#) that is a [SqlParameter](#) within the collection. Returns -1 when the object does not exist in the [SqlParameterCollection](#).

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

### IndexOf(Object) IndexOf(Object)

Gets the location of the specified [Object](#) within the collection.

```
public override int IndexOf (object value);  
override this.IndexOf : obj -> int
```

Parameters

value [Object Object](#)

The [Object](#) to find.

Returns

[Int32](#) [Int32](#)

The zero-based location of the specified [Object](#) that is a [SqlParameter](#) within the collection. Returns -1 when the object does not exist in the [SqlParameterCollection](#).

See

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

## IndexOf(String) IndexOf(String)

Gets the location of the specified [SqlParameter](#) with the specified name.

```
public override int IndexOf (string parameterName);  
override this.IndexOf : string -> int
```

Parameters

parameterName [String](#) [String](#)

The case-sensitive name of the [SqlParameter](#) to find.

Returns

[Int32](#) [Int32](#)

The zero-based location of the specified [SqlParameter](#) with the specified case-sensitive name. Returns -1 when the object does not exist in the [SqlParameterCollection](#).

See

Also

[Commands and Parameters \(ADO.NET\)](#)

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.Insert SqlParameterCollection.Insert

In this Article

## Overloads

|  |   |
|--|---|
| <code>Insert(Int32, SqlParameter) Insert(Int32, SqlParameter)</code> | Inserts a <a href="#">SqlParameter</a> object into the <a href="#">SqlParameterCollection</a> at the specified index. |
| <code>Insert(Int32, Object) Insert(Int32, Object)</code>             | Inserts an <a href="#">Object</a> into the <a href="#">SqlParameterCollection</a> at the specified index.             |

### Insert(Int32, SqlParameter) Insert(Int32, SqlParameter)

Inserts a [SqlParameter](#) object into the [SqlParameterCollection](#) at the specified index.

```
public void Insert (int index, System.Data.SqlClient.SqlParameter value);  
override this.Insert : int * System.Data.SqlClient.SqlParameter -> unit
```

Parameters

|  |                              |
|--|------------------------------|
| index  | <a href="#">Int32</a>        |
| The zero-based index at which value should be inserted.  |                              |
| value  | <a href="#">SqlParameter</a> |
| A <a href="#">SqlParameter</a> object to be inserted in the <a href="#">SqlParameterCollection</a> . |                              |

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

### Insert(Int32, Object) Insert(Int32, Object)

Inserts an [Object](#) into the [SqlParameterCollection](#) at the specified index.

```
public override void Insert (int index, object value);  
override this.Insert : int * obj -> unit
```

Parameters

|  |                        |
|--|------------------------|
| index  | <a href="#">Int32</a>  |
| The zero-based index at which value should be inserted.                                  |                        |
| value  | <a href="#">Object</a> |
| An <a href="#">Object</a> to be inserted in the <a href="#">SqlParameterCollection</a> . |                        |

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.IsFixedSize SqlParameterCollection.IsFixedSize

## In this Article

Gets a value that indicates whether the [SqlParameterCollection](#) has a fixed size.

```
public override bool IsFixedSize { get; }  
member this.IsFixedSize : bool
```

Returns

[Boolean Boolean](#)

Returns `true` if the [SqlParameterCollection](#) has a fixed size; otherwise `false`.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.IsReadOnly SqlParameterCollection.IsReadOnly

## In this Article

Gets a value that indicates whether the [SqlParameterCollection](#) is read-only.

```
public override bool IsReadOnly { get; }  
member this.IsReadOnly : bool
```

Returns

[Boolean Boolean](#)

Returns `true` if the [SqlParameterCollection](#) is read only; otherwise `false`.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.IsSynchronized SqlParameterCollection.IsSynchronized

## In this Article

Gets a value that indicates whether the [SqlParameterCollection](#) is synchronized.

```
public override bool IsSynchronized { get; }  
member this.IsSynchronized : bool
```

Returns

[Boolean Boolean](#)

Returns `true` if the [SqlParameterCollection](#) is synchronized; otherwise `false`.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.Item[Int32] SqlParameter Collection.Item[Int32]

In this Article

## Overloads

|   |  |
|---|--|
| <a href="#">Item[String] Item[String]</a> | Gets the <a href="#">SqlParameter</a> with the specified name. |
| <a href="#">Item[Int32] Item[Int32]</a>   | Gets the <a href="#">SqlParameter</a> at the specified index.  |

## Item[String] Item[String]

Gets the [SqlParameter](#) with the specified name.

```
[System.ComponentModel.Browsable(false)]
public System.Data.SqlClient.SqlParameter this[string parameterName] { get; set; }

member this.Item(string) : System.Data.SqlClient.SqlParameter with get, set
```

Parameters

parameterName [String](#) [String](#)

The name of the parameter to retrieve.

Returns

[SqlParameter](#) [SqlParameter](#)

The [SqlParameter](#) with the specified name.

Attributes

[BrowsableAttribute](#)

Exceptions

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

The specified `parameterName` is not valid.

Remarks

The `parameterName` is used to look up the index value in the underlying [SqlParameterCollection](#). If the `parameterName` is not valid, an [IndexOutOfRangeException](#) will be thrown.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## Item[Int32] Item[Int32]

Gets the [SqlParameter](#) at the specified index.

```
[System.ComponentModel.Browsable(false)]
public System.Data.SqlClient.SqlParameter this[int index] { get; set; }

member this.Item(int) : System.Data.SqlClient.SqlParameter with get, set
```

## Parameters

index Int32 Int32

The zero-based index of the parameter to retrieve.

## Returns

[SqlParameter](#) [SqlParameter](#)

The [SqlParameter](#) at the specified index.

Attributes BrowsableAttribute

## Exceptions

[IndexOutOfRangeException](#) [IndexOutOfRangeException](#)

The specified index does not exist.

## Examples

The following example demonstrates creating [SqlParameter](#) objects to supply an input parameter to a stored procedure that returns results in an output parameter. The code iterates through the items in the [SqlParameterCollection](#) and displays some parameter properties in the console window. This example assumes a valid connection string to the **AdventureWorks** sample database on an instance of SQL Server.

```

static private string CreateSqlParameters(int documentID)
{
    // Assumes GetConnectionString returns a valid connection string to the
    // AdventureWorks sample database on an instance of SQL Server 2005.
    using (SqlConnection connection =
        new SqlConnection(GetConnectionString()))
    {
        connection.Open();
        SqlCommand command = connection.CreateCommand();
        try
        {
            // Setup the command to execute the stored procedure.
            command.CommandText = "GetDocumentSummary";
            command.CommandType = CommandType.StoredProcedure;

            // Create the input parameter for the DocumentID.
            SqlParameter paramID =
                new SqlParameter("@DocumentID", SqlDbType.Int);
            paramID.Value = documentID;
            command.Parameters.Add(paramID);

            // Create the output parameter to retrieve the summary.
            SqlParameter paramSummary =
                new SqlParameter("@DocumentSummary", SqlDbType.NVarChar, -1);
            paramSummary.Direction = ParameterDirection.Output;
            command.Parameters.Add(paramSummary);

            // List the parameters and some of properties.
            SqlParameterCollection paramCollection = command.Parameters;
            string parameterList = "";
            for (int i = 0; i < paramCollection.Count; i++)
            {
                parameterList += String.Format(" {0}, {1}, {2}"
",
                    paramCollection[i], paramCollection[i].DbType,
                    paramCollection[i].Direction);
            }
            Console.WriteLine("Parameter Collection:
" + parameterList);

            // Execute the stored procedure; retrieve
            // and display the output parameter value.
            command.ExecuteNonQuery();
            Console.WriteLine((String)(paramSummary.Value));
            return (String)(paramSummary.Value);
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            return null;
        }
    }
}

```

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.Remove SqlParameterCollection.Remove

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">Remove(SqlParameter)</a> <a href="#">Remove(SqlParameter)</a> | Removes the specified <a href="#">SqlParameter</a> from the collection. |
| <a href="#">Remove(Object)</a> <a href="#">Remove(Object)</a>             | Removes the specified <a href="#">SqlParameter</a> from the collection. |

## Remove(SqlParameter) Remove(SqlParameter)

Removes the specified [SqlParameter](#) from the collection.

```
public void Remove (System.Data.SqlClient.SqlParameter value);
override this.Remove : System.Data.SqlClient.SqlParameter -> unit
```

Parameters

value [SqlParameter](#) [SqlParameter](#)

A [SqlParameter](#) object to remove from the collection.

Exceptions

[InvalidOperationException](#) [InvalidOperationException](#)

The parameter is not a [SqlParameter](#).

[SystemException](#) [SystemException](#)

The parameter does not exist in the collection.

Examples

The following example searches for a [SqlParameter](#) object in a [SqlParameterCollection](#) collection. If the parameter exists, the example removes it. This example assumes that a [SqlParameterCollection](#) collection has already been created by a [SqlCommand](#).

```
public void SearchSqlParams()
{
    // ...
    // create SqlCommand command and SqlParameter param
    // ...
    if (command.Parameters.Contains(param))
        command.Parameters.Remove(param);
}
```

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)  
[ADO.NET Overview](#)

## Remove(Object) Remove(Object)

Removes the specified [SqlParameter](#) from the collection.

```
public override void Remove (object value);  
override this.Remove : obj -> unit
```

Parameters

|       |                        |
|-------|------------------------|
| value | <a href="#">Object</a> |
|-------|------------------------|

The object to remove from the collection.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.RemoveAt SqlParameterCollection.RemoveAt

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">RemoveAt(Int32)</a> <a href="#">RemoveAt(Int32)</a>   | Removes the <a href="#">SqlParameter</a> from the <a href="#">SqlParameterCollection</a> at the specified index.          |
| <a href="#">RemoveAt(String)</a> <a href="#">RemoveAt(String)</a> | Removes the <a href="#">SqlParameter</a> from the <a href="#">SqlParameterCollection</a> at the specified parameter name. |

### RemoveAt(Int32) RemoveAt(Int32)

Removes the [SqlParameter](#) from the [SqlParameterCollection](#) at the specified index.

```
public override void RemoveAt (int index);  
override this.RemoveAt : int -> unit
```

Parameters

index Int32 Int32

The zero-based index of the [SqlParameter](#) object to remove.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

### RemoveAt(String) RemoveAt(String)

Removes the [SqlParameter](#) from the [SqlParameterCollection](#) at the specified parameter name.

```
public override void RemoveAt (string parameterName);  
override this.RemoveAt : string -> unit
```

Parameters

parameterName String String

The name of the [SqlParameter](#) to remove.

See

[Commands and Parameters \(ADO.NET\)](#)

Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlParameterCollection.SyncRoot SqlParameterCollection.SyncRoot

## In this Article

Gets an object that can be used to synchronize access to the [SqlParameterCollection](#).

```
public override object SyncRoot { get; }  
member this.SyncRoot : obj
```

## Returns

[Object Object](#)

An object that can be used to synchronize access to the [SqlParameterCollection](#).

## See

[Commands and Parameters \(ADO.NET\)](#)

## Also

[DataAdapter Parameters \(ADO.NET\)](#)

[Using the .NET Framework Data Provider for SQL Server](#)

[ADO.NET Overview](#)

# SqlProviderServices SqlProviderServices Class

The DbProviderServices implementation for the SqlClient provider for SQL Server.

## Declaration

```
[System.CLSCompliant(false)]
public sealed class SqlProviderServices : System.Data.Common.DbProviderServices

type SqlProviderServices = class
    inherit DbProviderServices
```

## Inheritance Hierarchy



## Properties

SingletonInstance

SingletonInstance

Gets the singleton instance of [SqlProviderServices](#).

# SqlProviderServices.SingletonInstance SqlProviderServices.SingletonInstance

## In this Article

Gets the singleton instance of [SqlProviderServices](#).

```
public static System.Data.SqlClient.SqlProviderServices SingletonInstance { get; }  
member this.SingletonInstance : System.Data.SqlClient.SqlProviderServices
```

Returns

[SqlProviderServices](#) [SqlProviderServices](#)

Returns [SqlProviderServices](#).

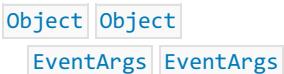
# SqlRowsCopiedEventArgs Class

Represents the set of arguments passed to the [SqlRowsCopiedEventHandler](#).

## Declaration

```
public class SqlRowsCopiedEventArgs : EventArgs  
type SqlRowsCopiedEventArgs = class  
    inherit EventArgs
```

## Inheritance Hierarchy



## Constructors

`SqlRowsCopiedEventArgs(Int64)`

`SqlRowsCopiedEventArgs(Int64)`

Creates a new instance of the [SqlRowsCopiedEventArgs](#) object.

## Properties

`Abort`

`Abort`

Gets or sets a value that indicates whether the bulk copy operation should be aborted.

`RowsCopied`

`RowsCopied`

Gets a value that returns the number of rows copied during the current bulk copy operation.

## See Also

# SqlRowsCopiedEventArgs.Abort SqlRowsCopiedEventArgs.Abort

## In this Article

Gets or sets a value that indicates whether the bulk copy operation should be aborted.

```
public bool Abort { get; set; }  
member this.Abandon : bool with get, set
```

## Returns

**Boolean** Boolean

`true` if the bulk copy operation should be aborted; otherwise `false`.

## Remarks

Use the [Abort](#) property to cancel a bulk copy operation. Set [Abort](#) to `true` to abort the bulk copy operation.

If you call the [Close](#) method from [SqlRowsCopied](#), an exception is generated, and the [SqlBulkCopy](#) object state does not change.

If an application specifically creates a [SqlTransaction](#) object in the [SqlCommand](#) constructor, the transaction is not rolled back. The application is responsible for determining whether it is required to rollback the operation, and if so, it must call the [SqlTransaction.Rollback](#) method. If the application does not create a transaction, the internal transaction corresponding to the current batch is automatically rolled back. However, changes related to previous batches within the bulk copy operation are retained, because the transactions for them already have been committed.

See

[ADO.NET Overview](#)

Also

# SqlRowsCopiedEventArgs.RowsCopied SqlRowsCopiedEventArgs.RowsCopied

## In this Article

Gets a value that returns the number of rows copied during the current bulk copy operation.

```
public long RowsCopied { get; }  
member this.RowsCopied : int64
```

Returns

[Int64](#) [Int64](#)

`int` that returns the number of rows copied.

## Remarks

The value in the [RowsCopied](#) property is reset on each call to any of the [SqlBulkCopy.WriteToServer](#) methods.

See

[ADO.NET Overview](#)

Also

# SqlRowsCopiedEventArgs

## In this Article

Creates a new instance of the [SqlRowsCopiedEventArgs](#) object.

```
public SqlRowsCopiedEventArgs (long rowsCopied);  
new System.Data.SqlClient.SqlRowsCopiedEventArgs : int64 ->  
System.Data.SqlClient.SqlRowsCopiedEventArgs
```

## Parameters

|            |                       |
|------------|-----------------------|
| rowsCopied | <a href="#">Int64</a> |
|------------|-----------------------|

An [Int64](#) that indicates the number of rows copied during the current bulk copy operation.

## Remarks

The value in the `rowsCopied` parameter is reset on each call to any one of the [SqlBulkCopy.WriteToServer](#) methods.

### See

[ADO.NET Overview](#)

### Also

# SqlRowsCopiedEventHandler SqlRowsCopiedEvent Handler Delegate

Represents the method that handles the [SqlRowsCopied](#) event of a [SqlBulkCopy](#).

## Declaration

```
public delegate void SqlRowsCopiedEventHandler(object sender, SqlRowsCopiedEventArgs e);  
type SqlRowsCopiedEventHandler = delegate of obj * SqlRowsCopiedEventArgs -> unit
```

## Inheritance Hierarchy

```
Object Object  
Delegate Delegate
```

## See Also

# SqlRowUpdatedEventArgs Class

Provides data for the [RowUpdated](#) event.

## Declaration

```
public sealed class SqlRowUpdatedEventArgs : System.Data.Common.RowUpdatedEventArgs  
type SqlRowUpdatedEventArgs = class  
    inherit RowUpdatedEventArgs
```

## Inheritance Hierarchy



## Remarks

The [RowUpdated](#) event is raised when an [Update](#) to a row is completed.

When using [Update](#), there are two events that occur for each data row updated. The order of execution is as follows:

1. The values in the [DataRow](#) are moved to the parameter values.
2. The [OnRowUpdating](#) event is raised.
3. The command executes.
4. If the command is set to [FirstReturnedRecord](#), and the first returned result is placed in the [DataRow](#).
5. If there are output parameters, they are placed in the [DataRow](#).
6. The [OnRowUpdated](#) event is raised.
7. [AcceptChanges](#) is called.

## Constructors

```
SqlRowUpdatedEventArgs(DataRow, IDbCommand, StatementType, DataTableMapping)
```

```
SqlRowUpdatedEventArgs(DataRow, IDbCommand, StatementType, DataTableMapping)
```

Initializes a new instance of the [SqlRowUpdatedEventArgs](#) class.

## Properties

[Command](#)

[Command](#)

Gets or sets the [SqlCommand](#) executed when [Update\(DataSet\)](#) is called.

## See Also

# SqlRowUpdatedEventArgs.Command SqlRowUpdatedEventArgs.Command

## In this Article

Gets or sets the [SqlCommand](#) executed when [Update\(DataSet\)](#) is called.

```
public System.Data.SqlClient.SqlCommand Command { get; }
```

```
member this.Command : System.Data.SqlClient.SqlCommand
```

## Returns

[SqlCommand](#) [SqlCommand](#)

The [SqlCommand](#) executed when [Update\(DataSet\)](#) is called.

See

[ADO.NET Overview](#)

Also

# SqlRowUpdatedEventArgs SqlRowUpdatedEventArgs

## In this Article

Initializes a new instance of the [SqlRowUpdatedEventArgs](#) class.

```
public SqlRowUpdatedEventArgs (System.Data.DataRow row, System.Data.IDbCommand command,
System.Data.StatementType statementType, System.Data.Common.DataTableMapping tableMapping);

new System.Data.SqlClient.SqlRowUpdatedEventArgs : System.Data.DataRow * System.Data.IDbCommand *
System.Data.StatementType * System.Data.Common.DataTableMapping ->
System.Data.SqlClient.SqlRowUpdatedEventArgs
```

## Parameters

row [DataRow](#) [DataRow](#)

The [DataRow](#) sent through an [Update\(DataSet\)](#).

command [IDbCommand](#) [IDbCommand](#)

The [IDbCommand](#) executed when [Update\(DataSet\)](#) is called.

statementType [StatementType](#) [StatementType](#)

One of the [StatementType](#) values that specifies the type of query executed.

tableMapping [DataTableMapping](#) [DataTableMapping](#)

The [DataTableMapping](#) sent through an [Update\(DataSet\)](#).

See [ADO.NET Overview](#)

Also

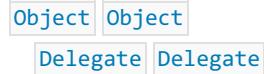
# SqlRowUpdatedEventHandler SqlRowUpdatedEvent Handler Delegate

Represents the method that will handle the [RowUpdated](#) event of a [SqlDataAdapter](#).

## Declaration

```
public delegate void SqlRowUpdatedEventHandler(object sender, SqlRowUpdatedEventArgs e);  
type SqlRowUpdatedEventHandler = delegate of obj * SqlRowUpdatedEventArgs -> unit
```

## Inheritance Hierarchy



## Remarks

The handler is not required to perform any action, and your code should avoid generating exceptions or allowing exceptions to propagate to the calling method. Any exceptions that do reach the caller are ignored.

When you create a [SqlRowUpdatedEventArgs](#) delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, unless you remove the delegate. For more information about event handler delegates, see [Handling and Raising Events](#).

## See Also

# SqlRowUpdatingEventArgs Class

Provides data for the [RowUpdating](#) event.

## Declaration

```
public sealed class SqlRowUpdatingEventArgs : System.Data.Common.RowUpdatingEventArgs  
type SqlRowUpdatingEventArgs = class  
    inherit RowUpdatingEventArgs
```

## Inheritance Hierarchy



## Remarks

The [RowUpdating](#) event is raised before an [Update](#) to a row.

When you are using [Update](#), there are two events that occur for each data row updated. The order of execution is as follows:

1. The values in the [DataRow](#) are moved to the parameter values.
2. The [OnRowUpdating](#) event is raised.
3. The command executes.
4. If the command is set to `FirstReturnedRecord`, and the first returned result is placed in the [DataRow](#).
5. If there are output parameters, they are placed in the [DataRow](#).
6. The [OnRowUpdated](#) event is raised.
7. [AcceptChanges](#) is called.

## Constructors

```
SqlRowUpdatingEventArgs(DataRow, IDbCommand, StatementType, DataTableMapping)
```

```
SqlRowUpdatingEventArgs(DataRow, IDbCommand, StatementType, DataTableMapping)
```

Initializes a new instance of the [SqlRowUpdatingEventArgs](#) class.

## Properties

[Command](#)

[Command](#)

Gets or sets the [SqlCommand](#) to execute when performing the [Update\(DataSet\)](#).

## See Also

# SqlRowUpdatingEventArgs.Command SqlRowUpdatingEventArgs.Command

## In this Article

Gets or sets the [SqlCommand](#) to execute when performing the [Update\(DataSet\)](#).

```
public System.Data.SqlClient.SqlCommand Command { get; set; }  
member this.Command : System.Data.SqlClient.SqlCommand with get, set
```

## Returns

[SqlCommand](#) [SqlCommand](#)

The [SqlCommand](#) to execute when performing the [Update\(DataSet\)](#).

See

[ADO.NET Overview](#)

Also

# SqlRowUpdatingEventArgs SqlRowUpdatingEventArgs

## In this Article

Initializes a new instance of the [SqlRowUpdatingEventArgs](#) class.

```
public SqlRowUpdatingEventArgs (System.Data.DataRow row, System.Data.IDbCommand command,
System.Data.StatementType statementType, System.Data.Common.DataTableMapping tableMapping);

new System.Data.SqlClient.SqlRowUpdatingEventArgs : System.Data.DataRow * System.Data.IDbCommand *
System.Data.StatementType * System.Data.Common.DataTableMapping ->
System.Data.SqlClient.SqlRowUpdatingEventArgs
```

## Parameters

row [DataRow](#) [DataRow](#)

The [DataRow](#) to [Update\(DataSet\)](#).

command [IDbCommand](#) [IDbCommand](#)

The [IDbCommand](#) to execute during [Update\(DataSet\)](#).

statementType [StatementType](#) [StatementType](#)

One of the [StatementType](#) values that specifies the type of query executed.

tableMapping [DataTableMapping](#) [DataTableMapping](#)

The [DataTableMapping](#) sent through an [Update\(DataSet\)](#).

See [ADO.NET Overview](#)

Also

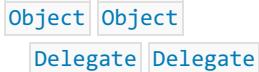
# SqlRowUpdatingEventHandler SqlRowUpdatingEvent Handler Delegate

Represents the method that will handle the [RowUpdating](#) event of a [SqlDataAdapter](#).

## Declaration

```
public delegate void SqlRowUpdatingEventHandler(object sender, SqlRowUpdatingEventArgs e);  
type SqlRowUpdatingEventHandler = delegate of obj * SqlRowUpdatingEventArgs -> unit
```

## Inheritance Hierarchy



## Remarks

The handler is not required to perform any action, and your code should avoid generating exceptions or allowing exceptions to propagate to the calling method. Any exceptions that do reach the caller are ignored.

The handler may use the [SqlRowUpdatingEventArgs](#) to influence the processing of the updates. For example, the handler may opt to skip the update of the current row or skip the update of all remaining rows. Note that the rows are updated in the order that they were received from the data source.

When you create a [SqlRowUpdatingEventArgs](#) delegate, you identify the method that will handle the event. To associate the event with your event handler, add an instance of the delegate to the event. The event handler is called whenever the event occurs, unless you remove the delegate. For more information about event handler delegates, see [Handling and Raising Events](#).

## See Also

# SqlTransaction SqlTransaction Class

Represents a Transact-SQL transaction to be made in a SQL Server database. This class cannot be inherited.

## Declaration

```
public sealed class SqlTransaction : System.Data.Common.DbTransaction, IDisposable  
  
type SqlTransaction = class  
    inherit DbTransaction  
    interface IDbTransaction  
    interface IDisposable
```

## Inheritance Hierarchy

```
Object Object  
MarshalByRefObject MarshalByRefObject
```

## Remarks

The application creates a [SqlTransaction](#) object by calling [BeginTransaction](#) on the [SqlConnection](#) object. All subsequent operations associated with the transaction (for example, committing or aborting the transaction), are performed on the [SqlTransaction](#) object.

### Note

[Try/Catch](#) exception handling should always be used when committing or rolling back a [SqlTransaction](#). Both [Commit](#) and [Rollback](#) generate an [InvalidOperationException](#) if the connection is terminated or if the transaction has already been rolled back on the server.

For more information on SQL Server transactions, see [Explicit Transactions](#) and [Coding Efficient Transactions](#).

## Properties

### Connection

#### Connection

Gets the [SqlConnection](#) object associated with the transaction, or `null` if the transaction is no longer valid.

### IsolationLevel

#### IsolationLevel

Specifies the [IsolationLevel](#) for this transaction.

## Methods

### Commit()

#### Commit()

Commits the database transaction.

### Dispose()

#### Dispose()

`Rollback()`

`Rollback()`

Rolls back a transaction from a pending state.

`Rollback(String)`

`Rollback(String)`

Rolls back a transaction from a pending state, and specifies the transaction or savepoint name.

`Save(String)`

`Save(String)`

Creates a savepoint in the transaction that can be used to roll back a part of the transaction, and specifies the savepoint name.

## See Also

# SqlTransaction.Commit SqlTransaction.Commit

## In this Article

Commits the database transaction.

```
public override void Commit ();  
override this.Commit : unit -> unit
```

Exceptions

[Exception](#) [Exception](#)

An error occurred while trying to commit the transaction.

[InvalidOperationException](#) [InvalidOperationException](#)

The transaction has already been committed or rolled back.

-or-

The connection is broken.

## Examples

The following example creates a [SqlConnection](#) and a [SqlTransaction](#). It also demonstrates how to use the [Commit](#), [BeginTransaction](#), and [Rollback](#) methods. The transaction is rolled back on any error. [Try/Catch](#) error handling is used to handle any errors when attempting to commit or roll back the transaction.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction("SampleTransaction");

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();

            // Attempt to commit the transaction.
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Commit Exception Type: {0}", ex.GetType());
            Console.WriteLine(" Message: {0}", ex.Message);

            // Attempt to roll back the transaction.
            try
            {
                transaction.Rollback();
            }
            catch (Exception ex2)
            {
                // This catch block will handle any errors that may have occurred
                // on the server that would cause the rollback to fail, such as
                // a closed connection.
                Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
                Console.WriteLine(" Message: {0}", ex2.Message);
            }
        }
    }
}

```

## Remarks

The [Commit](#) method is equivalent to the Transact-SQL COMMIT TRANSACTION statement. You cannot roll back a transaction once it has been committed, because all modifications have become a permanent part of the database. For more information, see [COMMIT TRANSACTION \(Transact-SQL\)](#).

### [Note](#)

[Try/Catch](#) exception handling should always be used when committing or rolling back a [SqlTransaction](#). Both [Commit](#) and [Rollback](#) generates an [InvalidOperationException](#) if the connection is terminated or if the transaction has already been rolled back on the server.

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

See

Also

[Local Transactions](#)

[ADO.NET Overview](#)

# SqlTransaction.Connection SqlTransaction.Connection

## In this Article

Gets the [SqlConnection](#) object associated with the transaction, or `null` if the transaction is no longer valid.

```
public System.Data.SqlClient.SqlConnection Connection { get; }  
member this.Connection : System.Data.SqlClient.SqlConnection
```

Returns

[SqlConnection](#) [SqlConnection](#)

The [SqlConnection](#) object associated with the transaction.

## Remarks

A single application may have multiple database connections, each with zero or more transactions. This property lets you determine the connection object associated with a particular transaction created by [BeginTransaction](#).

See

[Performing a Transaction](#)

Also

[ADO.NET Overview](#)

# SqlTransaction.Dispose SqlTransaction.Dispose

## In this Article

```
public void Dispose ();  
  
abstract member Dispose : unit -> unit  
override this.Dispose : unit -> unit
```

# SqlTransaction.IsolationLevel SqlTransaction.IsolationLevel

## In this Article

Specifies the [IsolationLevel](#) for this transaction.

```
public override System.Data.IsolationLevel IsolationLevel { get; }  
member this.IsolationLevel : System.Data.IsolationLevel
```

## Returns

[IsolationLevel](#) [IsolationLevel](#)

The [IsolationLevel](#) for this transaction. The default is [ReadCommitted](#).

## Remarks

Parallel transactions are not supported. Therefore, the [IsolationLevel](#) applies to the whole transaction.

For more information on SQL Server isolation levels, see [Transaction Isolation Levels](#).

See

[Performing a Transaction](#)

Also

[ADO.NET Overview](#)

# SqlTransaction.Rollback SqlTransaction.Rollback

In this Article

## Overloads

|   |   |
|---|---|
| <a href="#">Rollback()</a> <a href="#">Rollback()</a>             | Rolls back a transaction from a pending state.  |
| <a href="#">Rollback(String)</a> <a href="#">Rollback(String)</a> | Rolls back a transaction from a pending state, and specifies the transaction or savepoint name. |

## Rollback() Rollback()

Rolls back a transaction from a pending state.

```
public override void Rollback ();
override this.R rollback : unit -> unit
```

Exceptions

[Exception](#) [Exception](#)

An error occurred while trying to commit the transaction.

[InvalidOperationException](#) [InvalidOperationException](#)

The transaction has already been committed or rolled back.

-or-

The connection is broken.

Examples

The following example creates a [SqlConnection](#) and a [SqlTransaction](#). It also demonstrates how to use the [BeginTransaction](#), [Commit](#), and [Rollback](#) methods. The transaction is rolled back on any error. [Try / Catch](#) error handling is used to handle any errors when attempting to commit or roll back the transaction.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction("SampleTransaction");

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();

            // Attempt to commit the transaction.
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Commit Exception Type: {0}", ex.GetType());
            Console.WriteLine(" Message: {0}", ex.Message);

            // Attempt to roll back the transaction.
            try
            {
                transaction.Rollback();
            }
            catch (Exception ex2)
            {
                // This catch block will handle any errors that may have occurred
                // on the server that would cause the rollback to fail, such as
                // a closed connection.
                Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
                Console.WriteLine(" Message: {0}", ex2.Message);
            }
        }
    }
}

```

## Remarks

The [Rollback](#) method is equivalent to the Transact-SQL ROLLBACK TRANSACTION statement. For more information, see [ROLLBACK TRANSACTION \(Transact-SQL\)](#).

The transaction can only be rolled back from a pending state (after [BeginTransaction](#) has been called, but before [Commit](#) is called). The transaction is rolled back in the event it is disposed before [Commit](#) or [Rollback](#) is called.

#### Note

`Try/Catch` exception handling should always be used when rolling back a transaction. A `Rollback` generates an `InvalidOperationException` if the connection is terminated or if the transaction has already been rolled back on the server.

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

See

[Local Transactions](#)

Also

[ADO.NET Overview](#)

## Rollback(String) Rollback(String)

Rolls back a transaction from a pending state, and specifies the transaction or savepoint name.

```
public void Rollback (string transactionName);  
override this.Rollback : string -> unit
```

Parameters

transactionName

`String` `String`

The name of the transaction to roll back, or the savepoint to which to roll back.

Exceptions

[ArgumentException](#) [ArgumentNullException](#)

No transaction name was specified.

[InvalidOperationException](#) [InvalidOperationException](#)

The transaction has already been committed or rolled back.

-or-

The connection is broken.

Examples

The following example creates a `SqlConnection` and a `SqlTransaction`. It also demonstrates how to use the `BeginTransaction`, `Commit`, and `Rollback` methods. The transaction is rolled back on any error. `Try/Catch` error handling is used to handle any errors when attempting to commit or roll back the transaction.

```

private static void ExecuteSqlTransaction(string connectionString)
{
    using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();

        SqlCommand command = connection.CreateCommand();
        SqlTransaction transaction;

        // Start a local transaction.
        transaction = connection.BeginTransaction("SampleTransaction");

        // Must assign both transaction object and connection
        // to Command object for a pending local transaction
        command.Connection = connection;
        command.Transaction = transaction;

        try
        {
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (100, 'Description')";
            command.ExecuteNonQuery();
            command.CommandText =
                "Insert into Region (RegionID, RegionDescription) VALUES (101, 'Description')";
            command.ExecuteNonQuery();

            // Attempt to commit the transaction.
            transaction.Commit();
            Console.WriteLine("Both records are written to database.");
        }
        catch (Exception ex)
        {
            Console.WriteLine("Commit Exception Type: {0}", ex.GetType());
            Console.WriteLine(" Message: {0}", ex.Message);

            // Attempt to roll back the transaction.
            try
            {
                transaction.Rollback();
            }
            catch (Exception ex2)
            {
                // This catch block will handle any errors that may have occurred
                // on the server that would cause the rollback to fail, such as
                // a closed connection.
                Console.WriteLine("Rollback Exception Type: {0}", ex2.GetType());
                Console.WriteLine(" Message: {0}", ex2.Message);
            }
        }
    }
}

```

## Remarks

The [Rollback](#) method is equivalent to the Transact-SQL ROLLBACK TRANSACTION statement. For more information, see [Transactions \(Transact-SQL\)](#).

The transaction can only be rolled back from a pending state (after [BeginTransaction](#) has been called, but before [Commit](#) is called). The transaction is rolled back if it is disposed before [Commit](#) or [Rollback](#) is called.

**Note**

`Try/Catch` exception handling should always be used when rolling back a transaction. A `Rollback` generates an `InvalidOperationException` if the connection is terminated or if the transaction has already been rolled back on the server.

For more information on SQL Server transactions, see [Transactions \(Transact-SQL\)](#).

See

Also

[Performing a Transaction](#)

[ADO.NET Overview](#)

# SqlTransaction.Save SqlTransaction.Save

## In this Article

Creates a savepoint in the transaction that can be used to roll back a part of the transaction, and specifies the savepoint name.

```
public void Save (string savePointName);  
member this.Save : string -> unit
```

## Parameters

savePointName String String

The name of the savepoint.

## Exceptions

[Exception](#) [Exception](#)

An error occurred while trying to commit the transaction.

[InvalidOperationException](#) [InvalidOperationException](#)

The transaction has already been committed or rolled back.

-or-

The connection is broken.

## Remarks

[Save](#) method is equivalent to the Transact-SQL SAVE TRANSACTION statement.

The value used in the `savePoint` parameter can be the same value used in the `transactionName` parameter of some implementations of the [BeginTransaction](#) method.

Savepoints offer a mechanism to roll back parts of transactions. You create a savepoint using the [Save](#) method, and then later call the [Rollback](#) method to roll back to the savepoint instead of rolling back to the start of the transaction.

See

[Performing a Transaction](#)

Also

[ADO.NET Overview](#)