EDB Postgres™ Advanced Server .NET Connector Guide

Connectors Release 10.0.3
.NET 4.0 Connector Version 2.2.4.5
.NET 4.5/4.5.1 Connector Version 3.2.6.1
.NET Standard 1.3/2.0 Connector Version 3.2.6.1

July 24, 2018

EDB Postgres™ Advanced Server .NET Connector Guide
by EnterpriseDB® Corporation
Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

EnterpriseDB Corporation, 34 Crosby Drive Suite 201, Bedford, MA 01730, USA
T +1 781 357 3390 **F** +1 978 467 1307 **E** info@enterprisedb.com **www**.enterprisedb.com

Table of Contents

1 Introduction 5
1.1 Typographical Conventions Used in this Guide 6
2 The Advanced Server .NET Connector - Overview 7
2.1 The .NET Class Hierarchy 7
2.2 Installing the .NET Connector 8
3 Configuring the .NET Connector 12
3.1 General Instructions 12
3.2 .NET Framework Setup 13
3.2.1 .NET Framework 4.0 13
3.2.2 .NET Framework 4.5 13
3.2.3 .NET Framework 4.5.1 14
3.2.4 .NET Standard 1.3 14
3.2.5 .NET Standard 2.0 15
3.3 Entity Framework 5/6 15
3.4 EnterpriseDB VSIX for Visual Studio 2015/2017 16
3.4.1 Installation and Configuration for Visual Studio 2015/2017 17
3.4.2 Model First and Database First Usage 17
4 Using the .NET Connector 20
4.1 Opening a Database Connection 20
4.1.1 Connection String Parameters 21
4.1.2 Example - Opening a Database Connection using ASP.NET 24
4.1.3 Example - Opening a Database Connection from a Console Application 25
4.1.4 Example - Opening a Database Connection from a Windows Form Application 27
4.2 Retrieving Database Records 29
4.2.1 Retrieving a Single Database Record 31
4.3 Parameterized Queries 32
4.4 Inserting Records in a Database 34
4.5 Deleting Records in a Database 36
4.6 Using SPL Stored Procedures in your .NET Application 38
4.6.1 Example - Executing a Stored Procedure without Parameters 39
4.6.2 Example - Executing a Stored Procedure with IN Parameters 41

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.6.3 Example - Executing a Stored Procedure with IN, OUT, and INOUT Parameters 44
4.7 Using a Ref Cursor in a .NET Application 48

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

1 Introduction

The EDB Postgres™ Advanced Server .NET Connector distributed with EDB Postgres™ Advanced Server (Advanced Server) provides connectivity between a .NET client application and an Advanced Server database server. This guide provides installation instructions, usage instructions, and examples that demonstrate the functionality of the Advanced Server .NET Connector:

- How to connect to an instance of Advanced Server.
- How to retrieve information from an Advanced Server database.
- How to update information stored on an Advanced Server database.

This document assumes that you have a solid working knowledge of both C# and .NET. The Advanced Server .NET Connector functionality is built on the core functionality of the Npgsql open source project. The Npgsql User's Manual is available online at:

http://www.npgsql.org/doc/index.html

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

1.1 Typographical Conventions Used in this Guide

Certain typographical conventions are used in this manual to clarify the meaning and usage of various commands, statements, programs, examples, etc. This section provides a summary of these conventions.

In the following descriptions a *term* refers to any word or group of words that are language keywords, user-supplied values, literals, etc. A term's exact meaning depends upon the context in which it is used.

- *Italic font* introduces a new term, typically, in the sentence that defines it for the first time.
- `Fixed-width (mono-spaced) font` is used for terms that must be given literally such as SQL commands, specific table and column names used in the examples, programming language keywords, etc. For example, `SELECT * FROM emp;`
- *`Italic fixed-width font`* is used for terms for which the user must substitute values in actual usage. For example, `DELETE FROM` *`table_name`*;
- A vertical pipe | denotes a choice between the terms on either side of the pipe. A vertical pipe is used to separate two or more alternative terms within square brackets (optional choices) or braces (one mandatory choice).
- Square brackets [ ] denote that one or none of the enclosed term(s) may be substituted. For example, `[ a | b ]`, means choose one of "`a`" or "`b`" or neither of the two.
- Braces {} denote that exactly one of the enclosed alternatives must be specified. For example, `{ a | b }`, means exactly one of "`a`" or "`b`" must be specified.
- Ellipses ... denote that the proceeding term may be repeated. For example, `[ a | b ] ...` means that you may have the sequence, "`b a a b a`".

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

2 The Advanced Server .NET Connector - Overview

The Advanced Server .NET Connector is a .NET data provider that allows a client application to connect to a database stored on an Advanced Server host. The .NET Connector accesses the data directly, allowing the client application optimal performance, a broad spectrum of functionality, and access to Advanced Server features.

The .NET Connector supports .NET Framework versions 4.0 and 4.5.1, and Entity Framework 6.

2.1 The .NET Class Hierarchy

The .NET Class Hierarchy contains a number of classes that you can use to create objects that control a connection to the Advanced Server database and manipulate the data stored on the server. The following are just a few of the most commonly used object classes:

`EDBConnection`

> The `EDBConnection` class represents a connection to Advanced Server. An `EDBConnection` object contains a `ConnectionString` that instructs the .NET client how to connect to an Advanced Server database.

`EDBCommand`

> An `EDBCommand` object contains an SQL command that the client will execute against Advanced Server. Before you can execute an `EDBCommand` object, you must link it to an `EDBConnection` object.

`EDBDataReader`

> An `EDBDataReader` object provides a way to read an Advanced Server result set. You can use an `EDBDataReader` object to step through one row at a time, forward-only.

`EDBDataAdapter`

> An `EDBDataAdapter` object links a result set to the Advanced Server database. You can modify values and use the `EDBDataAdapter` class to update the data stored in an Advanced Server database.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

2.2 Installing the .NET Connector

You can use the EnterpriseDB Connectors Installation wizard to add the .NET Connector to your system; the installer is available at:

https://www.enterprisedb.com/advanced-downloads

After downloading the installer, right-click on the installer icon, and select `Run As Administrator` from the context menu. The installation wizard opens as shown in Figure 2.1.

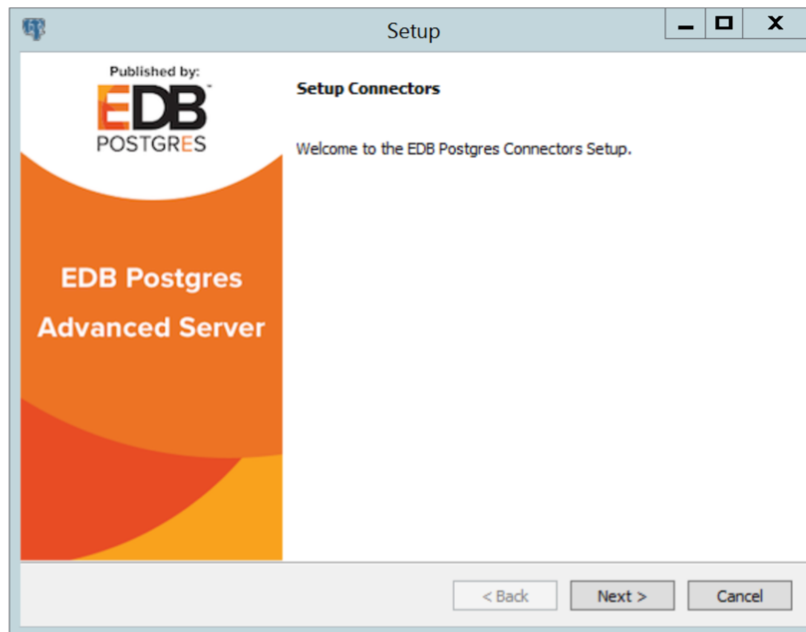


Figure 2.1 - The Connectors Installation wizard.

Click `Next` to continue.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

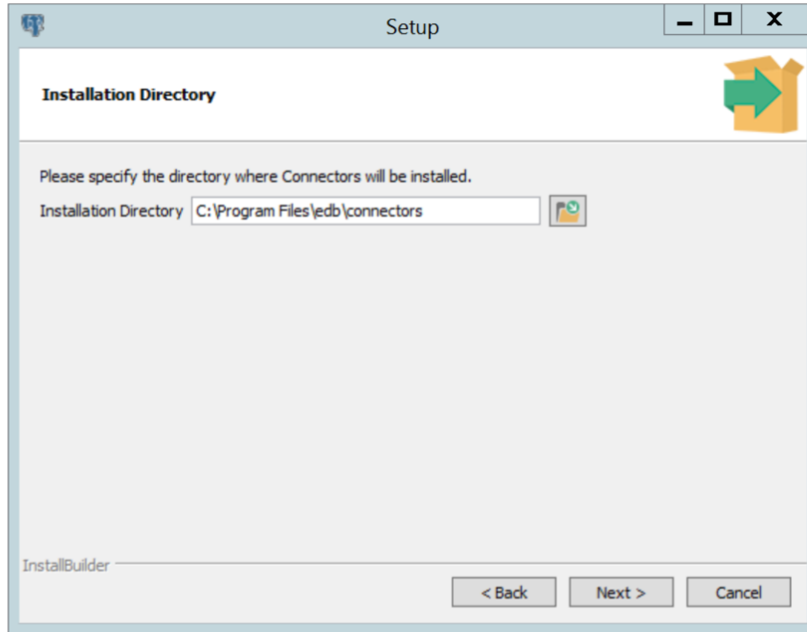


Figure 2.2- The Installation dialog.

Use the `Installation Directory` dialog (see Figure 2.2) to specify the directory in which the connector will be installed, and click `Next` to continue.

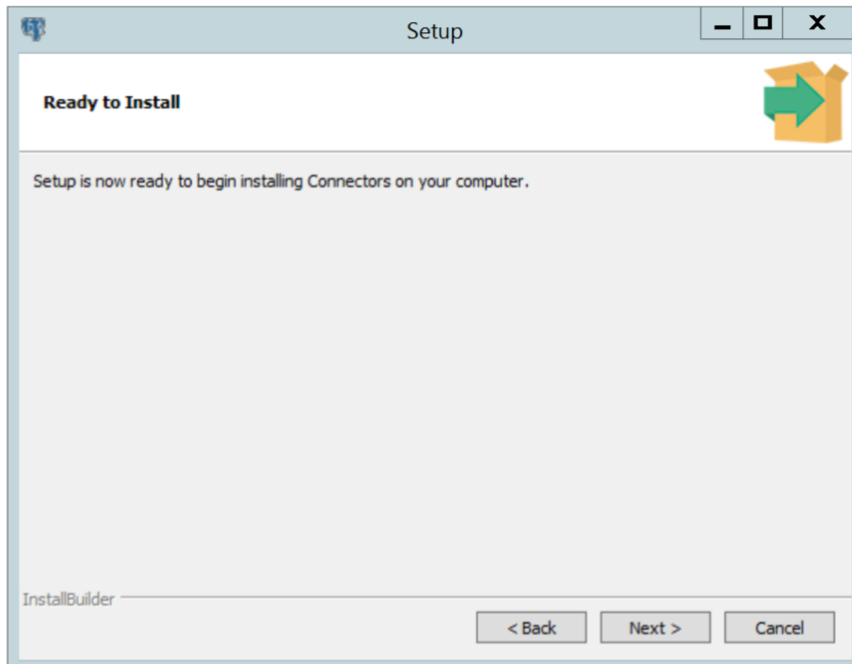


Figure 2.3- The Ready to Install dialog.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Click `Next` on the `Ready to Install` dialog (see Figure 2.3) to start the installation; popup dialogs confirm the progress of the installation wizard.

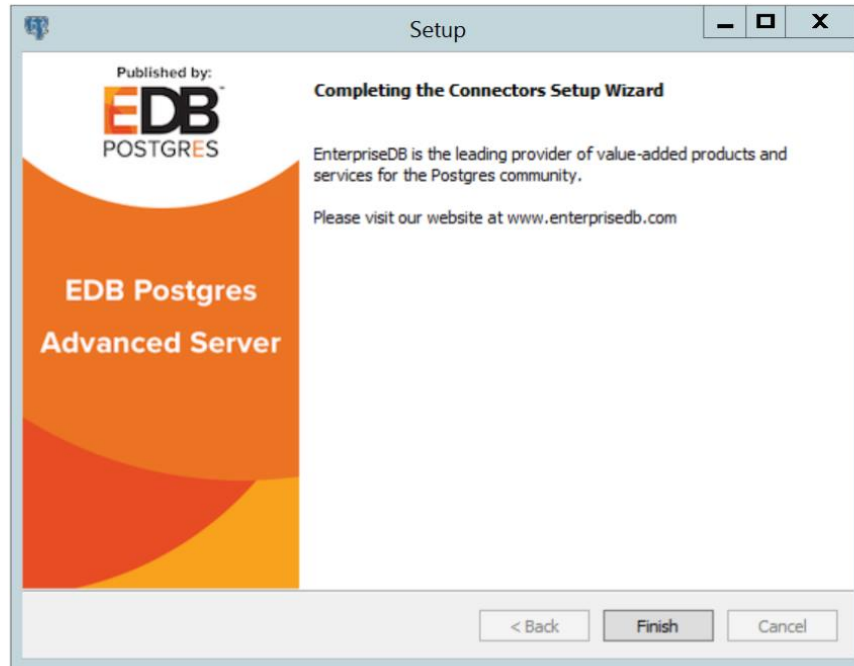


Figure 2.4 - The installation is complete.

When the wizard informs you that it has completed the setup, click the `Finish` button to exit the dialog (see Figure 2.4).

You can use StackBuilder Plus to add or update the connector on an existing Advanced Server installation; to open StackBuilder Plus, select StackBuilder Plus from the Windows `Apps` menu (see Figure 2.5).

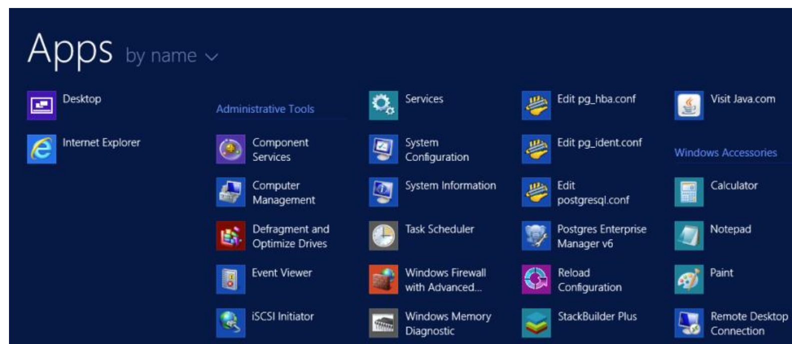


Figure 2.5 - Starting StackBuilder Plus.

When StackBuilder Plus opens, follow the onscreen instructions, selecting the `EnterpriseDB Connectors` option from the `Database Drivers` node of the tree control (see Figure 2.6).

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

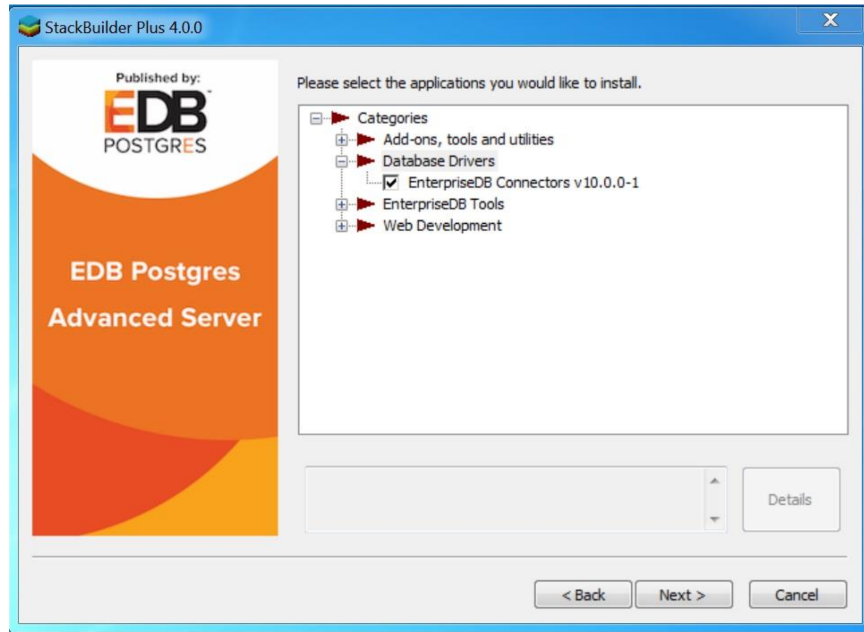


Figure 2.6 - Selecting the Connectors installer.

Follow the directions of the onscreen wizard to add or update an installation of the EnterpriseDB Connectors.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

3 Configuring the .NET Connector

The following sections describe how to configure the .NET Connector depending upon the framework version to be used:

- **General Instructions.** General instructional information applicable to all components
- **.NET Framework 4.0.** Instructions for configuring for usage with .NET Framework 4.0
- **.NET Framework 4.5.** Instructions for configuring for usage with .NET Framework 4.5
- **.NET Framework 4.5.1.** Instructions for configuring for usage with .NET Framework 4.5.1
- **.NET Standard 1.3.** Instructions for configuring for usage with .NET Standard 1.3
- **.NET Standard 2.0.** Instructions for configuring for usage with .NET Standard 2.0
- **Entity Framework 5/6.** Instructions for configuring for usage with Entity Framework
- **EnterpriseDB VSIX.** Instructions for configuring for usage with EnterpriseDB VSIX

3.1 General Instructions

To reference library files using Microsoft Visual Studio:

1. Select the project in the `Solution Explorer`.
2. Select `Add Reference` from the `Project` menu.
3. When the `Add Reference` dialog box opens, browse to select the appropriate library files.

Optionally, the library files can be copied to the specified location.

Before you can use an Advanced Server .NET class, you must import the namespace into your program. Importing a namespace makes the compiler aware of the classes available within the namespace. The namespace is the following:

```
EnterpriseDB.EDBClient
```

If you are using Entity Framework 6, the additional namespace is the following:

```
EntityFramework6.EntepriseDB.EDBClient
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

The method you use to include the namespace varies by the type of application you are writing. For example, the following command imports a namespace into an ASP.NET page:

```
<% import namespace="EnterpriseDB.EDBClient" %>
```

To import a namespace into a C# application, write:

```
using EnterpriseDB.EDBClient;
```

3.2 .NET Framework Setup

The following sections describe the setup for various .NET versions.

3.2.1 .NET Framework 4.0

To set up .NET Connector for usage with .NET Framework 4.0, reference the shared library files located in the following directory:

```
C:\Program Files\edb\connectors\dotnet\net40\
```

The shared library files are the following:

```
EDBDataProvider.2.0.2.dll
Mono.Security.dll
```

See Section 3.1 for information on referencing library files.

Depending upon the type of application, import the namespace into the source code (see Section 3.1).

3.2.2 .NET Framework 4.5

If you are using .NET Framework version 4.5:

To set up .NET Connector for usage with .NET Framework 4.5, reference the shared library files located in the following directory:

```
C:\Program Files\edb\connectors\dotnet\net45\
```

The shared library files are the following:

```
EnterpriseDB.EDBClient.dll
System.Threading.Tasks.Extensions.dll
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Note: It is also required to refer to the dependency on `System.Threading.Tasks.Extensions.dll` in the project.

See Section 3.1 for information on referencing library files.

Depending upon the type of application, import the namespace into the source code (see Section 3.1).

3.2.3 .NET Framework 4.5.1

If you are using .NET Framework version 4.5.1:

To set up .NET Connector for usage with .NET Framework 4.5.1, reference the shared library files located in the following directory:

```
C:\Program Files\edb\connectors\dotnet\net451\
```

The shared library files are the following:

```
EnterpriseDB.EDBClient.dll
System.Threading.Tasks.Extensions.dll
```

Note: It is also required to refer to the dependency on `System.Threading.Tasks.Extensions.dll` in the project.

See Section 3.1 for information on referencing library files.

Depending upon the type of application, import the namespace into the source code (see Section 3.1).

3.2.4 .NET Standard 1.3

For .NET Standard Framework 1.3, the data provider installation path is the following:

```
C:\Program Files\edb\connectors\dotnet\netstandard1.3\
```

The shared library file is the following:

```
EnterpriseDB.EDBClient.dll
```

See Section 3.1 for information on referencing library files.

Depending upon the type of application, import the namespace into the source code (see Section 3.1).

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

3.2.5 .NET Standard 2.0

For .NET Standard Framework 2.0, the data provider installation path is the following:

```
C:\Program Files\edb\connectors\dotnet\netstandard2.0\
```

The shared library file is the following:

```
EnterpriseDB.EDBClient.dll
```

See Section 3.1 for information on referencing library files.

Depending upon the type of application, import the namespace into the source code (see Section 3.1).

3.3 Entity Framework 5/6

To set up .NET Connector for usage with Entity Framework, reference the shared library files located in the following directory:

```
C:\Program Files\edb\connectors\dotnet\EF\
```

The shared library files are the following:

```
EntityFramework5.EnterpriseDB.EDBClient.dll
EntityFramework6.EnterpriseDB.EDBClient.dll
```

Note: Entity Framework can be used with `EnterpriseDB.EDBClient.dll` available in the `net45` and `net451` subdirectories. For reference, see sections 3.2.2 and 3.2.3.

See Section 3.1 for information on referencing library files.

Add the `<DbProviderFactories>` entries for the ADO.NET driver of EDB Postgres in the `app.config` file. The following are the entries:

```
<add name="EnterpriseDB.EDBClient"
  invariant="EnterpriseDB.EDBClient"
  description=".NET Data Provider for EnterpriseDB
PostgreSQL”
  type="EnterpriseDB.EDBClient.EDBFactory,
EnterpriseDB.EDBClient, Version=3.2.6.1, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7"
  support="FF"/>
```

In the project’s `app.config` file add the following entry for provider services under the EntityFramework/providers tag.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
<provider invariantName="EnterpriseDB.EDBClient"
  type="EnterpriseDB.EDBClient.EDBServices,
EntityFramework6.EnterpriseDB.EDBClient">
</provider>
```

The following is an example of the `app.config` file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false"/>
  </configSections>

    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>

  <entityFramework>
    <providers>
      <provider invariantName="EnterpriseDB.EDBClient"
type="EnterpriseDB.EDBClient.EDBServices,
EntityFramework6.EnterpriseDB.EDBClient"></provider>
    </providers>
  </entityFramework>

  <system.data>
    <DbProviderFactories>
      <remove invariant="EnterpriseDB.EDBClient"/>
      <add name="EnterpriseDB Data Provider" invariant="EnterpriseDB.EDBClient"
support="FF" description=".Net Framework Data Provider for Postgresql"
type="EnterpriseDB.EDBClient.EDBFactory, EnterpriseDB.EDBClient"/>
    </DbProviderFactories>
  </system.data>

</configuration>
```

Note: The same entries for `<providers>` and `<DbProviderFactories>` are valid for the `web.config` file as well as for `app.config`.

Depending upon the type of application, import the namespace into the source code (see Section 3.1).

For usage of the features of Entity Framework, refer to the Microsoft documentation.

3.4 EnterpriseDB VSIX for Visual Studio 2015/2017

EDB Data Designer Extensibility Provider (*EnterpriseDB VSIX*) is a component, which integrates Advanced Server database access into Visual Studio, thus providing Visual Studio integrated features.

It allows connecting to Advanced Server from within Visual Studio's Server Explorer, creating a model from an existing database, etc. Therefore, if Visual Studio features are desired, then EnterpriseDB VSIX must be utilized.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

EnterpriseDB VSIX is located in the following directory:

```
C:\Program Files\edb\connectors\dotnet\vsix\
```

The files available at the above location are the following:

```
EnterpriseDB.vsix
SSDLToPgSQL.tt
```

3.4.1 Installation and Configuration for Visual Studio 2015/2017

The following are the steps to install and configure EnterpriseDB VSIX.

Step 1: Install EnterpriseDB VSIX to the desired version of Visual Studio with the `EnterpriseDB.vsix` installer at the following location:

```
C:\Program Files\edb\connectors\dotnet\vsix\
EnterpriseDB.vsix
```

If you already have an earlier version of the VSIX installed, it's highly recommended that you uninstall them to avoid conflicts.

It is no longer necessary or recommended to have `EnterpriseDB.EDBClient` in your global assembly cache (GAC).

Step 2: Relaunch Visual Studio and verify from `Tools > Extensions and Updates…` menu that the EnterpriseDB extension is installed.

Step 3: From Server Explorer, right-click on `Data Connections`, click `Add Connection`, and verify the `Enterprisedb Postgres Database` data source is available.

3.4.2 Model First and Database First Usage

Step 1: Add the `EntityFramework5.EnterpriseDB.EDBClient.dll` assembly in the global assembly cache (GAC) with the `gacutil` utility using the Visual Studio Developers Command line.

For example:

```
> gacutil.exe /i EntityFramework5.EnterpriseDB.EDBClient.dll
```

Step 2: Add the `<DbProviderFactories>` entries for the ADO.NET driver of EDB Postgres in the `machine.config` file. The following are the entries:

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
<add name="EnterpriseDB.EDBClient"
  invariant="EnterpriseDB.EDBClient"
  description=".NET Data Provider for EnterpriseDB
PostgreSQL"
  type="EnterpriseDB.EDBClient.EDBFactory,
EnterpriseDB.EDBClient, Version=3.2.6.1, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7"
  support="FF"/>
```

Note: For the attribute-value pairs, the double-quoted strings should not contain excess white space characters, but be configured on a single line. The examples shown in this section may be split on multiple lines for clarity, but should actually be configured within a single line such as the following:

```
description=".NET Data Provider for EnterpriseDB PostgreSQL"
```

For 64-bit Windows, the `machine.config` file is in the following location:

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\
Config\machine.config
```

For 32-bit Windows, the `machine.config` file is in the following location:

```
C:\Windows\Microsoft.NET\Framework\v4.0.30319\
Config\machine.config
```

Step 3: Place the DDL generation template `SSDLToPgSQL.tt` in the Visual Studio `EntityFramework Tools\DBGen\` folder as in the following example:

```
C:\Program Files (x86)\Microsoft Visual Studio 14.0\
Common7\IDE\Extensions\Microsoft\EntityFramework Tools\DBGen\
```

Note: Select this template `SSDLToPgSQL.tt` in your EDMX file properties.

Step 4: Add files `EnterpriseDB.EDBClient.dll` and `EntityFramework6.EnterpriseDB.EDBClient.dll` in project references. See Section 3.1 for information on referencing library files.

Step 5: In the project's `app.config` file add the following entry for provider services under the EntityFramework/providers tag.

```
<provider invariantName="EnterpriseDB.EDBClient"
  type="EnterpriseDB.EDBClient.EDBServices,
EntityFramework6.EnterpriseDB.EDBClient">
</provider>
```

The following is an example of the `app.config` file.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework,
Version=6.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089"
requirePermission="false"/>
  </configSections>

    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
    </startup>

  <entityFramework>
    <providers>
      <provider invariantName="EnterpriseDB.EDBClient"
type="EnterpriseDB.EDBClient.EDBServices,
EntityFramework6.EnterpriseDB.EDBClient"></provider>
    </providers>
  </entityFramework>

  <system.data>
    <DbProviderFactories>
      <remove invariant="EnterpriseDB.EDBClient"/>
      <add name="EnterpriseDB Data Provider" invariant="EnterpriseDB.EDBClient"
support="FF" description=".Net Framework Data Provider for EDB Postgres"
type="EnterpriseDB.EDBClient.EDBFactory, EnterpriseDB.EDBClient"/>
    </DbProviderFactories>
  </system.data>

</configuration>
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4 Using the .NET Connector

The sections that follow provide examples that demonstrate using the EDB object classes that are provided by the Advanced Server .NET Connector that allow a .NET application to connect to and interact with an Advanced Server database.

To use the examples in this guide, place the .NET library files in the same directory as the compiled form of your application. All of the examples are written in C# and each is embedded in an ASP.NET page; the same logic and code would be applicable with other .NET applications (WinForm or console applications, for example).

Please create and save the following `web.config` file in the same directory as the sample code. The examples make use of the `DB_CONN_STRING` key from this configuration file to return a connection string from the Advanced Server host.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
<appSettings>
    <add key="DB_CONN_STRING" value="Server=127.0.0.1;Port=5444;
    User Id=enterprisedb;Password=enterprisedb;Database=edb" />
</appSettings>
</configuration>
```

An Advanced Server connection string for an ASP.NET web application is stored in the `web.config` file. If you are writing an application that does not use ASP.NET, provide the connection information in an application configuration file (such as `app.config`).

4.1 Opening a Database Connection

An `EDBConnection` object is responsible for handling the communication between an instance of Advanced Server and a .NET application. Before you can access data stored in an Advanced Server database, you must create and open an `EDBConnection`.

The examples that follow demonstrate the basic steps for connecting to an instance of Advanced Server. You must:

1. Import the namespace `EnterpriseDB.EDBClient`.

2. Create an instance of `EDBConnection`.

3. Initialize the `EDBConnection` object by passing a connection string as a parameter to the constructor for the `EDBConnection` class.

4. Call the `Open` method of the `EDBConnection` object to open the connection.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.1.1 Connection String Parameters

A valid connection string should specify location and authentication information for an Advanced Server instance. You must provide the connection string before opening the connection. A connection string must contain:

- The name or IP address of the server
- The name of the Advanced Server database
- The name of an Advanced Server user
- The password associated with that user

The following parameters may be included in the connection string:

`CommandTimeout`

> `CommandTimeout` specifies the length of time (in seconds) to wait for a command to finish execution before throwing an exception. The default value is `20`.

`ConnectionLifeTime`

> Use `ConnectionLifeTime` to specify the length of time (in seconds) to wait before closing unused connections in the pool. The default value is `15`.

`Database`

> Use the `Database` parameter to specify the name of the database to which the application should connect. If a database name is not specified, the database name will default to the name of the connecting user.

`Encoding`

> The `Encoding` parameter is obsolete; the parameter always returns the string `unicode`, and silently ignores attempts to set it.

`Integrated Security`

> By default, `Integrated Security` is set to `false`, and Windows Integrated Security is disabled. Specify a value of `true` to use Windows Integrated Security.

`MaxPoolSize`

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

MaxPoolSize instructs EDBConnection to dispose of pooled connections when the pool exceeds the specified number of connections. The default value is 20.

MinPoolSize

MinPoolSize instructs EDBConnection to pre-allocate the specified number of connections with the server. The default value is 1.

Password

When using clear text authentication, specify the password that will be used to establish a connection with the server.

Pooling

By default, Pooling is set to true to enable connection pooling. Specify a value of false to disable connection pooling.

Port

The Port parameter specifies the port to which the application should connect.

Protocol

The specific protocol version to use (instead of automatic); specify an integer value of 2 or 3.

SearchPath

Use the SearchPath parameter to change the search path to named and public schemas.

Server

The name or IP address of the Advanced Server host.

SSL

By default, SSL is set to false; specify a value of true to attempt a secure connection.

sslmode

Use sslmode to specify an SSL connection control preference. sslmode can be:

prefer - Use SSL if possible.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

`require` - Throw an exception if an SSL connection cannot be established.
`allow` - Connect without SSL. This parameter is not supported.
`disable` - Do not attempt an SSL connection. This is the default behavior.

`SyncNotification`

Use the `SyncNotification` parameter to specify that `EDBDataprovider` should use synchronous notifications. The default value is `false`.

`Timeout`

`Timeout` specifies the length of time (in seconds) to wait for an open connection. The default value is `15`.

`User Id`

The `User Id` parameter specifies the user name that should be used for the connection.

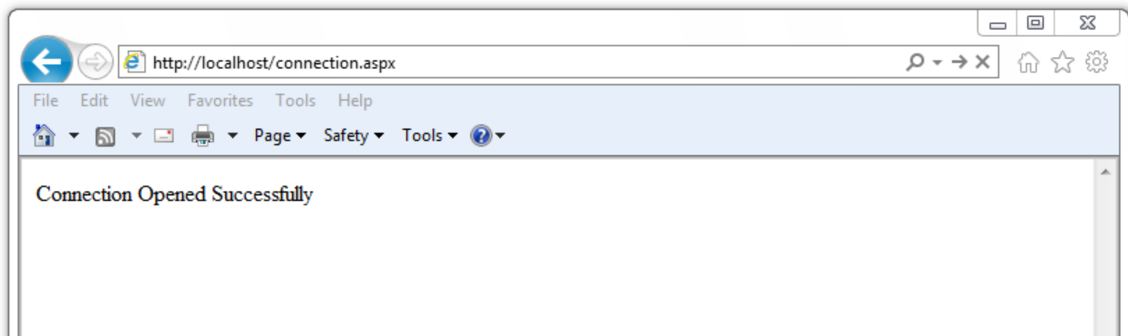
Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.1.2 Example - Opening a Database Connection using ASP.NET

The following example demonstrates how to open a connection to an instance of Advanced Server and then close the connection. The connection is established using the credentials specified in the DB_CONN_STRING configuration parameter (see the beginning of Chapter 4 for an introduction to connection information and Section 4.1.1 for connection parameters).

```
<% @ Page Language="C#" %>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server">

private void Page_Load(object sender, System.EventArgs e)
{
  string strConnectionString = ConfigurationSettings.AppSettings
    ["DB_CONN_STRING"];

  EDBConnection conn = new EDBConnection(strConnectionString);

  try
  {
    conn.Open();
    Response.Write("Connection opened successfully");
  }

  catch(EDBException exp)
  {
    exp.ToString();
  }

  finally
  {
    conn.Close();
  }
}

</script>
```

If the connection is successful, a browser will display the following:



Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.1.3 Example - Opening a Database Connection from a Console Application

The following example opens a connection with an Advanced Server database using a console-based application.

Before writing the code for the console application, create an `app.config` file that stores the connection string to the database. Using a configuration file makes it convenient to update the connection string if the information changes.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <appSettings>
          <add key="DB_CONN_STRING" value = "Server=127.0.0.1;Port=5444;
          User Id=enterprisedb;Password=enterprisedb;Database=edb"/>
    </appSettings>
</configuration>
```

Using your text editor of choice, enter the following code sample into a file:

```
using System;
using System.Data;
using EnterpriseDB.EDBClient;
using System.Configuration;

namespace EnterpriseDB
{

    class EDB
    {

      static void Main(string[] args)
      {
         string strConnectionString = ConfigurationSettings.AppSettings
           ["DB_CONN_STRING"];

         EDBConnection conn = new EDBConnection(strConnectionString);

          try
          {
             conn.Open();
             Console.WriteLine("Connection Opened Successfully");
          }

          catch(Exception exp)
          {
             throw new Exception(exp.ToString());
          }

          finally
          {
           conn.Close();
          }
       }
    }
}
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Save the file as `EDBConnection-Sample.cs` and compile it with the following command:

```
csc /r:EDBDataProvider.dll /out:Console.exe EDBConnection-Sample.cs
```

Compiling the sample should generate a `Console.exe` file; you can execute the sample code by entering `Console.exe`. When executed, the console should verify that the:

```
Connection Opened Successfully
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.1.4 Example - Opening a Database Connection from a Windows Form Application

The following example demonstrates opening a database connection using a .NET WinForm application. To use the example, save the following code as `WinForm-Example.cs` in a directory that contains the library files.

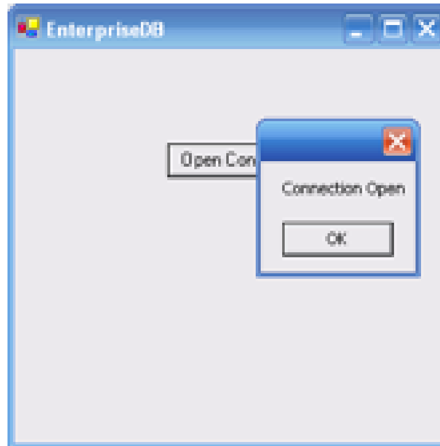
```
using System;
using System.Windows.Forms;
using System.Drawing;
using EnterpriseDB.EDBClient;

namespace EDBTestClient
{

     class Win_Conn
     {
            static void Main(string[] args)
            {
                   Form frmMain = new Form();
                   Button btnConn = new Button();
                   btnConn.Location = new System.Drawing.Point(104, 64);
                   btnConn.Name = "btnConn";
                   btnConn.Text = "Open Connection";
                   btnConn.Click += new System.EventHandler(btnConn_Click);

                   frmMain.Controls.Add(btnConn);
                   frmMain.Text = "EnterpriseDB";

                   Application.Run(frmMain);
            }

            private static void btnConn_Click(object sender, System.EventArgs
e)
            {
                   EDBConnection conn = null;
                   try
                   {
                          string connectionString =
"Server=10.90.1.29;port=5444;
                          username=edb;password=edb;database=edb";
                          conn = new EDBConnection(connectionString);
                          conn.Open();
                          MessageBox.Show("Connection Open");

                   }
                   catch(EDBException exp)
                   {
                          MessageBox.Show(exp.ToString());
                   }
                   finally
                   {
                          conn.Close();
                   }
            }
     }
}
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Note that you must change the database connection string to point to the database that you want to connect to before compiling the file with the following command:

```
csc /r:EDBDataProvider.dll /out:WinForm.exe WinForm-Example.cs
```

This command should generate a `WinForm.exe` file within the same folder that the executable was compiled under. Invoking the executable will display:



Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.2 Retrieving Database Records

You can use a SELECT statement to retrieve records from the database via a SELECT command. To execute a SELECT statement you must:

- Create and open a database connection.
- Create an EDBCommand object that represents the SELECT statement.
- Execute the command with the ExecuteReader() method of the EDBCommand object returning a EDBDataReader
- Loop through the EDBDataReader displaying the results or binding the EDBDataReader to some control.

An EDBDataReader object represents a forward-only and read-only stream of database records, presented one record at a time. To view a subsequent record in the stream, you must call the Read() method of the EDBDataReader object.

The example that follows:

1. Imports the Advanced Server namespace: EnterpriseDB.EDBClient
2. Initializes an EDBCommand object with a SELECT statement.
3. Opens a connection to the database.
4. Executes the EDBCommand by calling the ExecuteReader method of the EDBCommand object.

   The results of the SQL statement are retrieved into an EDBDataReader object.
5. Loops through the contents of the EDBDataReader object to display the records returned by the query within a WHILE loop.

The Read() method advances to the next record (if a record exists) and returns true if a record exists, or false to indicate that the EDBDataReader has reached the end of the result set.

```
<% @ Page Language="C#" %>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server">
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
private void Page_Load(object sender, System.EventArgs e)
{
     string strConnectionString = ConfigurationSettings.AppSettings
     ["DB_CONN_STRING"];
     EDBConnection conn = new EDBConnection(strConnectionString);

     try
     {
            conn.Open();
            EDBCommand cmdSelect = new EDBCommand("SELECT * FROM dept",conn);
            cmdSelect.CommandType = CommandType.Text;

            EDBDataReader drDept = cmdSelect.ExecuteReader();

            while(drDept.Read())
            {
                   Response.Write("Department Number: " + drDept["deptno"]);
                   Response.Write("\tDepartment Name: " + drDept["dname"]);
                   Response.Write("\tDepartment Location: " + drDept["loc"]);
                   Response.Write("<br>");
            }

     }

     catch(Exception exp)
     {
            Response.Write(exp.ToString());
     }
     finally
     {
            conn.Close();
     }
}
</script>
```

To exercise the sample code, save the code in your default web root directory in a file named:

```
selectEmployees.aspx
```

To invoke the program, open a web-browser, and browse to:

```
http://localhost/selectEmployees.aspx.
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.2.1 Retrieving a Single Database Record

To retrieve a single result from a query, use the `ExecuteScalar()` method of the `EDBCommand` object. The `ExecuteScalar()` method returns the first value of the first column of the first row of the `DataSet` generated by the specified query.

```
<% @ Page Language="C#" %>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server">

private void Page_Load(object sender, System.EventArgs e)
{
     string strConnectionString = ConfigurationSettings.AppSettings
     ["DB_CONN_STRING"];
     EDBConnection conn = new EDBConnection(strConnectionString);
     try
     {
        conn.Open();
        EDBCommand cmd = new EDBCommand("SELECT MAX(sal) FROM emp",conn);

        cmd.CommandType = CommandType.Text;

        int maxSal = Convert.ToInt32(cmd.ExecuteScalar());

        Response.Write("Emp Number: " + maxSal);

     }
     catch(Exception exp)
     {
        Response.Write(exp.ToString());
     }
     finally
     {
        conn.Close();
     }
}
</script>
```

Save the sample code in a file in a web root directory named:

```
selectscalar.aspx
```

To invoke the sample code, open a web-browser, and browse to:

```
http://localhost/selectScalar.aspx
```

Please note that the sample includes an explicit conversion of the value returned by the `ExecuteScalar()` method. The `ExecuteScalar()` method returns an object; to view the object, you must convert it into an integer value by using the `Convert.ToInt32` method.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.3 Parameterized Queries

A *parameterized query* is a query with one or more parameter markers embedded in the SQL statement. Before executing a parameterized query, you must supply a value for each marker found in the text of the SQL statement.

Parameterized queries are useful when you don't know the complete text of a query at the time you write your code. For example, the value referenced in a `WHERE` clause may be calculated from user input.

As demonstrated in the following example, you must declare the data type of each parameter specified in the parameterized query by creating an `EDBParameter` object and adding that object to the command's parameter collection. Then, you must specify a *value* for each parameter by calling the parameter's `Value()` function.

The example demonstrates use of a parameterized query with an `UPDATE` statement that increases an employee salary:

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
    string strConnectionString = ConfigurationSettings.AppSettings
    ["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);

    string updateQuery  = "UPDATE emp SET sal = sal+500 where empno = :ID";

    try {
        conn.Open();

        EDBCommand cmdUpdate = new EDBCommand(updateQuery,conn);

        cmdUpdate.Parameters.Add
        (new EDBParameter(":ID", EDBTypes.EDBDbType.Integer));

        cmdUpdate.Parameters[0].Value = 7788;

        cmdUpdate.ExecuteNonQuery();

        Response.Write("Record Updated");

    }
    catch(Exception exp) {
        Response.Write(exp.ToString());
    }
    finally {
        conn.Close();
    }
}
</script>
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Save the sample code in a file in a web root directory named:

```
updateSalary.aspx
```

To invoke the sample code, open a web-browser, and browse to:

```
http://localhost/updateSalary.aspx
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.4 Inserting Records in a Database

You can use the `ExecuteNonQuery()` method of `EDBCommand` to add records to a database stored on an Advanced Server host with an `INSERT` command.

In the example that follows, the `INSERT` command is stored in the variable `cmd`. The values prefixed with a colon (`:`) are placeholders for `EDBParameters` that are instantiated, assigned values, and then added to the `INSERT` command's parameter collection in the statements that follow. The `INSERT` command is executed by the `ExecuteNonQuery()` method of the `cmdInsert` object.

The example adds a new employee to the `emp` table:

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
     string strConnectionString = ConfigurationSettings.AppSettings
     ["DB_CONN_STRING"];
     EDBConnection conn = new EDBConnection(strConnectionString);

     try
     {
       conn.Open();

       string cmd = "INSERT INTO emp(empno,ename) VALUES(:EmpNo, :EName)";

       EDBCommand cmdInsert = new EDBCommand(cmd,conn);

       cmdInsert.Parameters.Add(new EDBParameter(":EmpNo",
         EDBTypes.EDBDbType.Integer));

       cmdInsert.Parameters[0].Value = 1234;

       cmdInsert.Parameters.Add(new EDBParameter(":EName",
         EDBTypes.EDBDbType.Text));

       cmdInsert.Parameters[1].Value = "Lola";

       cmdInsert.ExecuteNonQuery();
       Response.Write("Record inserted successfully");
     }

     catch(Exception exp)
     {
            Response.Write(exp.ToString());
     }
     finally
     {
            conn.Close();
     }
}
</script>
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Save the sample code in a file in a web root directory named:

```
insertEmployee.aspx
```

To invoke the sample code, open a web-browser, and browse to:

```
http://localhost/insertEmployee.aspx
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.5 Deleting Records in a Database

You can use the `ExecuteNonQuery()` method of `EDBCommand` to delete records from a database stored on an Advanced Server host with a `DELETE` statement.

In the example that follows, the `DELETE` command is stored in the variable `strDeleteQuery`. The code passes the employee number to the Delete command (specified by `:EmpNo`). The command is then executed using the `ExecuteNonQuery()` method. The following example deletes the employee inserted in the previous example:

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
     string strConnectionString = ConfigurationSettings.AppSettings
     ["DB_CONN_STRING"];

     EDBConnection conn = new EDBConnection(strConnectionString);

     string strDeleteQuery  = "DELETE FROM emp WHERE empno = :ID";

     try
     {
            conn.Open();

            EDBCommand deleteCommand = new EDBCommand(strDeleteQuery,conn);

            deleteCommand.Parameters.Add
            (new EDBParameter(":ID", EDBTypes.EDBDbType.Integer));

            deleteCommand.Parameters[0].Value = 1234;

            deleteCommand.ExecuteNonQuery();

            Response.Write("Record Deleted");

     }

     catch(Exception exp)
     {

            Response.Write(exp.ToString());

     }

     finally
     {

            conn.Close();

     }
}
</script>
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

Save the sample code in a file in a web root directory named:

```
deleteEmployee.aspx
```

To invoke the sample code, open a web-browser, and browse to:

```
http://localhost/deleteEmployee.aspx
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.6 Using SPL Stored Procedures in your .NET Application

You can include SQL statements in an application in two ways:

- By adding the SQL statements directly in the .NET application code.
- By packaging the SQL statements in a stored procedure, and executing the stored procedure from the .NET application.

In some cases, a stored procedure can provide advantages over embedded SQL statements. Stored procedures support complex conditional and looping constructs that are difficult to duplicate with SQL statements embedded directly in an application.

You can also see a significant improvement in performance by using stored procedures; a stored procedure only needs to be parsed, compiled and optimized once on the server side, while a SQL statement that is included in an application may be parsed, compiled and optimized each time it is executed from a .NET application.

To use a stored procedure in your .NET application you must:

1. Create an SPL stored procedure on the Advanced Server host.
2. Import the `EnterpriseDB.EDBClient` namespace.
3. Pass the name of the stored procedure to the instance of the `EDBCommand`.
4. Change the `EDBCommand.CommandType` to `CommandType.StoredProcedure`.
5. `Prepare()` the command.
6. Execute the command.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.6.1 Example - Executing a Stored Procedure without Parameters

Our sample procedure prints the name of department 10; the procedure takes no parameters, and returns no parameters. To create the sample procedure, invoke EDB-PSQL and connect to the Advanced Server host database. Enter the following SPL code at the command line:

```
CREATE OR REPLACE PROCEDURE list_dept10
IS
  v_deptname VARCHAR2(30);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Dept No: 10');
  SELECT dname INTO v_deptname FROM dept WHERE deptno = 10;
  DBMS_OUTPUT.PUT_LINE('Dept Name: ' || v_deptname);
END;
```

When Advanced Server has validated the stored procedure it will echo CREATE PROCEDURE.

Using the EDBCommand Object to Execute a Stored Procedure

The CommandType property of the EDBCommand object is used to indicate the type of command being executed. The CommandType property is set to one of three possible CommandType enumeration values:

- Use the default Text value when passing a SQL string for execution.
- Use the StoredProcedure value, passing the name of a stored procedure for execution.
- Use the TableDirect value when passing a table name. This value passes back all records in the specified table.

The CommandText property must contain a SQL string, stored procedure name, or table name depending on the value of the CommandType property.

The following example executes the stored procedure:

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
    string strConnectionString = ConfigurationSettings.AppSettings
    ["DB_CONN_STRING"];
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
    EDBConnection conn = new EDBConnection(strConnectionString);


    try
    {
           conn.Open();

           EDBCommand cmdStoredProc = new EDBCommand("list_dept10",conn);
           cmdStoredProc.CommandType = CommandType.StoredProcedure;

           cmdStoredProc.Prepare();
           cmdStoredProc.ExecuteNonQuery();


           Response.Write("Stored Procedure Executed Successfully");

    }
    catch(Exception exp)
    {
           Response.Write(exp.ToString());
    }
    finally
    {
           conn.Close();
    }
}
</script>
```

Save the sample code in a file in a web root directory named:

```
storedProc.aspx
```

To invoke the sample code, open a web-browser, and browse to:

```
http://localhost/storedProc.aspx
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.6.2 Example - Executing a Stored Procedure with IN Parameters

The following example demonstrates calling a stored procedure that includes `IN` parameters. To create the sample procedure, invoke EDB-PSQL and connect to the Advanced Server host database. Enter the following SPL code at the command line:

```
CREATE OR REPLACE PROCEDURE
  EMP_INSERT
  (
     pENAME IN VARCHAR,
     pJOB IN VARCHAR,
     pSAL IN FLOAT4,
     pCOMM IN FLOAT4,
     pDEPTNO IN INTEGER,
     pMgr IN INTEGER
   )
AS
DECLARE
  CURSOR TESTCUR IS SELECT MAX(EMPNO) FROM EMP;
  MAX_EMPNO INTEGER := 10;
BEGIN

  OPEN TESTCUR;
  FETCH TESTCUR INTO MAX_EMPNO;
  INSERT INTO EMP(EMPNO,ENAME,JOB,SAL,COMM,DEPTNO,MGR)
    VALUES(MAX_EMPNO+1,pENAME,pJOB,pSAL,pCOMM,pDEPTNO,pMgr);
  CLOSE testcur;
END;
```

When Advanced Server has validated the stored procedure it will echo `CREATE PROCEDURE`.

Passing Input values to a Stored Procedure

Calling a stored procedure that contains parameters is very similar to executing a stored procedure without parameters. The major difference is that when calling a parameterized stored procedure you must use the `EDBParameter` collection of the `EDBCommand` object. When the `EDBParameter` is added to the `EDBCommand` collection, properties such as `ParameterName`, `DbType`, `Direction`, `Size`, and `Value` are set.

The following example demonstrates the process of executing a parameterized stored procedure from a C#.

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
     string strConnectionString = ConfigurationSettings.AppSettings
     ["DB_CONN_STRING"];
     EDBConnection conn = new EDBConnection(strConnectionString);

     string empName    = "EDB";
     string empJob     = "Manager";
     double salary     = 1000;
     double commission = 0.0;
     int    deptno     = 20;
     int    manager    = 7839;

     try
     {
            conn.Open();

            EDBCommand cmdStoredProc = new EDBCommand
            ("emp_insert(:EmpName,:Job,:Salary,:Commission,:DeptNo,
            :Manager)",conn);
            cmdStoredProc.CommandType = CommandType.StoredProcedure;

            cmdStoredProc.Parameters.Add(new EDBParameter
            ("EmpName", EDBTypes.EDBDbType.Varchar));
            cmdStoredProc.Parameters[0].Value = empName;

            cmdStoredProc.Parameters.Add(new EDBParameter
            ("Job", EDBTypes.EDBDbType.Varchar));
            cmdStoredProc.Parameters[1].Value = empJob;

            cmdStoredProc.Parameters.Add(new EDBParameter
            ("Salary", EDBTypes.EDBDbType.Float));
            cmdStoredProc.Parameters[2].Value = salary;

            cmdStoredProc.Parameters.Add(new EDBParameter
            ("Commission", EDBTypes.EDBDbType.Float));
            cmdStoredProc.Parameters[3].Value = commission;

            cmdStoredProc.Parameters.Add(new EDBParameter
            ("DeptNo", EDBTypes.EDBDbType.Integer));
            cmdStoredProc.Parameters[4].Value = deptno;

            cmdStoredProc.Parameters.Add
            (new EDBParameter("Manager", EDBTypes.EDBDbType.Integer));
            cmdStoredProc.Parameters[5].Value = manager;

            cmdStoredProc.Prepare();

            cmdStoredProc.ExecuteNonQuery();

            Response.Write("Following Information Inserted Successfully
            <br>");
            string  empInfo  = "Employee Name: " + empName + "<br>";
            empInfo += "Job: " + empJob + "<br>";
            empInfo += "Salary: " + salary + "<br>";
            empInfo += "Commission: " + commission + "<br>";
            empInfo += "Manager: " + manager + "<br>";

            Response.Write(empInfo);

     }
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
    catch(Exception exp)
    {
           Response.Write(exp.ToString());
    }
    finally
    {
           conn.Close();
    }
}

</script>

</script>
```

Save the sample code in a file in a web root directory named:

```
storedProcInParam.aspx
```

To invoke the sample code, open a web-browser, and browse to:

```
http://localhost/storedProcInParam.aspx
```

In the example, the body of the `Page_Load` method declares and instantiates an `EDBConnection` object. The sample then creates an `EDBCommand` object with the properties needed to execute the stored procedure.

The example then uses the `Add` method of the `EDBCommand Parameter` collection to add six input parameters.

```
EDBCommand cmdStoredProc = new EDBCommand
("emp_insert(:EmpName,:Job,:Salary,:Commission,:DeptNo,:Man
ager)",conn);
cmdStoredProc.CommandType = CommandType.StoredProcedure;
```

It assigns a value to each parameter before passing them to the `EMP_INSERT` stored procedure

The `Prepare()` method prepares the statement before calling the `ExecuteNonQuery()` method.

The `ExecuteNonQuery` method of the `EDBCommand` object executes the stored procedure. After the stored procedure has executed, a test record is inserted into the `emp` table and the values inserted are displayed on the webpage.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.6.3 Example - Executing a Stored Procedure with IN, OUT, and INOUT Parameters

The previous example demonstrated how to pass `IN` parameters to a stored procedure; the following examples demonstrate how to pass `IN` values and return `OUT` values from a stored procedure.

Creating the Stored Procedure

The following stored procedure passes the department number, and returns the corresponding location and department name. To create the sample procedure, open the EDB-PSQL command line, and connect to the Advanced Server host database. Enter the following SPL code at the command line:

```
CREATE OR REPLACE PROCEDURE
  DEPT_SELECT
  (
    pDEPTNO IN  INTEGER,
    pDNAME  OUT VARCHAR,
    pLOC    OUT VARCHAR
  )
AS
DECLARE
  CURSOR TESTCUR IS SELECT DNAME,LOC FROM DEPT;
  REC RECORD;
BEGIN

  OPEN TESTCUR;
  FETCH TESTCUR INTO REC;

  pDNAME  := REC.DNAME;
  pLOC    := REC.LOC;

  CLOSE testcur;
END;
```

When Advanced Server has validated the stored procedure it will echo `CREATE PROCEDURE`.

Receiving Output values from a Stored Procedure

When retrieving values from `OUT` parameters you must explicitly specify the direction of out parameters as `Output`. You can retrieve the values from `Output` parameters in two ways:

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

- Call the `ExecuteReader` method of the `EDBCommand` and explicitly loop through the returned `EDBDataReader`, searching for the values of `OUT` parameters.

- Call the `ExecuteNonQuery` method of `EDBCommand` and explicitly get the value of a declared `Output` parameter by calling that `EDBParameter` value property.

In each method, you must declare each parameter, indicating the *direction* of the parameter (`ParameterDirection.Input`, `ParameterDirection.Output` or `ParameterDirection.InputOutput`). Before invoking the procedure, you must provide a value for each `IN` and `INOUT` parameter. After the procedure returns, you may retrieve the `OUT` and `INOUT` parameter values from the `command.Parameters[]` array.

The following code listing demonstrates using the `ExecuteReader` method to retrieve a result set:

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
     string strConnectionString =
        ConfigurationSettings.AppSettings["DB_CONN_STRING"];
     EDBConnection conn = new EDBConnection(strConnectionString);

     try
     {

       conn.Open();
       EDBCommand command = new EDBCommand("DEPT_SELECT
                  (:pDEPTNO,:pDNAME,:pLOC)", conn);
       command.CommandType = CommandType.StoredProcedure;

       command.Parameters.Add(new EDBParameter("pDEPTNO",
                  EDBTypes.EDBDbType.Integer,10,"pDEPTNO",
                  ParameterDirection.Input,false ,2,2,
                  System.Data.DataRowVersion.Current,1));

       command.Parameters.Add(new EDBParameter("pDNAME",
                  EDBTypes.EDBDbType.Varchar,10,"pDNAME",
                  ParameterDirection.Output,false ,2,2,
                  System.Data.DataRowVersion.Current,1));

       command.Parameters.Add(new EDBParameter("pLOC",
                  EDBTypes.EDBDbType.Varchar,10,"pLOC",
                  ParameterDirection.Output,false ,2,2,
                  System.Data.DataRowVersion.Current,1));

       command.Prepare();

       command.Parameters[0].Value = 10;
       EDBDataReader result = command.ExecuteReader();
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
      int fc = result.FieldCount;

      while(result.Read())
      {
           for(int i = 0;i < fc; i++)
           {
             Response.Write("RESULT["+i+"]="+ Convert.ToString
                  (command.Parameters[i].Value));
                  Response.Write("<br>");
           }
      }

    }

    catch(EDBException exp)
    {
      Response.Write(exp.ToString());
    }
    finally
    {
      conn.Close();
    }
}

</script>
```

The following code listing demonstrates using the `ExecuteNonQuery` method to retrieve a result set:

```
<% @ Page Language="C#" Debug="true"%>
<% @Import Namespace="EnterpriseDB.EDBClient" %>
<% @Import Namespace="System.Data" %>
<% @Import Namespace="System.Configuration" %>

<script language="C#" runat="server" >

private void Page_Load(object sender, System.EventArgs e)
{
    string strConnectionString = ConfigurationSettings.AppSettings
    ["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);

    try
    {

      conn.Open();
      EDBCommand command = new EDBCommand("DEPT_SELECT
           (:pDEPTNO,:pDNAME,:pLOC)", conn);
      command.CommandType = CommandType.StoredProcedure;

      command.Parameters.Add(new EDBParameter("pDEPTNO",
           EDBTypes.EDBDbType.Integer,10,"pDEPTNO",
           ParameterDirection.Input,false ,2,2,
           System.Data.DataRowVersion.Current,1));

      command.Parameters.Add(new EDBParameter("pDNAME",
           EDBTypes.EDBDbType.Varchar,10,"pDNAME",
           ParameterDirection.Output,false ,2,2,
           System.Data.DataRowVersion.Current,1));

      command.Parameters.Add(new EDBParameter("pLOC",
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

```
            EDBTypes.EDBDbType.Varchar,10,"pLOC",
            ParameterDirection.Output,false ,2,2,
            System.Data.DataRowVersion.Current,1));

       command.Prepare();
       command.Parameters[0].Value = 10;
       command.ExecuteNonQuery();

       Response.Write(command.Parameters["pDNAME"].Value.ToString());
       Response.Write(command.Parameters["pLOC"].Value.ToString());
     }

     catch(EDBException exp)
     {
       Response.Write(exp.ToString());
     }
     finally
     {
       conn.Close();
     }
}
</script>
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

4.7 Using a Ref Cursor in a .NET Application

A *ref cursor* is a cursor variable that contains a pointer to a query result set. The result set is determined by the execution of the `OPEN FOR` statement using the cursor variable. A cursor variable is not tied to a particular query like a static cursor. The same cursor variable may be opened a number of times with the `OPEN FOR` statement containing different queries and each time, a new result set will be created for that query and made available via the cursor variable. There are two ways to declare a cursor variable:

- Use the `SYS_REFCURSOR` built-in data type to declare a weakly-typed ref cursor.
- Define a strongly-typed ref cursor that declares a variable of that type.

`SYS_REFCURSOR` is a ref cursor type that allows any result set to be associated with it. This is known as a weakly-typed ref cursor. The following example is a declaration of a weakly-typed ref cursor:

```
name SYS_REFCURSOR;
```

Following is an example of a strongly-typed ref cursor:

```
TYPE cursor_type_name IS REF CURSOR RETURN emp%ROWTYPE;
```

Creating the Stored Procedure

The following sample code creates a stored procedure called `refcur_inout_callee`. To create the sample procedure, invoke EDB-PSQL and connect to the Advanced Server host database. Enter the following SPL code at the command line:

```
CREATE OR REPLACE PROCEDURE
  refcur_inout_callee(v_refcur IN OUT SYS_REFCURSOR)
IS
BEGIN
   OPEN v_refcur FOR SELECT ename FROM emp;
END;
```

To use the above defined procedure from .NET code, you must specify the data type of the ref cursor being passed as an `IN` parameter, as shown in the above script.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

The following C# code uses the stored procedure to retrieve employee names from the emp table:

```
using System;
using System.Data;
using EnterpriseDB.EDBClient;
using System.Configuration;

namespace EDBRefCursor
{
    class EmpRefcursor
    {
        [STAThread]
        static void Main(string[] args)
        {
            string strConnectionString =
            ConfigurationSettings.AppSettings["DB_CONN_STRING"];
            EDBConnection conn = new EDBConnection(strConnectionString);
            conn.Open();
            EDBTransaction tran = conn.BeginTransaction();
            try
            {
                EDBTransaction tran = conn.BeginTransaction();
                EDBCommand command = new EDBCommand("refcur_inout_callee",
                    conn);
                command.CommandType = CommandType.StoredProcedure;
                command.Transaction = tran;
                command.Parameters.Add(new EDBParameter("refCursor",
                    EDBTypes.EDBDbType.Refcursor, 10, "refCursor",

                ParameterDirection.InputOutput, false, 2, 2,
                    System.Data.DataRowVersion.Current, null));

                command.Prepare();
                command.Parameters[0].Value = null;

                command.ExecuteNonQuery();
                String cursorName = command.Parameters[0].Value.ToString();
                command.CommandText = "fetch all in \"" + cursorName + "\"";
                command.CommandType = CommandType.Text;

                EDBDataReader reader =
                    command.ExecuteReader(CommandBehavior.SequentialAccess);
                int fc = reader.FieldCount;
                while (reader.Read())
                {
                    for (int i = 0; i < fc; i++)
                    {
                        Console.WriteLine(reader.GetString(i));
                    }
                }
                reader.Close();
                tran.Commit();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message.ToString());
            }
        }
    }
}
```

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.

The following .NET code snippet displays the result on the console:

```
for(int i = 0;i < fc; i++)
{
    Console.WriteLine(reader.GetString(i));
}
```

Please note that you must bind the `EDBDbType.RefCursor` type in `EDBParameter()` if you are using a ref cursor parameter.

Copyright © 2009 - 2018 EnterpriseDB Corporation. All rights reserved.