EDB .NET Connector

Release 4.0.6.1

EDB .NET Connector User's Guide

Sep 22, 2020

Contents

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.

CHAPTER 1

What's New

The following features are added to create Advanced Server .NET Connector `4.0.6.1`:

- The Advanced Server .NET Connector has added Advanced Queueing feature that provides message queueing

and message processing support for the EDB Advanced Server database.

- Merged with the upstream community driver version 4.0.6.

CHAPTER 2

Requirements Overview

The following section details the supported platforms for the Advanced Server .NET Connector.

2.1 Supported Server Versions

The Advanced Server .NET Connector is certified with Advanced Server version 9.4 and above.

2.2 Supported Platforms

The Advanced Server .NET Connector graphical installers are supported on the following Windows platforms:

64-bit Windows:

- Windows Server 2016
- Windows Server 2012 R2

32-bit Windows:

- Windows 10
- Windows 8
- Windows 7

CHAPTER 3

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 5/6, and .Net Standard2.0.

3.1 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.

CHAPTER 4

Installing and Configuring the .NET Connector

This chapter describes how to install and configure the Advanced Server .NET Connector.

4.1 Installing the .NET Connector

You can use the EnterpriseDB .NET Connector Installer (available from the EnterpriseDB website) to add the .NET Connector to your system. After downloading the installer, right-click on the installer icon, and select `Run As Administrator` from the context menu. When prompted, select an installation language and click `OK` to continue to the `Setup` window.

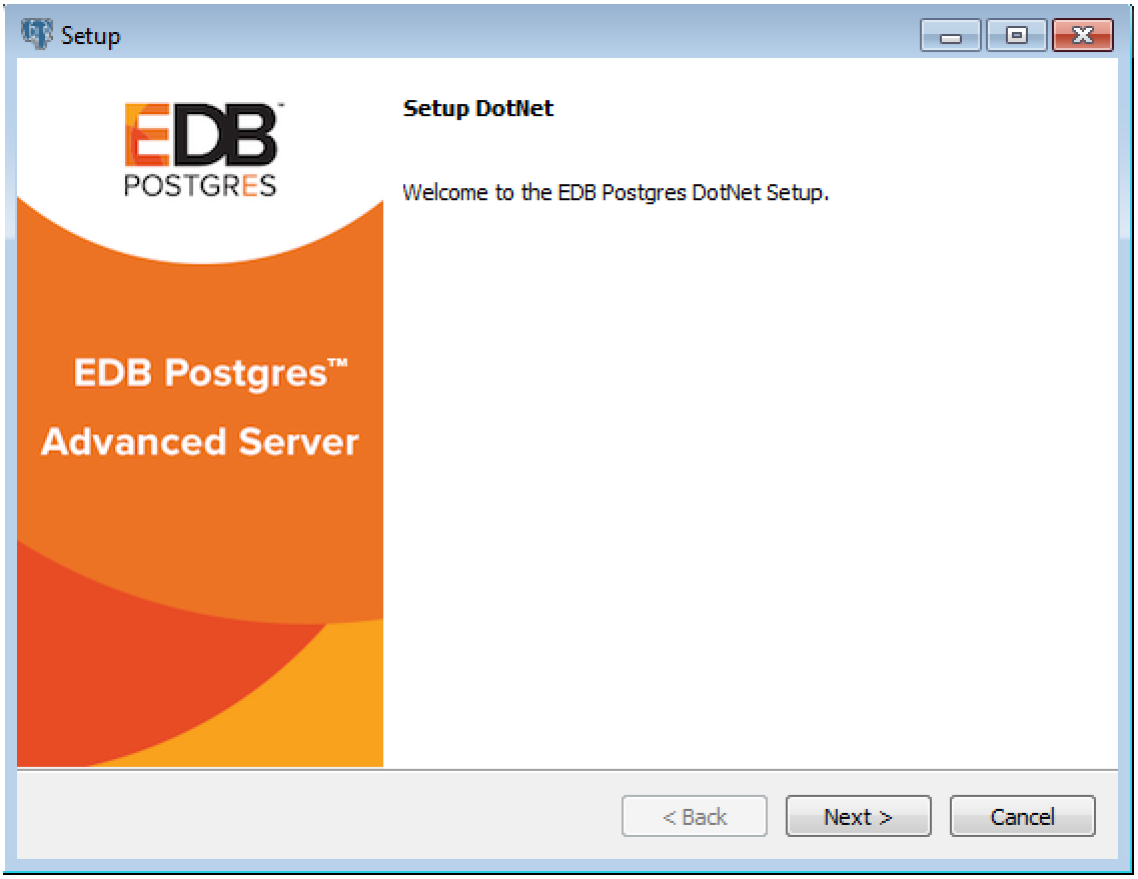

Fig. 1: *The .NET Connector Installation wizard*

Click `Next` to continue.

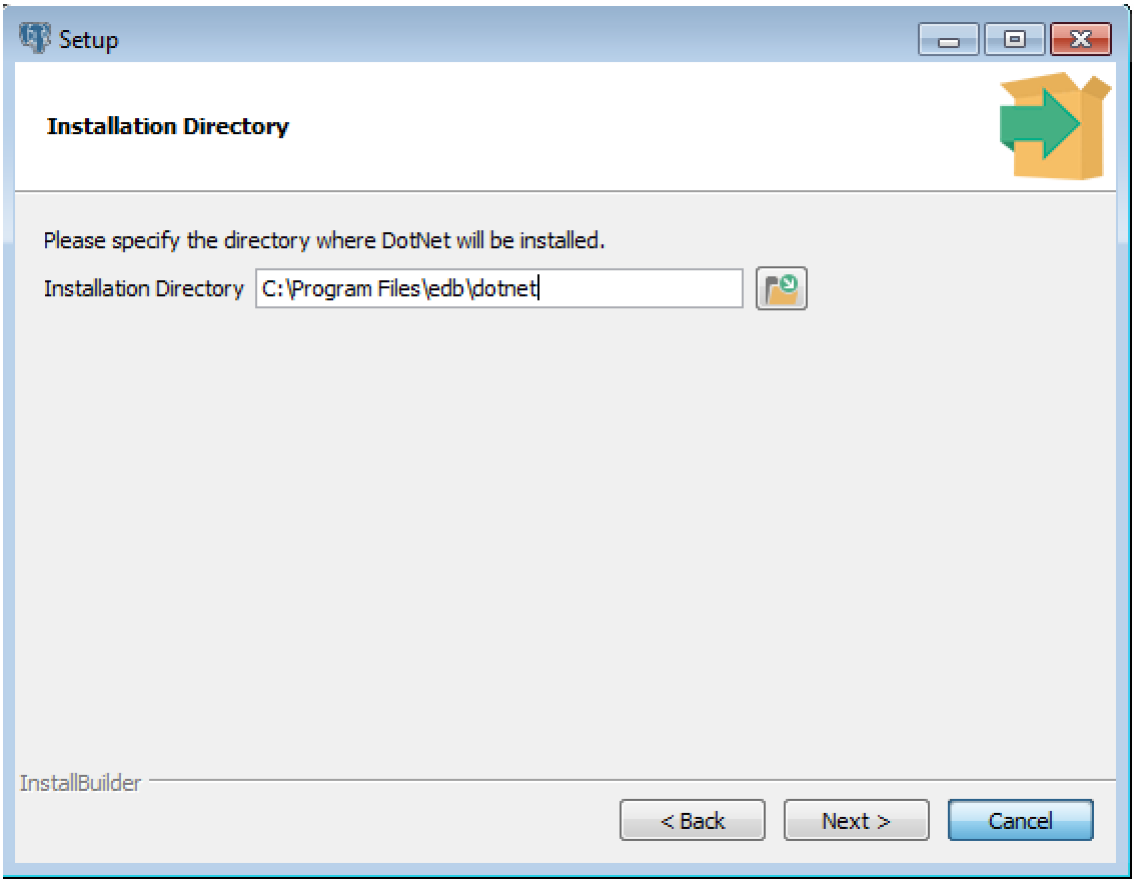

Fig. 2: *The Installation dialog*

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

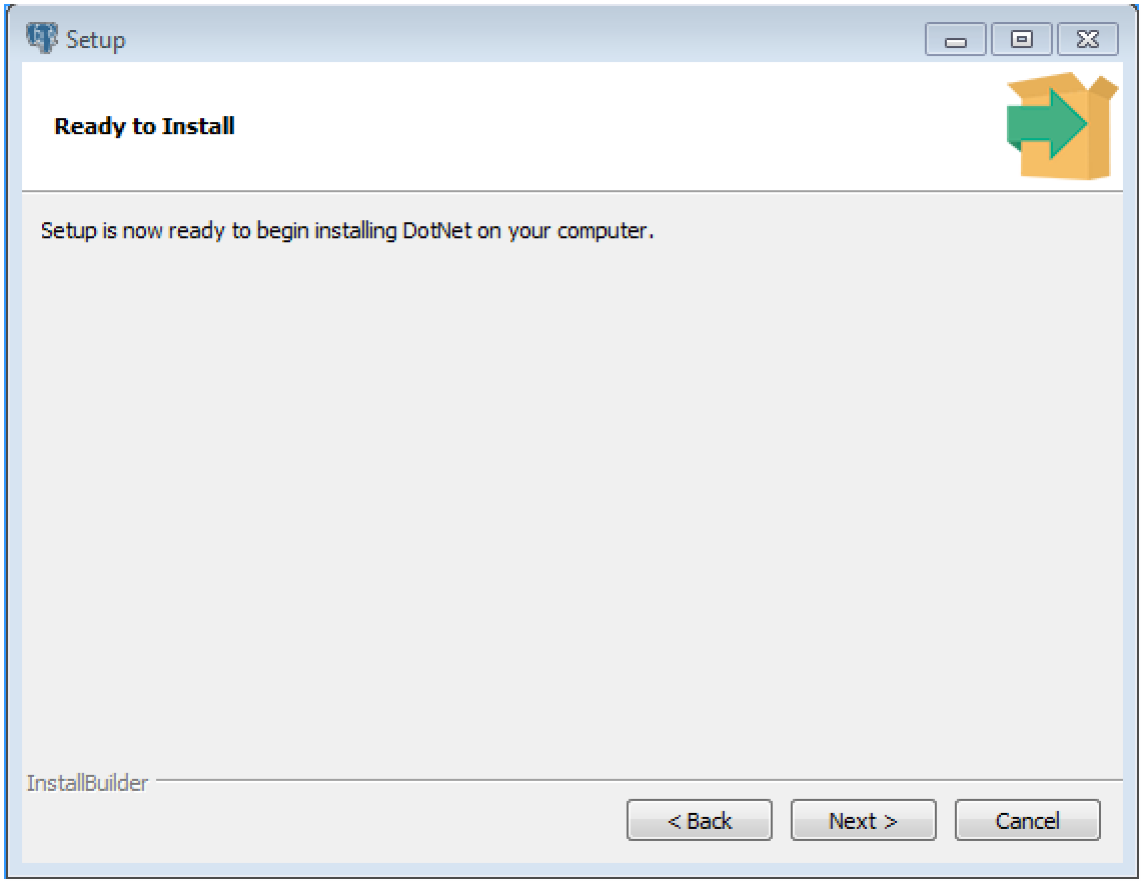

Fig. 3: *The Ready to Install dialog*

Click `Next` on the `Ready to Install` dialog to start the installation; popup dialogs confirm the progress of the installation wizard.

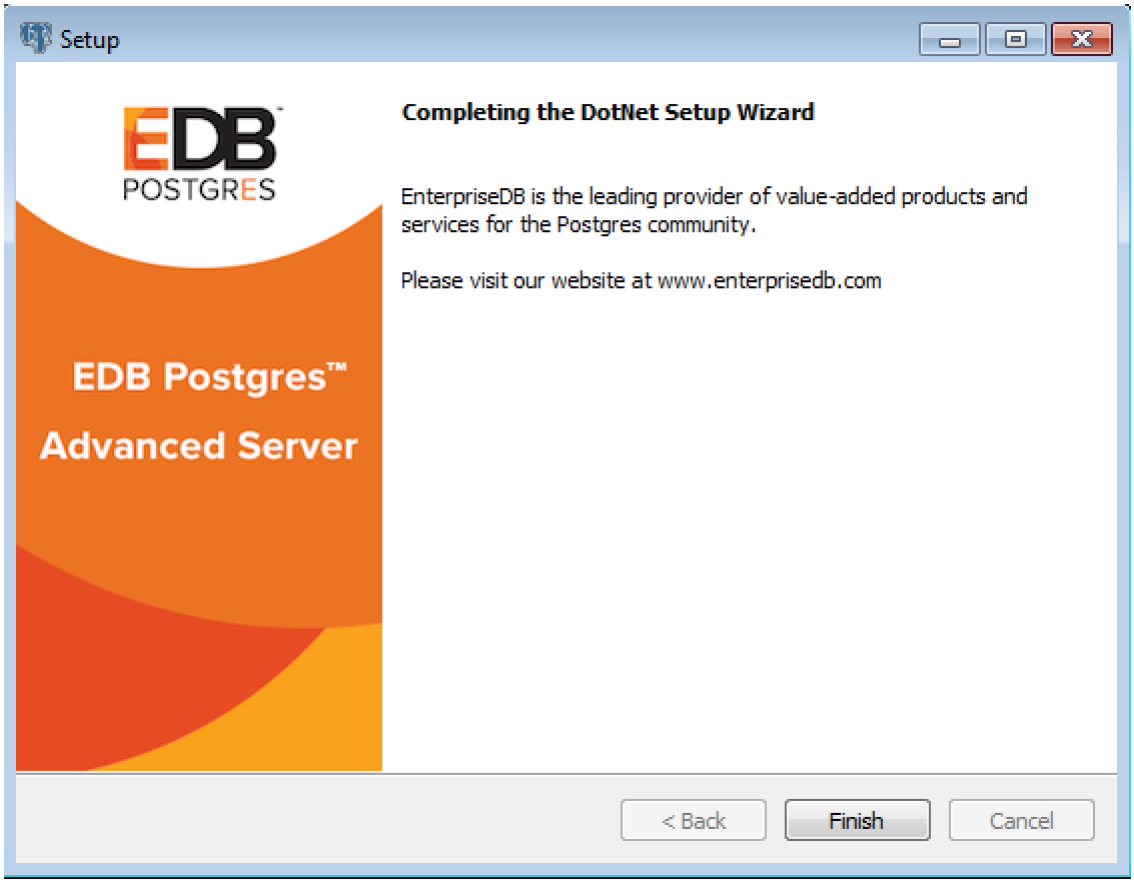

Fig. 4: *The installation is complete*

When the wizard informs you that it has completed the setup, click the `Finish` button to exit the dialog.

You can also 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.

Fig. 5: *Starting StackBuilder Plus*

When StackBuilder Plus opens, follow the onscreen instructions. Select the `EnterpriseDB`.

`Net Connector` option from the `Database Drivers` node of the tree control.

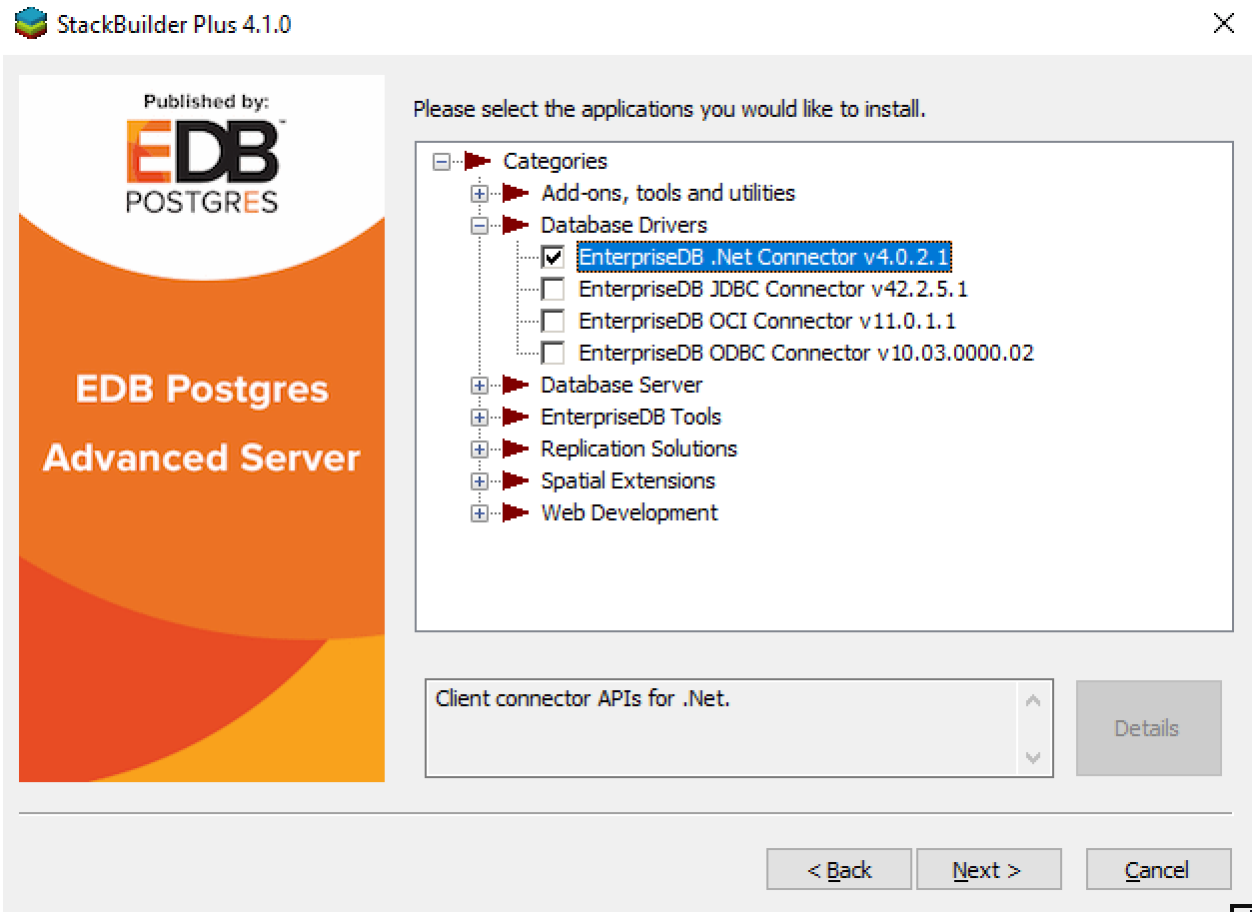

Fig. 6: *Selecting the Connectors installer*

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

4.2 Configuring the .NET Connector

Please see the following environment-specific sections for information about configuring the .NET Connector:

- **Referencing the Library Files.** *General configuration information* applicable to all components.
- **.NET Framework 4.0.** Instructions for configuring for use with *.NET Framework 4.0*.
- **.NET Framework 4.5.** Instructions for configuring for use with *.NET Framework 4.5*.
- **.NET Framework 4.5.1.** Instructions for configuring for use with *.NET Framework 4.5.1*.
- **.NET Standard 2.0.** Instructions for configuring for use with *.NET Standard 2.0*.
- **Entity Framework 5/6.** Instructions for configuring for use with *Entity Framework*.
- **EnterpriseDB VSIX.** Instructions for configuring for use with *EnterpriseDB VSIX*.

4.2.1 Referencing the Library Files

To reference library files with 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:

```
EnterpriseDB.EDBClient
```

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

```
EntityFramework6.EntepriseDB.EDBClient
```

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;
```

4.2.2 .NET Framework Setup

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

4.2.2.1 .NET Framework 4.0

If you are using .NET Framework version 4.0, the data provider installation path is:

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

The following shared library files are required:

```
EDBDataProvider.2.0.2.dll
```

```
Mono.Security.dll
```

see *Referencing the Library Files* for information about referencing library files.

Depending upon the type of application you use, you may be required to import the namespace into the source code (see *Referencing the Library Files*).

4.2.2.2 .NET Framework 4.5

If you are using .NET Framework version 4.5, the data provider installation path is:

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

The following shared library files are required:

```
EnterpriseDB.EDBClient.dll
System.Threading.Tasks.Extensions.dll
System.Runtime.CompilerServices.Unsafe.dll
System.ValueTuple.dll
System.Memory.dll
```

You must also add the following dependencies to your project:

```
System.Threading.Tasks.Extensions.dll
System.Runtime.CompilerServices.Unsafe.dll
System.ValueTuple.dll
System.Memory.dll
```

See *Referencing the Library Files* for information about referencing library files.

Depending upon the type of application you use, you may be required to import the namespace into the source code (see *Referencing the Library Files*).

4.2.2.3 .NET Framework 4.5.1

If you are using .NET Framework version 4.5.1, the data provider installation path is:

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

The following shared library files are required:

```
EnterpriseDB.EDBClient.dll
System.Threading.Tasks.Extensions.dll
System.Runtime.CompilerServices.Unsafe.dll
System.ValueTuple.dll
System.Memory.dll
```

You must also add the following dependencies to your project:

```
System.Threading.Tasks.Extensions.dll
System.Runtime.CompilerServices.Unsafe.dll
System.ValueTuple.dll
System.Memory.dll
```

See *Referencing the Library Files* for information on referencing library files.

Depending upon the type of application you use, you may be required to import the namespace into the source code (see *Referencing the Library Files*).

4.2.2.4 .NET Standard 2.0

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

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

The following shared library files are required:

```
EnterpriseDB.EDBClient.dll
System.Threading.Tasks.Extensions.dll
System.Runtime.CompilerServices.Unsafe.dll
System.ValueTuple.dll
```

You must also add the following dependencies to your project:

```
System.Runtime.CompilerServices.Unsafe.dll
System.ValueTuple.dll
```

Note: If your target framework is .Net Core 2.0, then include the following file in your project:

```
System.Threading.Tasks.Extensions.dll
```

See *Referencing the Library Files* for information about library files.

Depending upon the application type you use, you may be required to import the namespace into the source code (see *Referencing the Library Files*).

4.2.3 Entity Framework 5/6

To set up .NET Connector for usage with Entity Framework, the data provider installation path is:

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

The following shared library files are required:

```
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.

See *Referencing the Library Files* for information about referencing library files.

Add the `<DbProviderFactories>` entries for the `ADO.NET` driver for Postgres to the `app.config` file. Add the following entries:

```
<add name="EnterpriseDB.EDBClient"
  invariant="EnterpriseDB.EDBClient"
  description=".NET Data Provider for EnterpriseDB PostgreSQL"
  type="EnterpriseDB.EDBClient.EDBFactory, EnterpriseDB.EDBClient,␣
↪Version=4.0.10.1, Culture=neutral, PublicKeyToken=5d8b90d52f46fda7"
  support="FF"/>
```

In the projectâ€™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:

```
<?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>
```

(continues on next page)

(continued from previous page)

```
        <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 and the `app.config` file.

Depending upon the type of application you are using, you may be required to import the namespace into the source code (see *Referencing the Library Files*).

For usage information about Entity Framework, refer to the Microsoft documentation.

4.2.4 EnterpriseDB VSIX for Visual Studio 2015/2017/2019

`EDB Data Designer Extensibility Provider (EnterpriseDB VSIX)` is a component that 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.

EnterpriseDB VSIX is located in the following directory:

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

The files available at the above location are the following:

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

4.2.4.1 Installation and Configuration for Visual Studio 2015/2017/2019

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\dotnet\vsix\EnterpriseDB.vsix
```

If you already have an earlier version of the VSIX installed, itâ€™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 Updatesâ€¦` menu that the EnterpriseDB extension is installed.

Step 3: Add the `System.ValueTuple.dll` assembly in the global assembly cache (GAC) with the `gacutil` utility using the Visual Studio Developers Command line from the following location.

```
C:\Program Files\edb\dotnet\vsix\System.ValueTuple.dll
```

For example:

```
> gacutil.exe /i System.ValueTuple.dll
```

Step 4: From Server Explorer, right-click on `Data Connections`, click `Add Connection`, and verify the `Enterprisedb Postgres Database` data source is available.

4.2.4.2 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:

```
<add name="EnterpriseDB.EDBClient"
  invariant="EnterpriseDB.EDBClient"
  description=".NET Data Provider for EnterpriseDB PostgreSQL"
  type="EnterpriseDB.EDBClient.EDBFactory, EnterpriseDB.EDBClient,␣
↪Version=4.0.10.1, Culture=neutral, PublicKeyToken=5d8b90d52f46fda7"
  support="FF"/>
```

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 *Referencing the Library Files* for information about referencing library files.

Step 5: In the projectâ€™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.

```
<?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>
```

CHAPTER 5

Security and Encryption

5.1 Scram Compatibility

The EDB .NET driver provides SCRAM-SHA-256 support for Advanced Server versions 12, 11 and 10. This support is available from EDB .NET 4.0.2.1 release onwards.

CHAPTER 6

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`).

CHAPTER 7

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.

7.1 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`

> `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.

`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.

7.2 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 *Chapter - Using the .Net Connector* for an introduction to connection information and also see *Section - Connection String Parameters* 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:

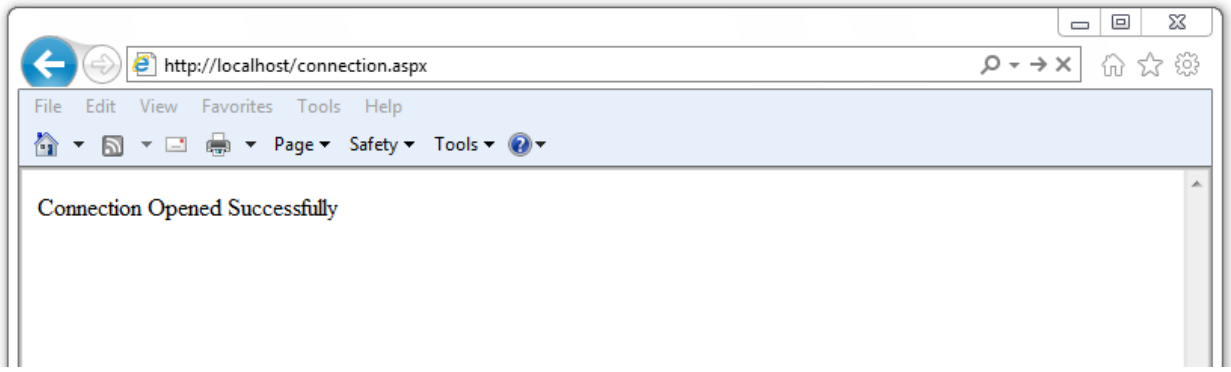

Fig. 1: *Connection Opened Successfully*

7.3 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");
            }
```

(continues on next page)

(continued from previous page)

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

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

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
```

7.4 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);
```

(continues on next page)

(continued from previous page)

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

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:

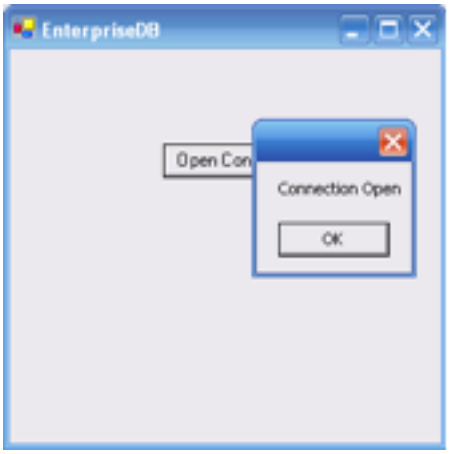

Fig. 2: *A successful connection*

CHAPTER 8

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">

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();
```

(continues on next page)

(continued from previous page)

```
        }
}
</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
```

8.1 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.

CHAPTER 9

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"];
```

(continues on next page)

(continued from previous page)

```
      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>
```

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
```

CHAPTER 10

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();
```

(continues on next page)

(continued from previous page)

```
        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>
```

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
```

CHAPTER 11

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
{
```

(continues on next page)

(continued from previous page)

```
                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>
```

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
```

CHAPTER 12

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.

12.1 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"];
      EDBConnection conn = new EDBConnection(strConnectionString);
```

(continues on next page)

(continued from previous page)

```
      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
```

12.2 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" %>
```

(continues on next page)

(continued from previous page)

```
<% @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 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;
```

(continues on next page)

(continued from previous page)

```
                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);

        }

        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,:Manager)",
 ↪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.

12.3 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:

- 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();
```

(continues on next page)

(continued from previous page)

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

        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);
```

(continues on next page)

(continued from previous page)

```
      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;
        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>
```

CHAPTER 13

Using Advanced Queueing

EDB Postgres Advanced Server Advanced Queueing provides message queueing and message processing for the Advanced Server database. User-defined messages are stored in a queue; a collection of queues is stored in a queue table. You should first create a queue table before creating a queue that is dependent on it.

On the server side, procedures in the `DBMS_AQADM` package create and manage message queues and queue tables. Use the `DBMS_AQ` package to add or remove messages from a queue, or register or unregister a PL/SQL callback procedure. For more information about `DBMS_AQ` and `DBMS_AQADM`, click here.

On the client side, application uses EDB.NET driver to enqueue/dequeue message.

13.1 Enqueue or Dequeue a message

13.1.1 Serve-side setup

To use Advanced Queueing functionality on your .NET application, you must first create a user defined type, queue table, and queue, and then start the queue on the database server. Invoke EDB-PSQL and connect to the Advanced Server host database. Use the following SPL commands at the command line:

Creating user defined type

To specify a RAW data type, you should create a user-defined type. The following example demonstrates creating a user-defined type named as `myxml`.

```
CREATE TYPE myxml AS (value XML);
```

Creating the Queue table

A queue table can hold multiple queues with the same payload type. The following example demonstrates creating a table named `MSG_QUEUE_TABLE`.

```
EXEC DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'MSG_QUEUE_TABLE',
       queue_payload_type => 'myxml',
       comment => 'Message queue table');
END;
```

Creating Queue

The following example demonstrates creating a queue named `MSG_QUEUE` within the table `MSG_QUEUE_TABLE`.

```
BEGIN
DBMS_AQADM.CREATE_QUEUE ( queue_name => 'MSG_QUEUE', queue_table =>
↪'MSG_QUEUE_TABLE', comment => 'This queue contains pending messages.
↪');
END;
```

Starting Queue

Once the queue is created, invoke the following SPL code at the command line to start a queue in the EDB database.

```
BEGIN
DBMS_AQADM.START_QUEUE
(queue_name => 'MSG_QUEUE');
END;
```

13.1.2 Client-side sample

Once you have created user defined type, followed by queue table and queue, start the queue. Then, you can enqueue or dequeue a message using EDB .Net drivers.

Enqueue a message:

To enqueue a message on your .NET application, you must:

1. Import the `EnterpriseDB.EDBClient` namespace.
2. Pass the name of the queue and create the instance of the `EDBAQQueue`.
3. Create the enqueue message and define payload.
4. Call the `queue.Enqueue` method.

The following code listing demonstrates using the `Queue.enqueue` method:

```
using EnterpriseDB.EDBClient;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AQXml
{
    class MyXML
    {
        public string value { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int messagesToSend = 1;
            if (args.Length > 0 && !string.IsNullOrEmpty(args[0]))
            {
                messagesToSend = int.Parse(args[0]);
            }
            for (int i = 0; i < 5; i++)
            {
                EnqueMsg("test message: " + i);
            }
        }

              private static EDBConnection GetConnection()
        {
```

(continues on next page)

(continued from previous page)

```
            string connectionString = "Server=127.0.0.1;Host=127.0.0.1;
↪Port=5444;User Id=enterprisedb;Password=test;Database=edb;Timeout=999
↪";
            EDBConnection connection = new␣
↪EDBConnection(connectionString);
            connection.Open();
            return connection;
        }


        private static string ByteArrayToString(byte[] byteArray)
        {
            // Sanity check if it's null so we don't incur overhead of␣
↪an exception
            if (byteArray == null)
            {
                return string.Empty;
            }
            try
            {
                StringBuilder hex = new StringBuilder(byteArray.Length␣
↪* 2);
                foreach (byte b in byteArray)
                {
                    hex.AppendFormat("{0:x2}", b);
                }

                return hex.ToString().ToUpper();
            }
            catch
            {
                return string.Empty;
            }
        }

        private static bool EnqueMsg(string msg)
        {
            EDBConnection con = GetConnection();
            using (EDBAQQueue queue = new EDBAQQueue("MSG_QUEUE", con))
            {
                queue.MessageType = EDBAQMessageType.Xml;
                EDBTransaction txn = queue.Connection.
↪BeginTransaction();
                QueuedEntities.Message queuedMessage = new␣
↪QueuedEntities.Message() { MessageText = msg };
```

(continues on next page)

(continued from previous page)

```
                try
                {
                    string rootElementName = queuedMessage.GetType().
↪Name;
                    if (rootElementName.IndexOf(".") != -1)
                    {
                        rootElementName = rootElementName.Split('.').
↪Last();
                    }

                    string xml = new Utils.XmlFragmentSerializer
↪<QueuedEntities.Message>().Serialize(queuedMessage);
                    EDBAQMessage queMsg = new EDBAQMessage();
                    queMsg.Payload = new MyXML { value = xml };
                    queue.MessageType = EDBAQMessageType.Udt;
                    queue.UdtTypeName = "myxml";
                    queue.Enqueue(queMsg);
                    var messageId = ByteArrayToString((byte[])queMsg.
↪MessageId);
                    Console.WriteLine("MessageID: " + messageId);
                    txn.Commit();
                    queMsg = null;
                    xml = null;
                    rootElementName = null;
                    return true;
                }
                catch (Exception ex)
                {
                    txn?.Rollback();
                    Console.WriteLine("Failed to enqueue message.");
                    Console.WriteLine(ex.ToString());
                    return false;
                }
                finally
                {
                    queue?.Connection?.Dispose();
                }
            }
        }

    }
}
```

Dequeue a message

To dequeue a message on your .NET application, you must:

1. Import the `EnterpriseDB.EDBClient` namespace.
2. Pass the name of the queue and create the instance of the `EDBAQQueue`.
3. Call the `queue.Dequeue` method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using EnterpriseDB.EDBClient;

namespace DequeueXML
{
    class MyXML
    {
        public string value { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DequeMsg();
        }


        private static EDBConnection GetConnection()
        {
            string connectionString = "Server=127.0.0.1;Host=127.0.0.1;
↪Port=5444;User Id=enterprisedb;Password=test;Database=edb;Timeout=999
↪";
            EDBConnection connection = new␣
↪EDBConnection(connectionString);
            connection.Open();
            return connection;
        }


        private static string ByteArrayToString(byte[] byteArray)
        {
            // Sanity check if it's null so we don't incur overhead of␣
↪an exception
            if (byteArray == null)
            {
                return string.Empty;
```

(continues on next page)

(continued from previous page)

```
            }
            try
            {
                StringBuilder hex = new StringBuilder(byteArray.Length␣
↪* 2);
                foreach (byte b in byteArray)
                {
                    hex.AppendFormat("{0:x2}", b);
                }

                return hex.ToString().ToUpper();
            }
            catch
            {
                return string.Empty;
            }
        }
        public static void DequeMsg(int waitTime = 10)
        {
            EDBConnection con = GetConnection();
            using (EDBAQQueue queueListen = new EDBAQQueue("MSG_QUEUE",
↪ con))
            {
                queueListen.UdtTypeName = "myxml";
                queueListen.DequeueOptions.Navigation =␣
↪EDBAQNavigationMode.FIRST_MESSAGE;
                queueListen.DequeueOptions.Visibility =␣
↪EDBAQVisibility.ON_COMMIT;
                queueListen.DequeueOptions.Wait = 1;
                EDBTransaction txn = null;

                while (1 == 1)
                    {

                    if (queueListen.Connection.State == System.Data.
↪ConnectionState.Closed)
                    {
                        queueListen.Connection.Open();
                    }

                    string messageId = "Unknown";
                    try
                    {
                        // the listen function is a blocking function.␣
↪It will Wait the specified waitTime or until a
```

(continues on next page)

(continued from previous page)

```
                        // message is received.
                        Console.WriteLine("Listening...");
                        string v = queueListen.Listen(null, waitTime);
                        // If we are waiting for a message and we␣
↪specify a Wait time,
                        // then if there are no more messages, we want␣
↪to just bounce out.
                        if (waitTime > -1 && v == null)
                        {
                            Console.WriteLine("No message received␣
↪during Wait period.");
                            Console.WriteLine();
                            continue;
                        }

                        // once we're here that means a message has␣
↪been detected in the queue. Let's deal with it.
                        txn = queueListen.Connection.
↪BeginTransaction();

                        Console.WriteLine("Attempting to dequeue␣
↪message...");
                        // dequeue the message
                        EDBAQMessage deqMsg;
                        try
                        {
                            deqMsg = queueListen.Dequeue();
                        }
                        catch (Exception ex)
                        {
                            if (ex.Message.Contains("ORA-25228"))
                            {
                                Console.WriteLine("Message was not␣
↪there.  Another process must have picked it up.");
                                Console.WriteLine();
                                txn.Rollback();
                                continue;
                            }
                            else
                            {
                                throw;
                            }
                        }

                        messageId = ByteArrayToString((byte[])deqMsg.
↪MessageId);
```

(continues on next page)

(continued from previous page)

```
                        if (deqMsg != null)
                        {
                            Console.WriteLine("Processing received␣
↪message...");
                            // process the message payload
                            MyXML obj = new MyXML();
                            queueListen.Map<MyXML>(deqMsg.Payload,␣
↪obj);

                            QueuedEntities.Message msg = new Utils.
↪XmlFragmentSerializer<QueuedEntities.Message>().Deserialize(obj.
↪value);

                            Console.WriteLine("Received Message:");
                            Console.WriteLine("MessageID: " +␣
↪messageId);
                            Console.WriteLine("Message: " + msg.
↪MessageText);
                            Console.WriteLine("Enqueue Time" +␣
↪queueListen.MessageProperties.EnqueueTime);

                            txn.Commit();

                            Console.WriteLine("Finished processing␣
↪message");
                            Console.WriteLine();

                        }
                        else
                        {
                            Console.WriteLine("Message was not␣
↪dequeued.");
                        }
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine("Failed To dequeue or␣
↪process the dequeued message.");
                        Console.WriteLine(ex.ToString());
                        Console.WriteLine();
                        if (txn != null)
                        {
                            txn.Rollback();
                            if (txn != null)
                            {
```

(continues on next page)

(continued from previous page)

```
                                    txn.Dispose();
                                }
                            }
                        }
                    }
                }

            }
        }
}
```

13.2 EDBAQ Classes

The following EDBAQ classes are used in this application:

EDBAQDequeueMode

The `EDBAQDequeueMode` class lists all the dequeuer modes available.

Value	Description
Browse	Read the message without locking.
Locked	Reads and gets a write lock on the message.
Remove	Deletes the message after reading. This is the default value.
Remove_NoData	Confirms receipt of the message.

EDBAQDequeueOptions

The `EDBAQDequeueOptions` class lists the options available when dequeuing a message.

Property	Description
ConsumerName	The name of the consumer for which to dequeue the message.
DequeueMode	This is set from EDBAQDequeueMode. It represents the locking behavior linked with the dequeue option.
Navigation	This is set from EDBAQNavigationMode. It represents the position of the message that will be fetched.
Visibility	This is set from EDBAQVisibility. It represents whether the new message is dequeued or not as part of the current transaction.
Wait	The wait time for a message as per the search criteria.
Msgid	The message identifier.
Correlation	The correlation identifier.
DeqCondition	The dequeuer condition. It is a Boolean expression.
Transformation	The transformation that will be applied before dequeuing the message.
DeliveryMode	The delivery mode of the dequeued message.

EDBAQEnqueueOptions

The `EDBAQEnqueueOptions` class lists the options available when enqueuing a message.

Property	Description
Visibility	This is set from EDBAQVisibility. It represents whether the new message is enqueued or not as part of the current transaction.
RelativeMsgid	The relative message identifier.
SequenceDeviation	The sequence when the message should be dequeued.
Transformation	The transformation that will be applied before enqueuing the message.
DeliveryMode	The delivery mode of the enqueued message.

EDBAQMessage

The `EDBAQMessage` class lists a message to be enqueued/dequeued.

Property	Description
Payload	The actual message to be queued.
MessageId	The ID of the queued message.

EDBAQMessageProperties

The `EDBAQMessageProperties` lists the message properties available.

Property	Description
Priority	The priority of the message.
Delay	The duration post which the message is available for dequeuing. This is specified in seconds.
Expiration	The duration for which the message is available for dequeuing. This is specified in seconds.
Correlation	The correlation identifier.
Attempts	The number of attempts taken to dequeue the message.
RecipientList	The receipients list that overthrows the default queue subscribers.
ExceptionQueue	The name of the queue where the unprocessed messages should be moved.
EnqueueTime	The time when the message was enqueued.
State	The state of the message while dequeue.
OriginalMsgid	The message identifier in the last queue.
TransactionGroup	The transaction group for the dequeued messages.
DeliveryMode	The delivery mode of the dequeued message.

EDBAQMessageState

The `EDBAQMessageState` class represents the state of the message during dequeue.

Value	Description
Expired	The message is moved to the exception queue.
Processed	The message is processed and kept.
Ready	The message is ready to be processed.
Waiting	The message is in waiting state. The delay is not reached.

EDBAQMessageType

The `EDBAQMessageType` class represents the types for payload.

Value	Description
Raw	The raw message type. Note: Currently, this payload type is not supported.
UDT	The user defined type message.
XML	The XML type message. Note: Currently, this payload type is not supported.

EDBAQNavigationMode

The `EDBAQNavigationMode` class represents the different types of navigation modes available.

Value	Description
First_Message	Returns the first available message that matches the search terms.
Next_Message	Returns the next available message that matches the search items.
Next_Transaction	Returns the first message of next transaction group.

EDBAQQueue

The `EDBAQQueue` class represents a SQL statement to execute `DMBS_AQ` functionality on a PostgreSQL database.

Property	Description
Connection	The connection to be used.
Name	The name of the queue.
MessageType	The message type that is enqueued/dequeued from this queue. For example EDBAQMessageType.Udt.
UdtTypeName	The user defined type name of the message type.
EnqueueOptions	The enqueue options to be used.
DequeuOptions	The dequeue options to be used.
MessageProperties	The message properties to be used.

EDBAQVisibility

The `EDBAQVisibility` class represents the visibility options available.

Value	Description
Immediate	The enqueue/dequeue is not part of the ongoing transaction.
On_Commit	The enqueue/dequeue is part of the current transaction.

Note:

- To review the default options for the above parameters, click here.
- EDBAQ functionality uses user defined types for calling enqueue/dequeue operations. `Server Compatibility Mode=NoTypeLoading` cannot be used with EDBAQ because `NoTypeLoading` will not load any user defined types.

CHAPTER 14

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)
```

(continues on next page)

(continued from previous page)

```
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.

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();
```

(continues on next page)

(continued from previous page)

```
                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());
            }
        }
    }
}
```

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.

CHAPTER 15

Using Plugins

EDB .Net driver plugins are introduced to support the enhanced capabilities for different data types, which are otherwise not available in .Net. The different plugins available support:

- GeoJSON
- Json.NET
- Legacy PostGIS
- NetTopologySuite
- NodaTime
- Rawpostgis

The plugins support the use of spatial, data/time and Json types. The following sections detail the supported frameworks and data provider installation path for these plugins.

15.1 GeoJSON

If you are using the GeoJSON plugin on .NET Framework 4.5, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\GeoJSON\net45
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.GeoJSON.dll
GeoJSON.Net.dll
Newtonsoft.Json.dll
```

If you are using the GeoJSON plugin on .NET Standard 2.0, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\GeoJSON\netstandard2.
0
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.GeoJSON.dll
```

For detailed information about using the GeoJSON plugin, see the Npgsql documentation.

15.2 Json.NET

If you are using the Json.NET plugin on .NET Framework 4.5, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\Json.NET\net45
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.Json.NET.dll
Newtonsoft.Json.dll
```

If you are using the Json.NET plugin on .NET Standard 2.0, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\Json.NET\
netstandard2.0
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.Json.NET.dll
```

For detailed information about using the Json.NET plugin, see the Npgsql documentation.

15.3 LegacyPostGIS

If you are using the LegacyPostGIS plugin on .Net Framework 4.5, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\LegacyPostgis\net45
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.LegacyPostgis.dll
```

If you are using the LegacyPostGIS plugin on .Net Standard 2.0, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\LegacyPostgis\
netstandard2.0
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.LegacyPostgis.dll
```

For detailed information about using the LegacyPostGIS plugin, see the Npgsql documentation.

15.4 NetTopologySuite

If you are using the NetTopologySuite plugin on .Net Framework 4.5, the data provider installation path is:

```
C:\Program Files\edb\dotnet\\ plugins\NetTopologySuite\
net45
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.NetTopologySuite.dll
GeoAPI.dll
NetTopologySuite.dll
NetTopologySuite.IO.PostGis.dll
```

If you are using the NetTopologySuite plugin on .Net Standard 2.0, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\NetTopologySuite\
netstandard2.0
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.NetTopologySuite.dll
```

For detailed information about using the NetTopologySuite type plugin, see the Npgsql documentation.

15.5 NodaTime

If you are using the NodaTime plugin on .Net Framework 4.5, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\NodaTime\net45
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.NodaTime.dll
NodaTime.dll
```

If you are using the NodaTime plugin on .Net Standard 2.0, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\NodaTime\
netstandard2.0
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.NodaTime.dll
```

For detailed information about using the NodaTime plugin, see the Npgsql documentation.

15.6 RawPostGIS

If you are using the RawPostGIS plugin on .Net Framework 4.5, the data provider installation path is:

```
C:\Program Files\edb\dotnet\plugins\RawPostgis\net45
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.RawPostgis.dll
```

If you are using the RawPostGIS type plugin on .Net Standard 2.0, the data provider installation path is:

```
C:\Program\Files\edb\dotnet\plugins\RawPostGis\
netstandard2.0
```

The following shared library files are required:

```
EnterpriseDB.EDBClient.RawPostgis.dll
```

For detailed information about using the RawPostGIS plugin, see the documentation.

CHAPTER 16

API Reference

For information about using the API, see the Npgsql documentation.

Usage notes:

- When using the API, replace references to `Npgsql` with `EnterpriseDB.EDBClient`.
- When referring to classes, replace `Npgsql` with `EDB`. For example, use the `EDBBinaryExporter` class instead of the `NpgsqlBinaryExporter` class.

CHAPTER 17

Conclusion

EDB Postgres .NET Data Provider Guide

Copyright © 2007 - 2020 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

Visit EnterpriseDB on the web at www.enterprisedb.com.

- EnterpriseDB and Postgres Enterprise Manager are registered trademarks of EnterpriseDB Corporation. EDB and EDB Postgres are trademarks of EnterpriseDB Corporation. Oracle is a registered trademark of Oracle, Inc. Other trademarks may be trademarks of their respective owners.

- EDB designs, establishes coding best practices, reviews, and verifies input validation for the logon UI for EDB products where present. EDB follows the same approach for additional input components, however the nature of the product may require that it accepts freeform SQL, WMI or other strings to be entered and submitted by trusted users for which limited validation is possible. In such cases it is not possible to prevent users from entering incorrect or otherwise dangerous inputs.
- EDB reserves the right to add features to products that accept freeform SQL, WMI or other potentially dangerous inputs from authenticated, trusted users in the future, but will ensure all such features are designed and tested to ensure they provide the minimum possible risk, and where possible, require superuser or equivalent privileges.

- EDB does not that warrant that we can or will anticipate all potential threats and therefore our process cannot fully guarantee that all potential vulnerabilities have been addressed or considered.

Index

Symbols

A

C

D

E

I

O

P

R

S

T

U

W