EDB .NET Connector

Version 9.0.3.1

Built at 2025-06-09T15:17:20

1 EDB .NET Connector

The EDB .NET Connector distributed with EDB Postgres Advanced Server provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server. You can:

- Connect to an instance of EDB Postgres Advanced Server.
- Retrieve information from an EDB Postgres Advanced Server database.
- Update information stored on an EDB Postgres Advanced Server database.

To understand these examples, you need a solid working knowledge of C# and .NET. The EDB .NET Connector functionality is built on the core functionality of the Npgsql open source project. For details, see the Npgsql User Guide.

2 Release notes

The EDB .NET connector documentation describes the latest version of EDB .NET connector.

These release notes describe what's new in each release. When a minor or patch release introduces new functionality, indicators in the content identify the version that introduced the new feature.

Version	Release Date
9.0.3.1	21 May 2025
8.0.5.1	22 Nov 2024
8.0.2.1	15 May 2024
7.0.6.2	15 Feb 2024
7.0.6.1	25 Oct 2023
7.0.4.1	07 Jul 2023
6.0.2.1	25 Jul 2022
5.0.7.1	06 Aug 2021
4.1.6.1	17 Dec 2020
4.1.5.1	11 Nov 2020
4.1.3.1	27 Aug 2020
4.0.10.2	12 Mar 2020
4.0.10.1	26 Sep 2019
4.0.6.1	01 Aug 2019

2.1 Version 9.0.3.1

Released: 21 May 2025

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `9.0.3.1` include:

Type	Description	Addresses
Upstream merge	Merged with community .NET driver version 9.0.3. See release notes for more information about merge updates.	
Bug fix	Populated the `EDBAQMessage.MessageId` property with a string uniquely identifying the message, instead of the previously used `byte[]`.	#41979
Deprecation	Removed .NET5, .NET6, and .NET7 targets as they have reached end of support.	

2.2 Version 8.0.5.1

Released: 22 Nov 2024

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `8.0.5.1` include:

Type	Description	Addresses
Upstream merge	Merged with community .NET driver version 8.0.5 and EF Core Driver 8.0.10. See release notes for more information about merge updates.	
Bug fix	Fixed a performance issue. Performance is now improved when reading data while targeting .NET Framework 4.7.2, 4.8, and 4.8.1.	#41979
Enhancement	Added support for EDB Postgres Advanced Server 17.2.	
Enhancement	Added support for `IS TABLE OF`. EDB Postgres Advanced Server supports Oracle nested table collection types created with `CREATE TYPE ... AS TABLE OF` statements. See Using nested tables for more information.	
Deprecation	Removed .NET5 and .NET7 targets as they have reached end of support.	

2.3 Version 8.0.2.1

Released: 15 May 2024

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `8.0.2.1` include:

Type	Description
Upstream merge	Merged with community .NET driver version 8.0.2. See release notes for more information about merge updates.
Security fix	Fixed a security issue CVE-2024-32655. This security fix fixes the Npgsql that was vulnerable to SQL injection via protocol message size overflow.
Bug fix	Fixed an issue for SPL CALLS. SPL CALLs with output parameters are now returning DataReader with a row of parameters on the batch commands.
Bug fix	EnableErrorBarriers is now functional on the batch commands. See the EnableErrorBarriers documentation for more information.

2.4 Version 7.0.6.2

Released: 15 Feb 2024

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `7.0.6.2` include:

Type	Description
Enhancement	.NET packages are now available on nuget.org.
Bug fix	Fixed an issue while any attempt to connect synchronously hung indefinitely, referencing the .Net Framework assembly using non-ASYNC code.

2.5 Version 7.0.6.1

Released: 25 Oct 2023

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `7.0.6.1` include:

> **Deprecation**
>
> This release removes support for .NET 5 and .NET Core 3.1.

Type	Description
Upstream merge	Merged with community .NET driver version 7.0.6. For more information about the merge updates, see community release notes.
Enhancement	Added support for .NET 4.7.2, .NET 4.8, .NET 4.8.1

2.6 Version 7.0.4.1

Released: 07 Jul 2023

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `7.0.4.1` include:

Type	Description
Upstream merge	Merged with community .NET driver version 7.0.4. For more information about the merge updates, see https://www.npgsql.org/doc/release-notes/7.0.html.
Enhancement	Added support for .NET 7.0.

2.7 Version 6.0.2.1

Released: 25 Jul 2022

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `6.0.2.1` include:

Type	Description
Upstream merge	Merged with community .NET driver version 6.0.2. For more information about the merge updates, see https://www.npgsql.org/doc/release-notes/6.0.html.
Enhancement	Support for .NET 6.0 is added.

2.8 Version 5.0.7.1

Released: 06 Aug 2021

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `5.0.7.1` include:

Type	Description
Upstream merge	Merged with the upstream Npgsql driver version 5.0.7. For more information about the merge updates, see https://www.nuget.org/packages/Npgsql/5.0.7.
Enhancement	Support for .NET 5.0 and .NET Core 3.1 (earlier available as .NET Core 3.0).

2.9 Version 4.1.6.1

Released: 17 Dec 2020

The EDB .NET Connector provides connectivity between a .NET client application and an Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `4.1.6.1` include:

Type	Description
Upstream Merge	Merged with the upstream Npgsql driver version 4.1.6. For more information about the merge updates, see https://www.nuget.org/packages/Npgsql/4.1.6.
Enhancement	Support for .NET Framework 4.7.2 and .NET Framework 4.8.

2.10 Version 4.1.5.1

Released: 11 Nov 2020

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `4.1.5.1` include:

Type	Description
Upstream merge	Merged with the upstream Npgsql driver version 4.1.5. For more information about the merge updates, see https://www.nuget.org/packages/Npgsql/4.1.5.
Enhancement	Support for EDB Postgres Advanced Server 13.

2.11 Version 4.1.3.1

Released: 27 Aug 2020

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `4.1.3.1` include:

Type	Description
Upstream merge	Merged with the upstream Npgsql driver version 4.1.3. For more information about the merge updates, see https://www.nuget.org/packages/Npgsql/4.1.3.
Enhancement	Support for .NET Framework 4.6.1, .NET Core 3.0 and .NET Standard 2.1.

2.12 Version 4.0.10.2

Released: 12 Mar 2020

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `4.0.10.2` include:

Type	Description
Enhancement	Added connection parameter, `Load Role Based Tables`.

2.13 Version 4.0.10.1

Released: 26 Sep 2019

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `4.0.10.1` include:

Type	Description
Upstream merge	Merged with the upstream community driver version 4.0.10.
Enhancement	Added support for Windows Server 2019 platform.
Enhancement	Added support for VSIX for Visual Studio 2019.

2.14 Version 4.0.6.1

Released: 01 Aug 2019

The EDB .NET Connector provides connectivity between a .NET client application and an EDB Postgres Advanced Server database server.

New features, enhancements, bug fixes, and other changes in the EDB .NET Connector `4.0.6.1` include:

Type	Description
Upstream merge	Merged with the upstream community driver version 4.0.6.
Enhancement	Added Advanced Queueing feature that provides message queueing and message processing support for the EDB Advanced Server database.

3 Product compatibility

The following sections detail the supported platforms and database versions for the EDB .NET Connector.

Supported platforms

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

64-bit Windows:

- Windows Server 2019 and 2022
- Windows 10 and 11

32-bit Windows:

- Windows 10

Supported database server versions

This table lists the latest EDB .NET Connector versions and their supported corresponding EDB Postgres Advanced Server (EPAS) versions.

EDB .NET Connector	EPAS 17	EPAS 16	EPAS 15	EPAS 14	EPAS 13	EPAS 12
9.0.3.1	Y	Y	Y	Y	Y	Y
8.0.5.1	Y	Y	Y	Y	Y	Y
8.0.2.1	N	Y	Y	Y	Y	Y
7.0.6.2	N	Y	Y	Y	Y	Y
7.0.6.1	N	Y	Y	Y	Y	Y
7.0.4.1	N	N	Y	Y	Y	Y
6.0.2.1	N	N	N	Y	Y	Y
5.0.7.1	N	N	N	N	Y	Y
4.1.6.1	N	N	N	N	Y	Y
4.1.5.1	N	N	N	N	Y	Y
4.1.3.1	N	N	N	N	Y	Y
4.0.10.2	N	N	N	N	N	Y
4.0.10.1	N	N	N	N	N	Y
4.0.6.1	N	N	N	N	N	Y

4 EDB .NET Connector overview

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

The .NET Connector supports the following frameworks:

- .NET 8.0
- .NET Framework 4.7.2, 4.8, and 4.8.1
- .NET Standard 2.0 and 2.1

The .NET class hierarchy

The .NET class hierarchy contains classes that you can use to create objects that control a connection to the EDB Postgres Advanced Server database and manipulate the data stored on the server. The following are a few of the most commonly used object classes.

`EDBDataSource`

`EDBDataSource` is the entry point for all the connections made to the database. It's responsible for issuing connections to the server and efficiently managing them. Starting with EDB .NET Connector 7.0.4.1, you no longer need direct instantiation of `EDBConnection`. Instantiate `EDBDataSource` and use the method provided to create commands or execute queries.

`EDBConnection`

The `EDBConnection` class represents a connection to EDB Postgres Advanced Server. An `EDBConnection` object contains a `ConnectionString` that tells the .NET client how to connect to an EDB Postgres Advanced Server database. Obtain `EDBConnection` from an `EDBDataSource` instance, and use it directly only in specific scenarios, such as transactions.

`EDBCommand`

An `EDBCommand` object contains a SQL command that the client executes against EDB Postgres 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 EDB Postgres 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 EDB Postgres Advanced Server database. You can modify values and use the `EDBDataAdapter` class to update the data stored in an EDB Postgres Advanced Server database.

5 Installing and configuring the .NET Connector

Installing the .NET Connector

You can install the EDB .NET Connector using either the EDB installer or the installer from NuGet.org.

Installing and configuring the .NET Connector from NuGet.org

Install NuGet package via command line

Launch a terminal from your solution folder and run:

```
dotnet add package EnterpriseDB.EDBClient
```

This command downloads and installs the EDB .NET Connector matching your .NET version. Your project is then ready to import the EDB .NET Connector namespace:

```
using EnterpriseDB.EDBClient;
```

You can find all the EDB .NET Connector satellite packages at NuGet.org.

For more information, see the EDB .NET Connector Now Published on NuGet blog post.

Install NuGet package via Visual Studio interface

1. Right-click your project or solution and select **Manage NuGet package**.
2. Search the package using `enterprisedb.edbclient` as the search text.
3. Select the EnterpriseDB.EDBClient package.
4. Select **Install** to proceed to package download and installation.

This command downloads and installs the EDB .NET Connector matching your .NET version. Your project is then ready to import the EDB .NET Connector namespace:

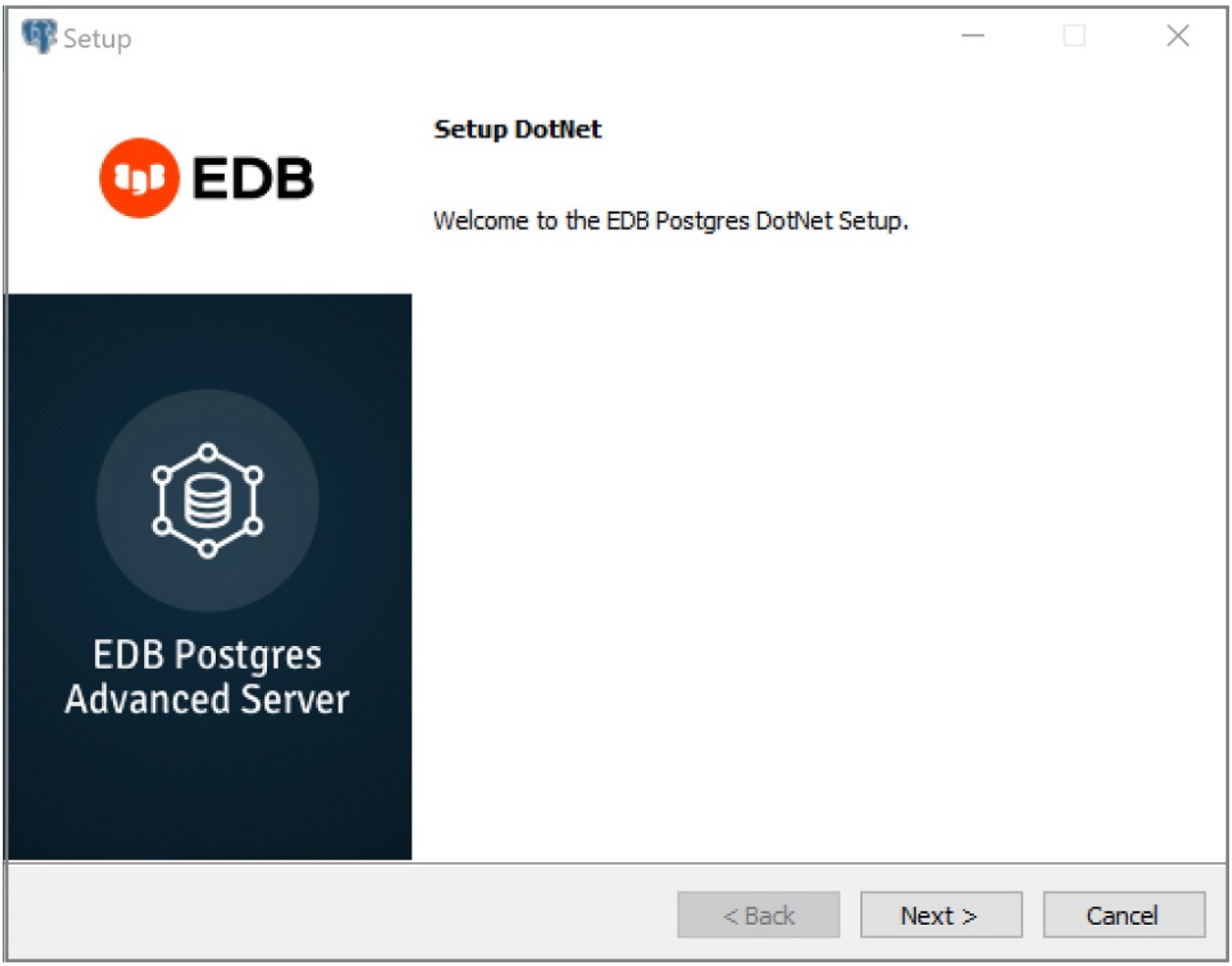
```
using EnterpriseDB.EDBClient;
```

For more information, see the EDB .NET Connector Now Published on NuGet blog post.

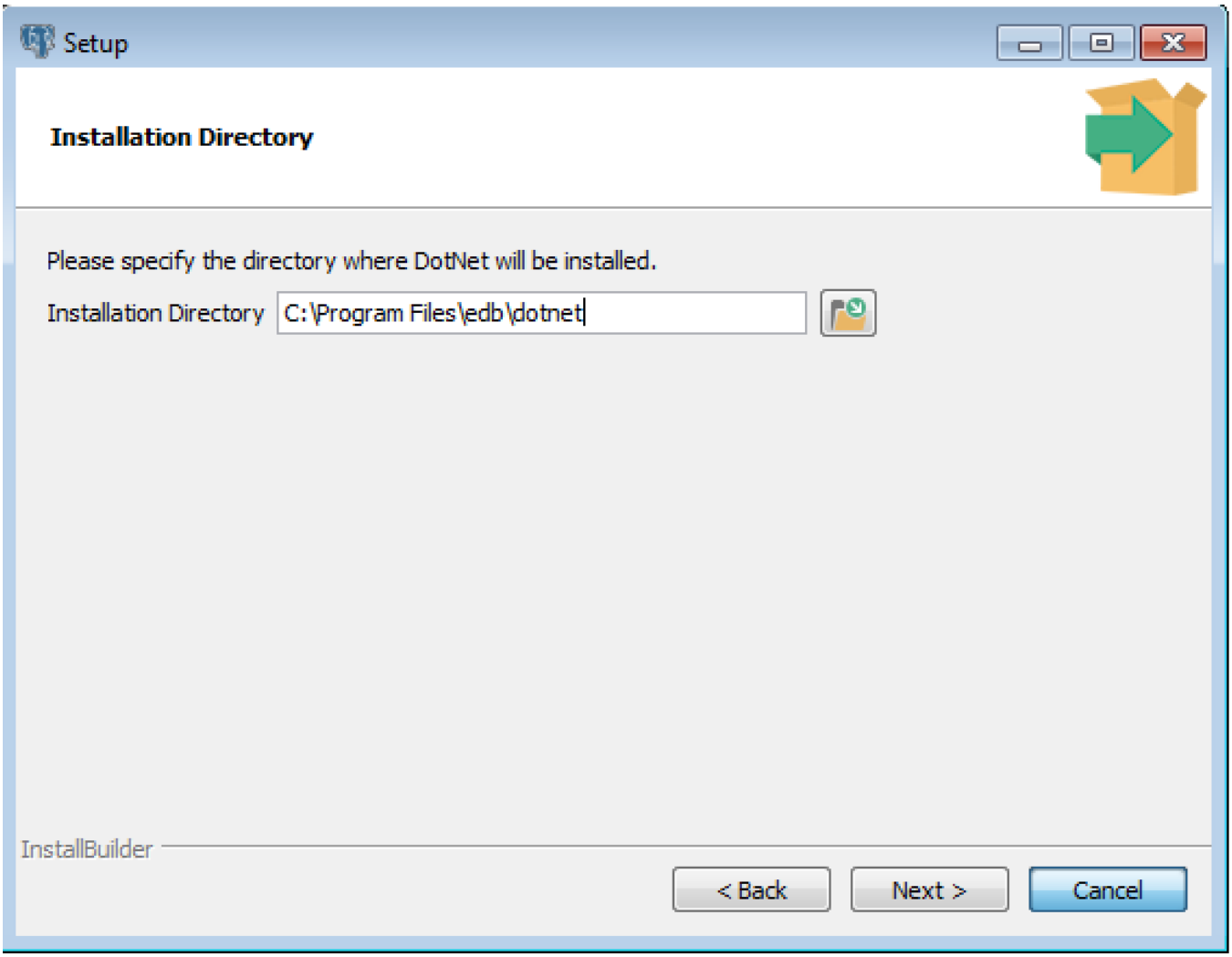
Installing the .NET Connector using EDB installer

You can use the EDB .NET Connector installer to add the .NET Connector to your system. The installer is available from the EDB website.

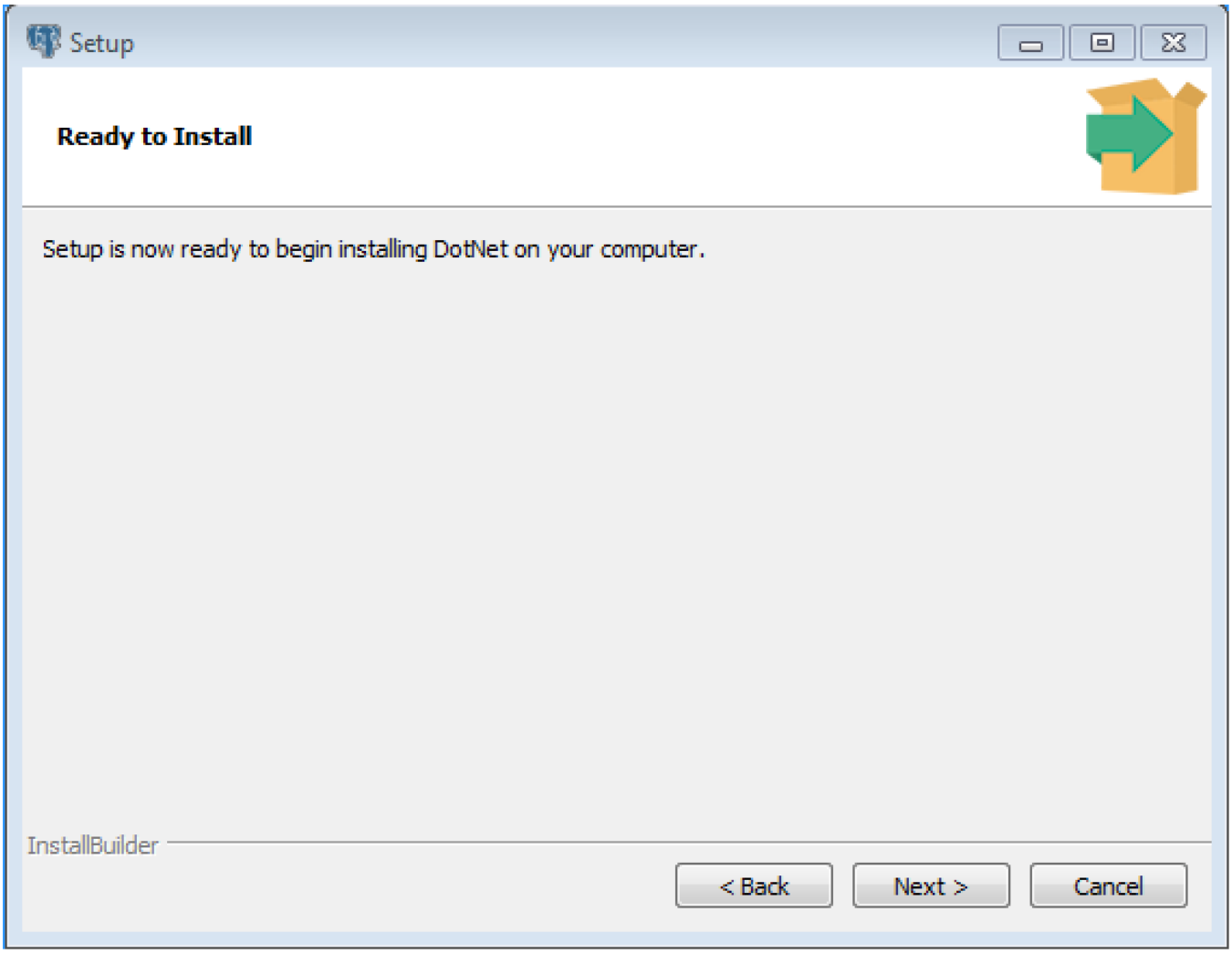
1. After downloading the installer, right-click the installer icon, and select **Run As Administrator**. When prompted, select an installation language and select **OK** to continue to the Setup window.



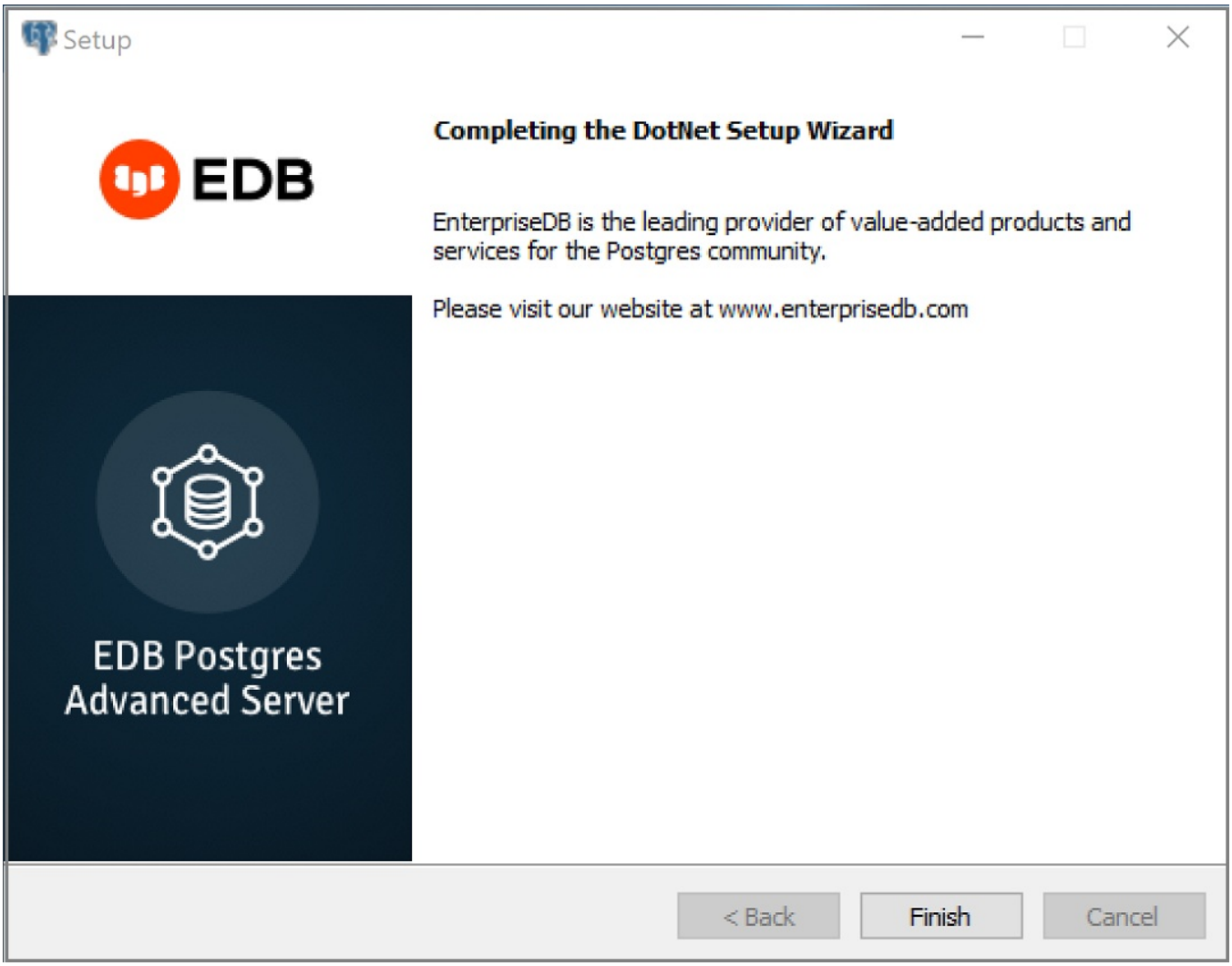
2. Select **Next**.



3. Use the Installation Directory dialog box to specify the directory in which to install the connector. Select**Next**.



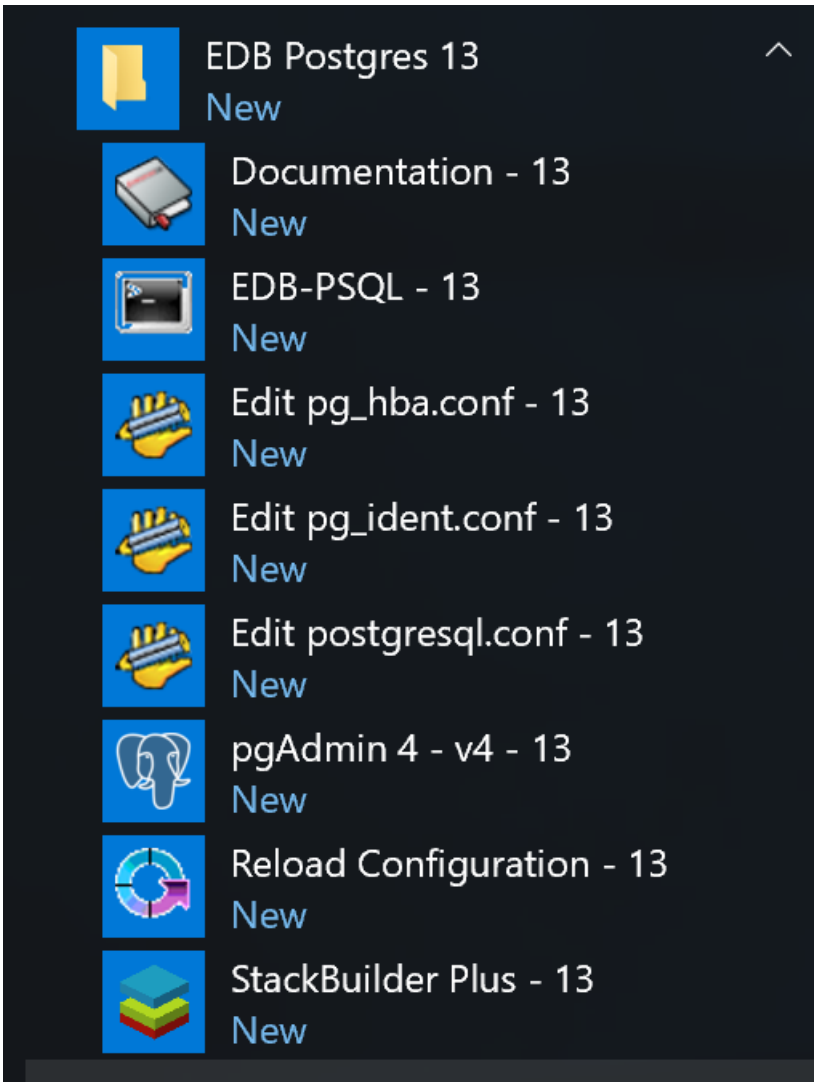
4. To start the installation, on the Ready to Install dialog box, select **Next**. Popups confirm the progress of the installation wizard.



5. When the wizard informs you that it has completed the setup, select **Finish**.

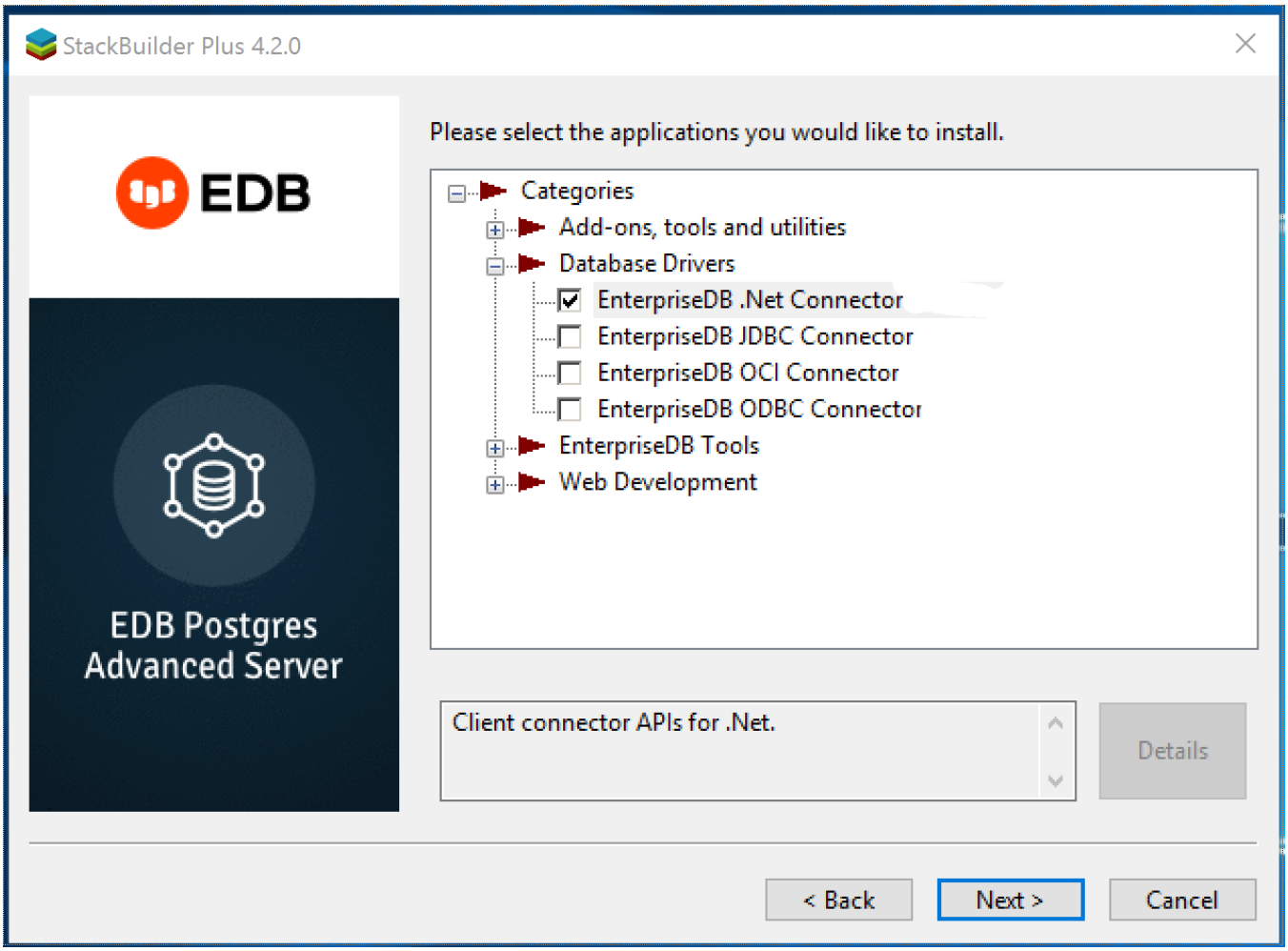
You can also use StackBuilder Plus to add or update the connector on an existing EDB Postgres Advanced Server installation.

1. To open StackBuilder Plus, from the Windows **Apps** menu, select **StackBuilder Plus**.



2. When StackBuilder Plus opens, follow the onscreen instructions.

3. From the Database Drivers node of the tree control, select the **EnterpriseDB.Net Connector** option.



4. Follow the directions of the onscreen wizard to add or update an installation of an EDB Connector.

Configuring the .NET Connector

For information about configuring the .NET Connector in each environment, see:

- **Referencing the Library Files.** General configuration information applicable to all components.
- **.NET 8.0** Instructions for configuring for use with .NET 8.0.
- **.NET Framework 4.7.2** Instructions for configuring for use with .NET framework 4.7.2.
- **.NET Framework 4.8** Instructions for configuring for use with .NET Framework 4.8.
- **.NET Framework 4.8.1** Instructions for configuring for use with .NET Framework 4.8.1.
- **.NET Standard 2.0** Instructions for configuring for use with .NET Standard 2.0.
- **.NET Standard 2.1** Instructions for configuring for use with .NET Standard 2.1.
- **.NET EntityFramework Core** Instructions for configuring for use with .NET EntityFramework Core.

Referencing the library files

To reference library files with Microsoft Visual Studio:

1. In the Solution Explorer, select the project.
2. Select **Project > Add Reference**.
3. In the Add Reference dialog box, browse to select the appropriate library files.

Optionally, you can copy the library files to the specified location.

Before you can use an EDB .NET class, you must import the namespace into your program. Importing a namespace makes the compiler aware of the classes available in the namespace. The namespace is `EnterpriseDB.EDBClient`.

The method you use to include the namespace varies by the type of application you're 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, use:

```
using EnterpriseDB.EDBClient;
```

.NET framework setup

Each .NET version has specific setup instructions.

.NET 8.0

For .NET 8.0, the data provider installation path is:

```
C:\Program Files\edb\dotnet\net8.0\
```

You must add the following dependencies to your project:

```
EnterpriseDB.EDBClient.dll
```

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 for this and other information about referencing library files.

.NET Framework 4.7.2

For .NET Framework 4.7.2, the data provider installation path is:

`C:\Program Files\edb\dotnet\net472\`.

You must add the following dependency to your project. You may also need to add other dependencies from the same directory:

- `EnterpriseDB.EDBClient.dll`

Depending on your application type, you might need to import the namespace into the source code. See Referencing the library files for this and the other information about referencing the library files.

.NET Framework 4.8

For .NET Framework 4.8, the data provider installation path is:

`C:\Program Files\edb\dotnet\net48\`.

You must add the following dependency to your project. You may also need to add other dependencies from the same directory:

- `EnterpriseDB.EDBClient.dll`

Depending on your application type, you might need to import the namespace into the source code. See Referencing the library files for this and the other information about referencing the library files.

.NET Framework 4.8.1

For .NET Framework 4.8.1, the data provider installation path is:

`C:\Program Files\edb\dotnet\net481\`.

You must add the following dependency to your project. You may also need to add other dependencies from the same directory:

- `EnterpriseDB.EDBClient.dll`

Depending on your application type, you might need to import the namespace into the source code. See Referencing the library files for this and the other information about referencing the library files.

.NET Standard 2.0

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

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

You must add the following dependencies to your project:

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

Depending on your application type, you might need to import the namespace into the source code. See Referencing the library files for this and the other information about referencing the library files.

.NET Standard 2.1

For .NET Standard Framework 2.1, the data provider installation path is `C:\Program Files\edb\dotnet\netstandard2.1\`.

The following shared library files are required:

- `EnterpriseDB.EDBClient.dll`
- `System.Memory.dll`
- `System.Runtime.CompilerServices.Unsafe.dll`

- `System.Text.Json.dll`
- `System.Threading.Tasks.Extensions.dll`
- `System.ValueTuple.dll`

Depending on your application type, you might need to import the namespace into the source code. See Referencing the library files for this and the other information about referencing the library files.

.NET Entity Framework Core

To configure the .NET Connector for use with Entity Framework Core, the data provider installation path is:

`C:\Program Files\edb\dotnet\EF.Core\EFCore.PG\net8.0` The following shared library file is required:

- `EnterpriseDB.EDBClient.EntityFrameworkCore.PostgreSQL.dll`

See Referencing the library files for information about referencing the library files.

The following NuGet packages are required:

- `Microsoft.EntityFrameworkCore.Design`
- `Microsoft.EntityFrameworkCore.Relational`
- `Microsoft.EntityFrameworkCore.Abstractions`

For usage information about Entity Framework Core, see the Microsoft documentation.

Prerequisite

To open a command prompt:

Select **Tools > Command Line > Developer Command Prompt**.

Install dotnet-ef (using the command prompt),

```
dotnet tool install --global dotnet-ef
```

Sample project

Create a new Console Application based on .NET 8.0.

Add Reference to the following EDB assemblies:

- `EnterpriseDB.EDBClient.EntityFrameworkCore.PostgreSQL.dll`
- `EnterpriseDB.EDBClient.dll`

Add the following NuGet packages:

- `Microsoft.EntityFrameworkCore.Design`
- `Microsoft.EntityFrameworkCore.Relational`
- `Microsoft.EntityFrameworkCore.Abstractions`

Database-first scenario

Issue the following command to create model classes corresponding to all objects in the specified database:

```
dotnet ef dbcontext scaffold Host=<HOST>;Database=<DATABASE>;Username=<USER>;Password=<PASSWORD>;Port=
<PORT> EnterpriseDB.EDBClient.EntityFrameworkCore.PostgreSQL -o Models
```

Code-first scenario

Add code for defining a DbContext and create, read, update, and delete operations.

For more details, see the Microsoft documentation.

Issue the following commands to create the initial database and tables:

```
  dotnet ef migrations add InitialCreate --context BloggingContext

  dotnet ef database update --context BloggingContext
```

6 Using the .NET Connector

These examples show how you can use the EDB object classes that are provided by the EDB .NET Connector. These object classes allow a .NET application to connect to and interact with an EDB Postgres Advanced Server database.

To use these examples, place the .NET library files in the same directory as the compiled form of your application. All of these examples are written in C#, and each is embedded in an ASP.NET page. The same logic and code applies to other .NET applications (WinForm or console applications, for example).

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 EDB Postgres 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 EDB Postgres Advanced Server connection string for an ASP.NET web application is stored in the `web.config` file. If you're writing an application that doesn't use ASP.NET, provide the connection information in an application configuration file, such as `app.config`.

7 Opening a database connection

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

Creating an EDBConnection object

You can open a connection using one of the following approaches. In either case, you must import the namespace `EnterpriseDB.EDBClient`.

Connection with a data source

1. Create an instance of the `EDBDataSource` object using a connection string as a parameter to the create method of the `EDBDataSource` class.

2. To open a connection, call the `OpenConnection` method of the `EDBDataSource` object.

This example shows how to open a connection using a data source:

```
await using var dataSource = EDBDataSource.Create(ConnectionString);
var connection = dataSource.OpenConnection();
```

Connection without a data source

1. Create an instance of the `EDBConnection` object using a connection string as a parameter to the constructor of the `EDBConnection` class.

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

This example shows how to open a connection without a data source:

```
EDBConnection conn = new EDBConnection(ConnectionString);
conn.Open();
```

> **Note**
>
> For `EnterpriseDB.EDBClient 7.0.4` and later, we recommend `EDBDataSource` to connect to EDB Postgres Advanced Server database or execute SQL directly against it. For more information on the data source, see the Npgsql documentation.

Connection string parameters

A valid connection string specifies location and authentication information for an EDB Postgres 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 EDB Postgres Advanced Server database
- The name of an EDB Postgres Advanced Server user
- The password associated with that user

You can include the following parameters in the connection string:

`CommandTimeout`

`CommandTimeout` specifies the length of time (in seconds) to wait for a command to finish executing 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 for the application to connect to. The default is 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`

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

`Load Role Based Tables`

Use `Load Role Based Tables` to load table OIDs based on role. This change affects only the loading of table type OID and not the composite type. Setting this parameter to `true` triggers the new functionality. The default value is `false`.

`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 preallocate the specified number of connections with the server. The default value is `1`.

`Password`

When using clear text authentication, specify the password to use to establish a connection with the server.

`Pooling`

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

`No Reset On Close`

When `Pooling` is enabled and the connection is closed, reopened, and the underlying connection is reused, then some operations are executed to discard the previous connection resources. You can override this behavior by enabling `No Reset On Close`.

`Port`

The `Port` parameter specifies the port for the application to connect to.

`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 EDB Postgres Advanced Server host.

`SSL`

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

`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 can't be established.
- `allow` — Connect without SSL. This parameter isn't supported.
- `disable` — Don't attempt an SSL connection. This is the default behavior.

`SyncNotification`

Use the `SyncNotification` parameter to specify for `EDBDataprovider` to 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 to use for the connection.

Example: Opening a database connection using ASP.NET

This example shows how to open a connection to an instance of EDB Postgres Advanced Server and then close the connection. The connection is established using the credentials specified in the `DB_CONN_STRING` configuration parameter. See Using the .Net Connector for an introduction to connection information. Also see 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(strConnectionString);
        var conn =
dataSource.OpenConnection();
        Response.Write("Connection opened
successfully");
        conn.Close();
    }
    catch(EDBException
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

If the connection is successful, a message appears indicating that the connection opened successfully.

Example: Opening a database connection from a console application

This example opens a connection with an EDB Postgres 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>
```

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)
        {
            var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
            try
            {
                await using var dataSource = EDBDataSource.Create(strConnectionString);
                var conn = dataSource.OpenConnection();
                Console.WriteLine("Connection Opened
Successfully");
                conn.Close();
            }
            catch(Exception
exp)
            {
            throw new Exception(exp.ToString());
            }
        }
    }
}
```

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

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

Compiling the sample generates a `Console.exe` file. You can execute the sample code by entering `Console.exe`. When executed, the console verifies that it opened successfully.

Example: Opening a database connection from a Windows form application

This example opens 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)
        {
            try
            {
                var strConnectionString =
"Server=localhost;port=5444;username=edb;password=edb;database=edb";
                await using var dataSource = EDBDataSource.Create(strConnectionString);
                var conn = dataSource.OpenConnection();
                MessageBox.Show("Connection Opened
Successfully");
                conn.Close();
          }
          catch(EDBException
exp)
          {
              MessageBox.Show(exp.ToString());
          }
        }
    }
}
```

Change the database connection string to point to the database that you want to connect to. Then compile the file with the following command:

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

This command generates a `WinForm.exe` file in the same folder that the executable was compiled under. Invoking the executable displays a message that the connection was successful.

8 Retrieving database records

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

1. Create and open a database connection.
2. Create an `EDBCommand` object that represents the `SELECT` statement.
3. Execute the command with the `ExecuteReader()` method of the `EDBCommand` object returning `EDBDataReader`.
4. 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 EDB Postgres 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.

Loop through the contents of the `EDBDataReader` object to display the records returned by the query in a `WHILE` loop.

The `Read()` method advances to the next record (if there is one) and returns `true` if a record exists. It returns `false` if `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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(strConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var cmdSelect = new EDBCommand("SELECT * FROM dept",
conn);
        cmdSelect.CommandType =
CommandType.Text;
        using var drDept = await
cmdSelect.ExecuteReaderAsync();
        while (await
drDept.ReadAsync())
        {
            Response.Write("Department Number: " +
drDept["deptno"]);
            Response.Write("\tDepartment Name: " +
drDept["dname"]);
            Response.Write("\tDepartment Location: " +
drDept["loc"]);
            Response.Write("
<br>");
        }
        await
drDept.CloseAsync();
        await conn.CloseAsync();
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

To exercise the sample code, save the code in your default web root directory in a file named `selectEmployees.aspx`. Then, to invoke the program, enter the following in a browser: `http://localhost/selectEmployees.aspx`.

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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(strConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var cmd = new EDBCommand("SELECT MAX(sal) FROM emp",
conn);
        cmd.CommandType =
CommandType.Text;
        var maxSal = Convert.ToInt32(await
cmd.ExecuteScalarAsync());
        Response.Write("Max Salary: " +
maxSal);
        await conn.CloseAsync();
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

Save the sample code in a file named `selectscalar.aspx` in a web root directory.

To invoke the sample code, enter the following in a browser: `http://localhost/selectScalar.aspx`

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 to an integer value by using the `Convert.ToInt32` method.

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 when you write your code. For example, the value referenced in a `WHERE` clause can be calculated from user input.

As shown 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 shows using 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    var updateQuery  = "UPDATE emp SET sal = sal+500 where empno =
:ID";
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var cmdUpdate = new EDBCommand(updateQuery,
conn);
        cmdUpdate.Parameters.Add(new EDBParameter(":ID",
EDBTypes.EDBDbType.Integer));
        cmdUpdate.Parameters[0].Value = 7788;
        await cmdUpdate.ExecuteNonQueryAsync();
        Response.Write("Record
Updated");
        await conn.CloseAsync();
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

Save the sample code in a file named `updateSalary.aspx` in a web root directory.

To invoke the sample code, enter the following in a browser: `http://localhost/updateSalary.aspx`

10 Inserting records in a database

You can use the `ExecuteNonQuery()` method of `EDBCommand` to add records to a database stored on an EDB Postgres 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 an 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        var cmdQuery = "INSERT INTO emp(empno,ename) VALUES(:EmpNo,
:EName)";
        using var cmdInsert = new EDBCommand(cmdQuery,
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";
        await cmdInsert.ExecuteNonQueryAsync();
        Response.Write("Record inserted
successfully");
        await conn.CloseAsync();
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

Save the sample code in a file named `insertEmployee.aspx` in a web root directory.

To invoke the sample code, enter the following in a browser: `http://localhost/insertEmployee.aspx`

11 Deleting records in a database

You can use the `ExecuteNonQuery()` method of `EDBCommand` to delete records from a database stored on an EDB Postgres 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 specified by `EmpNo` to the `DELETE` command. The command is then executed using the `ExecuteNonQuery()` method.

```
<% @ 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    var strDeleteQuery  = "DELETE FROM emp WHERE empno =
:ID";
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        var strDeleteQuery = "DELETE FROM emp WHERE empno =
:ID";
        using var deleteCommand = new EDBCommand(strDeleteQuery,
conn);
        deleteCommand.Parameters.Add(new EDBParameter(":ID",
EDBTypes.EDBDbType.Integer));
        deleteCommand.Parameters[0].Value = 1234;
        await deleteCommand.ExecuteNonQueryAsync();
        Response.Write("Record
Deleted");
        await conn.CloseAsync();
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

Save the sample code in a file named `deleteEmployee.aspx` in a web root directory.

To invoke the sample code, enter the following in a browser: `http://localhost/deleteEmployee.aspx`

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 an improvement in performance by using stored procedures. A stored procedure needs to be parsed, compiled, and optimized only once on the server side. A SQL statement that's included in an application might be parsed, compiled, and optimized each time it's executed from a .NET application.

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

1. Create an SPL stored procedure on the EDB Postgres 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.

Example: Executing a stored procedure without parameters

This 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 EDB Postgres 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 EDB Postgres Advanced Server validates the stored procedure, it echoes `CREATE PROCEDURE`.

Using the EDBCommand object to execute a stored procedure

The `CommandType` property of the `EDBCommand` object indicates 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.

This 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var cmdStoredProc = new EDBCommand("list_dept10",
conn);
        cmdStoredProc.CommandType =
CommandType.StoredProcedure;
        await cmdStoredProc.PrepareAsync();
        await cmdStoredProc.ExecuteNonQueryAsync();
        Response.Write("Stored Procedure Executed
Successfully");
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

Save the sample code in a file named `storedProc.aspx` in a web root directory.

To invoke the sample code, enter the following in a browser: `http://localhost/storedProc.aspx`

Example: Executing a stored procedure with IN parameters

This example calls a stored procedure that includes `IN` parameters. To create the sample procedure, invoke EDB-PSQL and connect to the EDB Postgres 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 EDB Postgres Advanced Server validates the stored procedure, it echoes `CREATE PROCEDURE`.

Passing input values to a stored procedure

Calling a stored procedure that contains parameters is 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.

This example shows the process of executing a parameterized stored procedure from a C# script:

```
<% @ 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    var empName    =
"EDB";
    var empJob     =
"Manager";
    var salary     =
1000.0;
    var commission =
0.0;
    var deptno     =
20;
```

```
    var manager    =
7839;
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var 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.Real));
        cmdStoredProc.Parameters[2].Value =
salary;
        cmdStoredProc.Parameters.Add(new EDBParameter
        ("Commission",
EDBTypes.EDBDbType.Real));
        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;
        await cmdStoredProc.PrepareAsync();
        await cmdStoredProc.ExecuteNonQueryAsync();
        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);
        await conn.CloseAsync();
    }
    catch(Exception
exp)
    {

Response.Write(exp.ToString());
    }
```

```
}
</script>
```

Save the sample code in a file named `storedProcInParam.aspx` in a web root directory.

To invoke the sample code, enter the following in a browser: `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 executes, a test record is inserted into the `emp` table, and the values inserted are displayed on the web page.

Example: Executing a stored procedure with IN, OUT, and INOUT parameters

The previous example showed how to pass `IN` parameters to a stored procedure. The following examples show 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, invoke EDB-PSQL and connect to the EDB Postgres 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 EDB Postgres Advanced Server validates the stored procedure, it echoes `CREATE PROCEDURE`.

Receiving output values from a stored procedure

When retrieving values from `OUT` parameters, you must explicitly specify the direction of those 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 can retrieve the `OUT` and `INOUT` parameter values from the `command.Parameters[]` array.

This code shows 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var 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));
        await command.PrepareAsync();
        command.Parameters[0].Value = 10;
        await using var result = await
command.ExecuteReaderAsync();
        var fc =
result.FieldCount;
        for (var i = 0; i <= fc;
i++)
        {
            Response.Write("RESULT[" + i + "]=" +
Convert.ToString(command.Parameters[i].Value));

Response.Write("\n");
        }
        await
result.CloseAsync();
        await conn.CloseAsync();
    }
    catch(EDBException
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

This code shows 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)
{
    var strConnectionString =
ConfigurationManager.AppSettings["DB_CONN_STRING"];
    try
    {
        await using var dataSource =
EDBDataSource.Create(ConnectionString);
        var conn = await
dataSource.OpenConnectionAsync();
        using var 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));
        await command.PrepareAsync();
        command.Parameters[0].Value = 10;
        await command.ExecuteNonQueryAsync();

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

Response.Write(command.Parameters["pLOC"].Value.ToString());
        await conn.CloseAsync();
    }
    catch(EDBException
exp)
    {

Response.Write(exp.ToString());
    }
}
</script>
```

13 Using advanced queueing

EDB Postgres Advanced Server advanced queueing provides message queueing and message processing for the EDB Postgres Advanced Server database. User-defined messages are stored in a queue. A collection of queues is stored in a queue table. Create a queue table before creating a queue that depends 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 messages to or remove messages from a queue or register or unregister a PL/SQL callback procedure. For more information about `DBMS_AQ` and `DBMS_AQADM`, see DBMS_AQ.

On the client side, the application uses the EDB.NET driver to enqueue and dequeue messages.

Enqueueing or dequeueing a message

For more information about using EDB Postgres Advanced Server's advanced queueing functionality, seeBuilt-in packages.

Server-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 EDB Postgres Advanced Server host database. Use the following SPL commands at the command line.

Creating a user-defined type

To specify a RAW data type, create a user-defined type. This example shows creating a user-defined type named `myxml`:

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

Creating the queue table

A queue table can hold multiple queues with the same payload type. This example shows 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 the queue

This example shows creating a queue named `MSG_QUEUE` in 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 the 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;
```

Client-side example

Once you've created a user-defined type, the queue table, and the 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 an `EDBAQMessage` message set its payload.
4. Call the `EDBAQQueue.Enqueue` method.
5. The `EDBAQMessage.MessageID` property will be populated with a string uniquely identifying your message.

The following code shows how to use `EDBAQQueue.Enqueue` method. A custom message payload is created and then enqueued.

> **Note**
>
> As an example, we are using the ambient Connection via `EDBAQQueue.Connection` to begin a transaction, so that if anything goes wrong the queue won't be polluted.

```
// .NET Framework 4.7.2
using
System;
using
System.Threading.Tasks;
using EnterpriseDB.EDBClient;

namespace EnterpriseDB
{
    internal static class Program
    {
        // Sample message
payload
        class MyXML
        {
            public string Value { get; set; }
        }

        public static async Task Main(string[] args)
        {
            // not for production, move connection string to app
settings
            string connectionString = "Server=127.0.0.1;Port=5444;User
Id=enterprisedb;Password=edb;Database=edb";
```

```
            // Note registration of MyXml type
            var dataSourceBuilder = new EDBDataSourceBuilder(connectionString);
            dataSourceBuilder
                .MapComposite<MyXML>("enterprisedb.myxml");

            using (var dataSource = dataSourceBuilder.Build())
            using (var connection = await dataSource.OpenConnectionAsync())
            using (var queue = CreateQueue("MSG_QUEUE", connection))
            {
                // Enqueue 5
messages
                int messagesToSend =
5;
                for (int i = 0; i < messagesToSend;
i++)
                {
                    var payload = new MyXML()
                    {
                        Value = $"(<Message><MessageText>Test message: {i}</MessageText>
</Message>)"
                    };

                    if (TryEnqueueMessage(queue, payload, out var
_))
                    {
                        // MessageId is populated with a unique
identifier
                        Console.WriteLine($"Message {i} ({message.MessageId})
enqueued");
                    }
                    else
                    {
                        Console.WriteLine($"Message {i} enqueue
failed");
                    }
                }
            }
        }

        // Creates and returns a queue ready for use in our
sample
        private static EDBAQQueue CreateQueue(string queueName, EDBConnection connection)
        {
            var queue = new EDBAQQueue(queueName, connection);
            queue.MessageType =
EDBAQMessageType.Udt;
            queue.EnqueueOptions.Visibility = EDBAQVisibility.ON_COMMIT;
            queue.UdtTypeName = "myxml";

            return queue;
        }

        // Enqueues the
payload
        // If the enqueuing was successfull, message variable receives the queue message and the
function returns true
        // otherwise message is null and the function returns
false
        private static bool TryEnqueueMessage<T>(EDBAQQueue queue, T payload, out EDBAQMessage
message)
        {
            using (EDBTransaction transaction =
queue.Connection.BeginTransaction())
```

```
            {
                try
                {
                    message = new EDBAQMessage() { Payload = payload };
                    queue.Enqueue(message);

transaction.Commit();

                    return true;
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Error while enqueing message:
{ex.Message}");

transaction?.Rollback();

                    message = null;
                    return false;
                }
            }
        }
    }
}

// .NET8
using EnterpriseDB.EDBClient;

namespace EnterpriseDB;

internal static class Program
{
    // Sample message
payload
    class MyXML
    {
        public string Value { get; set; }
    }

    public static async Task Main(string[] args)
    {
        // not for production, move connection string to app
settings
        string connectionString = "Server=127.0.0.1;Port=5444;User
Id=enterprisedb;Password=edb;Database=edb";

        // Note registration of MyXml type
        var dataSourceBuilder = new EDBDataSourceBuilder(connectionString);
        dataSourceBuilder
            .MapComposite<MyXML>("enterprisedb.myxml");

        await using var dataSource = dataSourceBuilder.Build();
        await using var connection = await dataSource.OpenConnectionAsync();
        using var queue = CreateQueue("MSG_QUEUE", connection);

        // Enqueue 5
messages
        int messagesToSend =
5;
        for (int i = 0; i < messagesToSend;
i++)
        {
```

```
            var payload = new MyXML()
            {
                Value = $"(<Message><MessageText>Test message: {i}</MessageText>
</Message>)"
            };

            if (TryEnqueueMessage(queue, payload, out var
_))
            {
                // MessageId is populated with a unique
identifier
                Console.WriteLine($"Message {i} ({message.MessageId})
enqueued");
            }
            else
            {
                Console.WriteLine($"Message {i} enqueue
failed");
            }
        }

    }

    // Creates and returns a queue ready for use in our
sample
    private static EDBAQQueue CreateQueue(string queueName, EDBConnection connection)
    {
        var queue = new EDBAQQueue(queueName, connection);
        queue.MessageType =
EDBAQMessageType.Udt;
        queue.EnqueueOptions.Visibility = EDBAQVisibility.ON_COMMIT;
        queue.UdtTypeName = "myxml";

        return queue;
    }

    // Enqueues the
payload
    // If the enqueuing was successfull, message variable receives the queue message and the function
returns true
    // otherwise message is null and the function returns
false
    private static bool TryEnqueueMessage<T>(EDBAQQueue queue, T payload, out EDBAQMessage
message)
    {
        using EDBTransaction transaction =
queue.Connection.BeginTransaction();

        try
        {
            message = new EDBAQMessage() { Payload = payload };
            queue.Enqueue(message);

transaction.Commit();

            return true;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error while enqueing message:
{ex.Message}");

transaction?.Rollback();
```

```
            message = null;
            return false;
        }
    }
}
```

Dequeueing 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 `EDBAQQueue.Dequeue()` method.

> **Note**
>
> The following code shows how to use the `EDBAQQueue.Dequeue` method. A queue is retrieved by its name, and a attempt is made to dequeue a message.
>
> If a `PostgresException` with `SqlState` set to `P0002` is raised, then the queue is empty or the wait time (set with `queue.DequeueOptions.Wait`) has expired, and the code gracefully returns a `null` message.

```
// .NET Framework 4.7.2
using
System;
using
System.Threading.Tasks;
using EnterpriseDB.EDBClient;

namespace EnterpriseDB
{
    internal static class Program
    {
        // Sample message
payload
        class MyXML
        {
            public string Value { get; set; }
        }

        public static async Task Main(string[] args)
        {
            // not for production, move connection string to app
settings
            string connectionString = "Server=127.0.0.1;Port=5444;User
Id=enterprisedb;Password=edb;Database=edb";

            // Note registration of MyXml type
            var dataSourceBuilder = new EDBDataSourceBuilder(connectionString);
            dataSourceBuilder
                .MapComposite<MyXML>("enterprisedb.myxml");

            using (var dataSource = dataSourceBuilder.Build())
            using (var connection = await dataSource.OpenConnectionAsync())
            using (var queue = CreateQueue("MSG_QUEUE", connection))
            {
                // Dequeue 5
messages
```

```
                int messagesToDequeue = 5;
                for (int i = 0; i < messagesToDequeue;
i++)
                {
                    if (TryDequeueMessage(queue, out var message))
                    {
                        Console.WriteLine($"Message {message.MessageId}
dequeued");

                        if (message?.Payload is MyXML myXML)
                        {
                            Console.WriteLine($"MyXML Message received:
{myXML.Value}");
                        }
                        else
                        {
                            Console.WriteLine($"Other message received");
                        }
                    }
                    else
                    {
                        Console.WriteLine($"No
message");
                    }
                }
            }
        }

        // Creates and returns a queue ready for use in our
sample
        private static EDBAQQueue CreateQueue(string queueName, EDBConnection connection)
        {
            var queue = new EDBAQQueue(queueName, connection);
            queue.MessageType =
EDBAQMessageType.Udt;
            queue.DequeueOptions.Navigation =
EDBAQNavigationMode.FIRST_MESSAGE;
            queue.DequeueOptions.Visibility = EDBAQVisibility.ON_COMMIT;
            queue.DequeueOptions.Wait = 1; // wait for 1
seconds
            queue.UdtTypeName = "myxml";

            return queue;
        }

        // Dequeus a
payload
        // If the dequeuing was successfull, message variable receives the queue message and the
function returns true
        // otherwise message is null and the function returns
false
        private static bool TryDequeueMessage(EDBAQQueue queue, out EDBAQMessage message)
        {
            using (EDBTransaction transaction =
queue.Connection.BeginTransaction())
            {
                try
                {
                    message = queue.Dequeue();

transaction.Commit();

                    return true;
```

```
                }
                catch (PostgresException pgException) when (pgException.SqlState ==
"P0002")
                {
                    // Queue empty or time
out

transaction.Commit();

                    message = null;
                    return false;
                }
                catch (Exception ex)
                {
                    Console.WriteLine($"Error while dequeuing message:
{ex.Message}");

transaction?.Rollback();

                    message = null;
                    return false;
                }
            }
        }
    }
}

// .NET8
using EnterpriseDB.EDBClient;

namespace EnterpriseDB;
internal static class Program
{
    // Sample message
payload
    class MyXML
    {
        public string Value { get; set; }
    }

    public static async Task Main(string[] args)
    {
        // not for production, move connection string to app
settings
        string connectionString = "Server=127.0.0.1;Port=5444;User
Id=enterprisedb;Password=edb;Database=edb";

        // Note registration of MyXml type
        var dataSourceBuilder = new EDBDataSourceBuilder(connectionString);
        dataSourceBuilder
            .MapComposite<MyXML>("enterprisedb.myxml");

        await using var dataSource = dataSourceBuilder.Build();
        await using var connection = await dataSource.OpenConnectionAsync();
        using var queue = CreateQueue("MSG_QUEUE", connection);

        // Dequeue 5
messages
        int messagesToDequeue = 5;
        for (int i = 0; i < messagesToDequeue;
i++)
        {
```

```
            if (TryDequeueMessage(queue, out var message))
            {
                Console.WriteLine($"Message {message.MessageId}
dequeued");

                if (message?.Payload is MyXML myXML)
                {
                    Console.WriteLine($"MyXML Message received:
{myXML.Value}");
                }
                else
                {
                    Console.WriteLine($"Other message received");
                }
            }
            else
            {
                Console.WriteLine($"No
message");
            }
        }
    }

    // Creates and returns a queue ready for use in our
sample
    private static EDBAQQueue CreateQueue(string queueName, EDBConnection connection)
    {
        var queue = new EDBAQQueue(queueName, connection);
        queue.MessageType =
EDBAQMessageType.Udt;
        queue.DequeueOptions.Navigation =
EDBAQNavigationMode.FIRST_MESSAGE;
        queue.DequeueOptions.Visibility = EDBAQVisibility.ON_COMMIT;
        queue.DequeueOptions.Wait = 1; // wait for 1
seconds
        queue.UdtTypeName = "myxml";

        return queue;
    }

    // Dequeus a
payload
    // If the dequeuing was successfull, message variable receives the queue message and the function
returns true
    // otherwise message is null and the function returns
false
    private static bool TryDequeueMessage(EDBAQQueue queue, out EDBAQMessage message)
    {
        using EDBTransaction transaction =
queue.Connection.BeginTransaction();

        try
        {
            message = queue.Dequeue();

transaction.Commit();

            return true;
        }
        catch (PostgresException pgException) when (pgException.SqlState ==
"P0002")
        {
```

```
            // Queue empty or time
out

transaction.Commit();

            message = null;
            return false;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error while dequeuing message:
{ex.Message}");

transaction?.Rollback();

            message = null;
            return false;
        }
    }
}
```

EDBAQ classes

The following EDBAQ classes are used in this application.

EDBAQDequeueMode

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

Value	Description
Browse	Reads 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	Set from `EDBAQDequeueMode`. It represents the locking behavior linked with the dequeue option.
Navigation	Set from `EDBAQNavigationMode`. It represents the position of the message to fetch.
Visibility	Set from `EDBAQVisibility`. It represents whether the new message is dequeued 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's a Boolean expression.

Property	Description
Transformation	The transformation to apply 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	Set from `EDBAQVisibility`. It represents whether the new message is enqueued as part of the current transaction.
RelativeMsgid	The relative message identifier.
SequenceDeviation	The sequence when to dequeue the message.
Transformation	The transformation to apply before enqueuing the message.
DeliveryMode	The delivery mode of the enqueued message.

EDBAQMessage

The `EDBAQMessage` class lists a message to enqueue/dequeue.

Property	Description
Payload	The actual message to queue.
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 after which the message is available for dequeuing, in seconds.
Expiration	The duration for which the message is available for dequeuing, in seconds.
Correlation	The correlation identifier.
Attempts	The number of attempts taken to dequeue the message.
RecipientList	The recipients list that overthrows the default queue subscribers.
ExceptionQueue	The name of the queue to move the unprocessed messages to.
EnqueueTime	The time when the message was enqueued.
State	The state of the message while dequeued.
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 isn't reached.

EDBAQMessageType

The `EDBAQMessageType` class represents the types for payload.

Value	Description
Raw	The raw message type. Note: Currently, this payload type isn't supported.
UDT	The user-defined type message.
XML	The XML type message. Note: Currently, this payload type isn't 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 use.
Name	The name of the queue.
MessageType	The message type that's enqueued/dequeued from this queue, for example `EDBAQMessageType.Udt`.
UdtTypeName	The user-defined type name of the message type.
EnqueueOptions	The enqueue options to use.
DequeuOptions	The dequeue options to use.
MessageProperties	The message properties to use.

EDBAQVisibility

The `EDBAQVisibility` class represents the visibility options available.

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

> **Note**
>
> - To review the default options for these parameters, see DBMS_AQ.
> - EDB advanced queueing functionality uses user-defined types for calling enqueue/dequeue operations. `Server Compatibility Mode=NoTypeLoading` can't be used with advanced queueing because `NoTypeLoading` doesn't load any user-defined types.

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 executing the `OPEN FOR` statement using the cursor variable. A cursor variable isn't tied to a particular query like a static cursor. You can open the same cursor variable a number of times with the `OPEN FOR` statement containing different queries each time. A new result set is created for that query and made available by way of the cursor variable. You can declare a cursor variable in two ways:

- 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

This sample code creates a stored procedure called `refcur_inout_callee`. It specifies the data type of the ref cursor being passed as an OUT parameter. To create the sample procedure, invoke EDB-PSQL and connect to the EDB Postgres Advanced Server host database. Enter the following SPL code at the command line:

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

This 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)
        {
            var strConnectionString
=
            ConfigurationManager.AppSettings["DB_CONN_STRING"];
            try
```

```
            {
                await using var dataSource =
EDBDataSource.Create(ConnectionString);
                var conn = await
dataSource.OpenConnectionAsync();
                await using var tran = await
connection.BeginTransactionAsync();
                using var 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.Output, false, 2, 2,
                    System.Data.DataRowVersion.Current,
null));
                await command.PrepareAsync();
                command.Parameters[0].Value = null;
                await command.ExecuteNonQueryAsync();
                var cursorName =
command.Parameters[0].Value.ToString();
                command.CommandText = "fetch all in \"" + cursorName +
"\"";
                command.CommandType =
CommandType.Text;
                await using var reader =
                    await command.ExecuteReaderAsync(CommandBehavior.SequentialAccess);
                var fc =
reader.FieldCount;
                while (await
reader.ReadAsync())
                {
                    for (int i = 0; i < fc;
i++)
                    {
                        Console.WriteLine(reader.GetString(i));
                    }
                }
                await
reader.CloseAsync();
                await tran.CommitAsync();
                await conn.CloseAsync();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message.ToString());
            }
        }
    }
}
```

This .NET code snippet displays the result on the console:

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

You must bind the `EDBDbType.RefCursor` type in `EDBParameter()` if you're using a ref cursor parameter.

15 Using plugins

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

- GeoJSON
- Json.NET
- NetTopologySuite
- NodaTime

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

GeoJSON

If you're using the GeoJSON plugin on .NET Standard 2.0, the data provider installation paths are:

- `C:\Program Files\edb\dotnet\plugins\GeoJSON\netstandard2.0`
- `C:\Program Files\edb\dotnet\plugins\GeoJSON\net472`
- `C:\Program Files\edb\dotnet\plugins\GeoJSON\net48`
- `C:\Program Files\edb\dotnet\plugins\GeoJSON\net481`

The following shared library files are required:

- `EnterpriseDB.EDBClient.GeoJSON.dll`

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

Json.NET

If you're using the Json.NET plugin on .NET Standard 2.0, the data provider installation paths are:

- `C:\Program Files\edb\dotnet\plugins\Json.NET\netstandard2.0`
- `C:\Program Files\edb\dotnet\plugins\Json.NET\net472`
- `C:\Program Files\edb\dotnet\plugins\Json.NET\net48`
- `C:\Program Files\edb\dotnet\plugins\Json.NET\net481`

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.

NetTopologySuite

If you're using the NetTopologySuite plugin on .Net Standard 2.0, the data provider installation paths are:

- `C:\Program Files\edb\dotnet\plugins\NetTopologySuite\netstandard2.0`
- `C:\Program Files\edb\dotnet\plugins\NetTopologySuite\net472`
- `C:\Program Files\edb\dotnet\plugins\NetTopologySuite\net48`
- `C:\Program Files\edb\dotnet\plugins\NetTopologySuite\net481`

The following shared library files are required:

- `EnterpriseDB.EDBClient.NetTopologySuite.dll`

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

NodaTime

If you're using the NodaTime plugin on .Net Standard 2.0, the data provider installation paths are:

- `C:\Program Files\edb\dotnet\plugins\NodaTime\netstandard2.0`
- `C:\Program Files\edb\dotnet\plugins\NodaTime\net472`
- `C:\Program Files\edb\dotnet\plugins\NodaTime\net48`
- `C:\Program Files\edb\dotnet\plugins\NodaTime\net481`

The following shared library files are required:

- `EnterpriseDB.EDBClient.NodaTime.dll`

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

16 Using object types in .NET

The SQL `CREATE TYPE` command creates a user-defined object type, which is stored in the EDB Postgres Advanced Server database. You can then reference these user-defined types in SPL procedures, SPL functions, and .NET programs.

Create the basic object type with the `CREATE TYPE AS OBJECT` command. Optionally, use the `CREATE TYPE BODY` command.

Using an object type

To use an object type, you must first create the object type in the EDB Postgres Advanced Server database. Object type `addr_object_type` defines the attributes of an address:

```
CREATE OR REPLACE TYPE addr_object_type AS
OBJECT
(
    street
VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip
NUMBER(5)
);
```

Object type `emp_obj_typ` defines the attributes of an employee. One of these attributes is object type `ADDR_OBJECT_TYPE`, as previously described. The object type body contains a method that displays the employee information:

```
CREATE OR REPLACE TYPE emp_obj_typ AS
OBJECT
(
    empno           NUMBER(4),
    ename           VARCHAR2(20),
    addr            ADDR_OBJECT_TYPE,
    MEMBER PROCEDURE display_emp(SELF IN OUT
emp_obj_typ)
);

CREATE OR REPLACE TYPE BODY emp_obj_typ
AS
  MEMBER PROCEDURE display_emp (SELF IN OUT
emp_obj_typ)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Employee No   : ' ||
SELF.empno);
    DBMS_OUTPUT.PUT_LINE('Name           : ' ||
SELF.ename);
    DBMS_OUTPUT.PUT_LINE('Street         : ' ||
SELF.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', '
||
      SELF.addr.state || ' ' ||
LPAD(SELF.addr.zip,5,'0'));
  END;
END;
```

This example is a complete .NET program that uses these user-defined object types:

```
using EnterpriseDB.EDBClient;
using
System.Data.Common;
namespace TypesTest
{
    internal class Program
    {
        static async Task Main(string[] args)
        {
            var connString = "Server=localhost;Port=5444;database=edb;User
ID=enterprisedb;password=edb;";
            var dataSourceBuilder = new EDBDataSourceBuilder(connString);
            dataSourceBuilder.MapComposite<addr_object_type>("enterprisedb.addr_object_type");
            dataSourceBuilder.MapComposite<emp_obj_typ>("enterprisedb.emp_obj_typ");
            await using var dataSource = dataSourceBuilder.Build();
            await using var conn = await dataSource.OpenConnectionAsync();
            try
            {
                var address = new addr_object_type()
                {
                    street = "123 MAIN
STREET",
                    city = "EDISON",
                    state = "NJ",
                    zip =
8817
                };
                var emp = new
emp_obj_typ()
                {
                    empno = 9001,
                    ename = "JONES",
                    addr = address
                };
                await using (var cmd = new EDBCommand("emp_obj_typ.display_emp",
conn))
                {
                    cmd.CommandType =
System.Data.CommandType.StoredProcedure;
                    EDBCommandBuilder.DeriveParameters(cmd);
                    cmd.Parameters[0].Value =
emp;

cmd.Prepare();

cmd.ExecuteNonQuery();
                    var empOut =
(emp_obj_typ?)cmd.Parameters[0].Value;
                    Console.WriteLine("Emp No: " +
empOut.empno);
                    Console.WriteLine("Emp Name: " +
empOut.ename);
                    Console.WriteLine("Emp Address Street: " +
empOut.addr.street);
                    Console.WriteLine("Emp Address City: " +
empOut.addr.city);
                    Console.WriteLine("Emp Address State: " +
empOut.addr.state);
                    Console.WriteLine("Emp Address Zip: " +
empOut.addr.zip);
                    Console.WriteLine("Emp No: " +
empOut.empno);
                }
            }
```

```
            catch (EDBException
exp)
            {
                Console.WriteLine(exp.Message.ToString());
            }
            finally
            {
                conn.Close();
            }
        }
    }
public class
addr_object_type
    {
        public string?
street;
        public string? city;
        public string? state;
        public decimal
zip;
    }
    public class
emp_obj_typ
    {
        public decimal empno;
        public string? ename;
        public addr_object_type?
addr;
    }
}
```

The following .NET types are defined to map to the types in EDB Postgres Advanced Server:

```
public class
addr_object_type
{
public string?
street;
public string? city;
public string? state;
public decimal
zip;
}

public class
emp_obj_typ
{
    public decimal empno;
    public string? ename;
    public addr_object_type?
addr;
}
```

A call to `EDBDataSourceBuilder.MapComposite` maps the .NET type to the EDB Postgres Advanced Server types:

```
dataSourceBuilder.MapComposite<addr_object_type>("enterprisedb.addr_object_type");
dataSourceBuilder.MapComposite<emp_obj_typ>("enterprisedb.emp_obj_typ");
```

A call to `EDBCommandBuilder.DeriveParameters()` gets parameter information for a stored procedure. This allows you to just set the

parameter values and call the stored procedure:

```
EDBCommandBuilder.DeriveParameters(cmd);
```

Set the value of the parameter by creating an object of the .NET type and assigning it to the `Value` property of the parameter:

```
addr_object_type address = new addr_object_type()
{
    street = "123 MAIN
STREET",
    city = "EDISON",
    state = "NJ",
    zip =
8817
};

emp_obj_typ emp = new
emp_obj_typ()
{
    empno = 9001,
    ename = "JONES",
    addr = address
};
cmd.Parameters[0].Value =
emp;
```

A call to `cmd.ExecuteNonQuery()` executes the call to the `display_emp()` method:

```
cmd.ExecuteNonQuery();
```

17 Using nested tables

EDB Postgres Advanced Server supports nested table collection types created with `CREATE TYPE ... AS TABLE OF` statements. The EDB .NET Connector supports output parameters declared as nested tables out of the box, whether free-standing types or declared inside packages.

Nested table types mapping

Nested table types are mapped to `List<object>` s in C#, as it is preferred over `ArrayList` . These lists contain as many elements as the nested table type's rows. The nested table items are translated to be compatible with the C# application using the following rules:

- The connector resolves all nested table rows into a `List<object>` in C# while maintaining and converting each column's underlying type. For example, a `[text1, text2, num1]` row will be resolved as a `[string, string, decimal]` item in the list.
- If the nested type `IS TABLE OF` a domain type (int, varchar, decimal, etc.), all the rows will be their C# counterpart according to the Supported Types and their Mappings.
- If the nested type `IS TABLE OF` a record or composite type **not mapped** to a C# class, all rows become a nested List containing as many elements as the record or composite fields, with proper type translation.
- If the nested type `IS TABLE OF` a record or composite type **mapped** to a C# class (for example, `MyComposite`), all rows will be `MyComposite` instances.

Example: Retrieving nested table output parameter

This program:

- Creates a package with a nested `emp_tbl_typ` table type of `emp_rec_typ` . This package has a stored procedure that fills the nested table output parameter.
- Maps the nested table type to a C# class via `MapComposite<>` .
- Executes and displays the results.

> **Note**
>
> Always provide type names in lowercase.

Program example

Create an empty console program and paste the following code.

```
internal static class Program
{
    const string ConnectionString =
"your_connection_string";

    // Composite type, will be mapped to the nested table
type
```

```
    // This will work if field types are convertible from database
types
    public class
Employee
    {
        [PgName("empno")]
        public decimal
Number;
        [PgName("ename")]
        public string? Name;
    }

    public static void Sample_NestedTableTypes(string
ConnectionString)
    {
        var dataSourceBuilder = new EDBDataSourceBuilder(ConnectionString);
        dataSourceBuilder.MapComposite<Employee>("pkgextendtest.emp_rec_typ");

        using (var dataSource = dataSourceBuilder.Build())
        {
            using (var connection = dataSource.OpenConnection())
            {

                try
                {
                    CreatePackage(connection);

                    var commandText =
"pkgExtendTest.nestedTableExtendTest";
                    var cstmt = new EDBCommand(commandText, connection);
                    cstmt.CommandType =
CommandType.StoredProcedure;

                    var tableOfParam = cstmt.Parameters.Add(new EDBParameter()
                    {
                        Direction = ParameterDirection.Output,
                        DataTypeName = "pkgextendtest.emp_tbl_typ"
                    });

                    cstmt.Prepare();
                    cstmt.ExecuteNonQuery();

                    List<object>? employees = tableOfParam.Value as List<object>;
                    if (employees == null)
                    {
                        Console.WriteLine($"No employee
found");
                        return;
                    }

                    foreach (var employeeRecord in
employees)
                    {
                        var employee = employeeRecord as
Employee;
                        if (employee != null)
                        {
                            Console.WriteLine($"Employee {employee.Number}:
{employee.Name}");
                        }
                    }
                }
```

```
                catch (Exception ex)
                {
                    Console.WriteLine($"Error: {ex.Message}");
                }
                finally
                {
                    Cleanup(connection);
                }
            }
        }
    }

    // helper methods to create package and cleaning up
    static void CreatePackage(EDBConnection connection)
    {
        var createPackage =
"                CREATE OR REPLACE PACKAGE pkgExtendTest IS  \n"
+
"                   TYPE emp_rec_typ IS RECORD (  \n"
+
"                      empno  NUMBER(4),  \n" +
"                      ename       VARCHAR2(10)  \n" +
"                     ); \n" +
"                   TYPE emp_tbl_typ IS TABLE OF emp_rec_typ;  \n"
+
"                   PROCEDURE nestedTableExtendTest(emp_tbl OUT emp_tbl_typ); \n"
+
"                END pkgExtendTest;
\n";
        using (var com = new EDBCommand(createPackage, connection) { CommandType = CommandType.Text
})
        {

com.ExecuteNonQuery();
        }

        var createPackageBody =
"                    CREATE OR REPLACE PACKAGE BODY pkgExtendTest IS \n"
+
"                    PROCEDURE nestedTableExtendTest(emp_tbl OUT emp_tbl_typ) IS \n"
+
"                      DECLARE  \n" +
"                      CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10 order by empno;
\n" +
"                      i  INTEGER := 0;  \n"
+
"                    BEGIN \n" +
"                       emp_tbl := emp_tbl_typ();  \n"
+
"                       FOR r_emp IN emp_cur LOOP  \n"
+
"                          i := i + 1;  \n"
+
"                          emp_tbl.EXTEND;  \n" +
"                          emp_tbl(i) := r_emp;  \n"
+
"                       END LOOP;  \n"
+
"                    END nestedTableExtendTest; \n"
+
"                    END pkgExtendTest;
\n";
        using (var com = new EDBCommand(createPackageBody, connection) { CommandType = CommandType.Text
})
```

```
        {

com.ExecuteNonQuery();
        }

        connection.ReloadTypes();
    }

    static void Cleanup(EDBConnection connection)
    {
        var dropPackageBody = "DROP PACKAGE BODY
pkgExtendTest";
        var dropPackage = "DROP PACKAGE
pkgExtendTest";

        using (var com = new EDBCommand(dropPackageBody, connection) { CommandType = CommandType.Text
})
        {

com.ExecuteNonQuery();
        }
        using (var com = new EDBCommand(dropPackage, connection) { CommandType = CommandType.Text
})
        {

com.ExecuteNonQuery();
        }
    }
}
```

The output should look like this:

```
Employee 7499: ALLEN
Employee 7521: WARD
Employee 7566: JONES
Employee 7654: MARTIN
Employee 7698: BLAKE
Employee 7782: CLARK
Employee 7788: SCOTT
Employee 7839: KING
Employee 7844: TURNER
Employee 7876: ADAMS
```

18 Scram compatibility

The EDB .NET driver provides SCRAM-SHA-256 support for EDB Postgres Advanced Server version 10 and later. This support is available in EDB .NET 4.0.2.1 release and later.

19 EDB .NET Connector logging

EDB .NET Connector supports the use of logging to help resolve issues with the .NET Connector when used in your application. EDB .NET Connector supports logging using the standard .NET `Microsoft.Extensions.Logging` package. For more information about logging in .Net, see Logging in C# and .NET in the Microsoft documentation.

> **Note**
>
> For versions earlier than 7.x, EDB .NET Connector had its own, custom logging API.

Console logging provider

The .NET logging API works with a variety of built-in and third-party logging providers. The console logging provider logs output to the console.

Console logging with EDBDataSource

Create a `Microsoft.Extensions.Logging.LoggerFactory` and configure an `EDBDataSource` with it. Any use of connections opened through this data source log using this logger factory.

```
var loggerFactory = LoggerFactory.Create(builder => builder.AddSimpleConsole());

var dataSourceBuilder = new EDBDataSourceBuilder(connectionString);
dataSourceBuilder.UseLoggerFactory(loggerFactory);
await using var dataSource = dataSourceBuilder.Build();

await using var connection = await dataSource.OpenConnectionAsync();
await using var command = new EDBCommand("SELECT 1", connection);
_ = await command.ExecuteScalarAsync();
```

Console logging without EDBDataSource

Create a `Microsoft.Extensions.Logging.LoggerFactory` and configure EDB .NET Connector's logger factory globally using `EDBLoggingConfiguration.InitializeLogging`. Configure it at the start of your program, before using any other EDB .NET Connector API.

```
var loggerFactory = LoggerFactory.Create(builder => builder.AddSimpleConsole());
EDBLoggingConfiguration.InitializeLogging(loggerFactory);

await using var conn = new EDBConnection(connectionString);
await conn.OpenAsync();
await using var command = new EDBCommand("SELECT 1", conn);
_ = await command.ExecuteScalarAsync();
```

Log levels

The following log levels are available:

- Trace
- Debug
- Information
- Warning
- Error
- Fatal

This example shows how to change the log level to `Trace`:

```
var loggerFactory = LoggerFactory.Create(builder => builder
.SetMinimumLevel(LogLevel.Trace
.AddSimpleConsole()
);
```

Formatting the log output

This example shows how to format your log output. Create a `LoggerFactory` to restrict each log message to a single line and add a date and time to the log:

```
var loggerFactory = LoggerFactory.Create(builder =>
builder
.SetMinimumLevel(LogLevel.Trace)
.AddSimpleConsole(
    options =>
    {
        options.SingleLine = true;
        options.TimestampFormat = "yyyy/MM/dd HH:mm:ss
";
    }
    ));
```

20 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.
- To find the Npgsql API version that was included with a specific EDB .NET release, see the EDB .NET release notes. The release notes specify the upstream Npgsql version that was merged. The version information is important because the available API features can vary between versions.