



EDB

EDB .NET Connector

リリース *6.0.2.1*

EnterpriseDB Corporation

2022 年 09 月 26 日

目次

第 1 章	Release notes	3
第 2 章	Product compatability	5
第 3 章	The EDB .NET Connector overview	7
3.1	.NET クラス階層	7
第 4 章	Installing and configuring the .NET Connector	9
4.1	.NET コネクタのインストール	9
4.2	.NET コネクタの構成	15
4.2.1	ライブラリファイルの参照	15
4.2.2	.NET フレームワークのセットアップ	16
4.2.3	.NET 6.0	16
4.2.4	.NET 5.0	16
4.2.5	.NET Core 3.1	16
4.2.6	.NET Standard 2.0	17
4.2.7	.NET Standard 2.1	17
4.2.8	.NET EntityFramework コア	18
第 5 章	Using the .NET Connector	21
第 6 章	Opening a database connection	23
6.1	接続文字列パラメーター	23
6.2	例：ASP.NET を使用してデータベース接続を開く	25
6.3	例：コンソールアプリケーションからデータベース接続を開く	26
6.4	例：Windows フォームアプリケーションからデータベース接続を開く	27
第 7 章	Retrieving database records	31
7.1	単一のデータベースレコードを取得する	32
第 8 章	Parameterized queries	35
第 9 章	Inserting records in a database	37
第 10 章	Deleting records in a database	39
第 11 章	Using SPL stored procedures in your .NET application	41
11.1	例：パラメーターなしでストアードプロシージャを実行する	41
11.1.1	EDBCommand オブジェクトを使用してストアードプロシージャを実行する	42
11.2	例：IN パラメーターを使用したストアードプロシージャの実行	43
11.2.1	入力値をストアードプロシージャに渡す	44
11.3	例：IN、OUT、および INOUT パラメーターを使用したストアードプロシージャの実行	46
11.3.1	ストアードプロシージャの作成	47

11.3.2	ストアードプロシージャからの出力値の受信	47
第 12 章	Using advanced queueing	51
12.1	メッセージのエンキューまたはデキュー	51
12.1.1	サーバーサイドのセットアップ	51
12.1.2	クライアントサイドの例	52
12.2	EDBAQ クラス	59
第 13 章	Using a ref cursor in a .NET application	63
13.1	ストアードプロシージャの作成	63
第 14 章	Using plugins	67
14.1	GeoJSON	67
14.2	Json.NET	67
14.3	LegacyPostGIS	68
14.4	NetTopologySuite	68
14.5	NodaTime	68
14.6	RawPostGIS	68
第 15 章	Using object types in .NET	69
15.1	オブジェクトタイプの使用	69
第 16 章	Scram compatibility	75
第 17 章	EDB .NET Connector logging	77
17.1	標準エラーへのコンソールロギング	77
17.2	カスタムロガーを書く	77
第 18 章	API reference	81

EDB Postgres Advanced Server とともに配布される EDB .NET Connector は、.NET クライアントアプリケーションと EDB Postgres Advanced Server データベースサーバー間の接続を提供します。次のことができます。

EDB Postgres Advanced Server のインスタンスに接続します。

- EDB Postgres Advanced Server データベースから情報を取得します。
- EDB Postgres Advanced Server データベースに保存されている情報を更新します。

これらの例を理解するには、C#と.NET に関する確かな知識が必要です。EDB .NET コネクタ機能は、Npgsql オープンソースプロジェクトのコア機能に基づいて構築されています。詳細については、

[Npgsql User Guide](#) を参照してください。

[release_notes](#) [requirements_overview](#) [the_advanced_server_net_connector_overview](#) [installing_and_configuring_the_net_connector](#) [using_the_net_connector](#) [opening_a_database_connection](#)
[retrieving_database_records](#) [parameterized_queries](#) [inserting_records_in_a_database](#) [deleting_records_in_a_database](#) [using_spl_stored_procedures_in_your_net_application](#) [using_advanced_queueing](#)
[using_a_ref_cursor_in_a_net_application](#) [using_plugins](#) [using_object_types](#) [scram_compatibility](#) [advanced_server_net_connector_logging](#) [api_reference](#) [conclusion](#)

第1章 Release notes

EDB .NET Connector は、.NET クライアントアプリケーションと EDB Postgres Advanced Server データベースサーバー間の接続を提供します。

EDB .NET Connector 6.0.2.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream merge	コミュニティ.NET ドライバーバージョン 6.0.2 と統合されました。マージ更新の詳細については、< https://www.npgsql.org/doc/release-notes/6.0.html >を参照してください。
Enhancement	.NET 6.0 のサポートが追加されました。

第2章 Product compatability

これらは、EDB .NET Connector でサポートされているバージョンとプラットフォームです。

EDB .NET Connector は、 EDB Postgres Advanced Server バージョン 10 以降で認定されています。

EDB .NET Connector グラフィカルインストーラーは、次の Windows プラットフォームでサポートされています。

64 ビット Windows :

- Windows Server 2019
- Windows 10

32 ビット Windows :

- Windows 10

第3章 The EDB .NET Connector overview

EDB .NET Connector は、クライアントアプリケーションが EDB Postgres Advanced Server ホストに保存されているデータベースに接続できるようにする .NET データプロバイダーです。 .NET コネクタはデータに直接アクセスするため、クライアントアプリケーションは最適なパフォーマンス、幅広い機能、および EDB Postgres Advanced Server 機能へのアクセスが可能になります。

.NET コネクタは、次のフレームワークをサポートしています。

- .NET 6.0
- .NET 5.0
- .NET コア 3.1
- .NET Standard 2.0 および 2.1

3.1 .NET クラス階層

.NET クラス階層には、 EDB Postgres Advanced Server データベースへの接続を制御し、サーバーに保存されているデータを操作するために使用できるクラスが含まれています。以下は、最も一般的に使用されるオブジェクトクラスのほんの一部です。

EDBConnection

EDBConnection クラスは、 EDB Postgres Advanced Server への接続を表します。 EDBConnection オブジェクトには、 .NET クライアントに EDB Postgres Advanced Server データベースに接続する方法を指示する `ConnectionString` が含まれています。

EDBCommand

EDBCommand オブジェクトには、クライアントが EDB Postgres Advanced Server に対して実行する SQL コマンドが含まれています。 EDBCommand オブジェクトを実行する前に、それを EDBConnection オブジェクトにリンクする必要があります。

EDBDataReader

EDBDataReader オブジェクトは、 EDB Postgres Advanced Server の結果セットを読み取る方法を提供します。 EDBDataReader オブジェクトを使用して、一度に 1 行ずつ順方向にのみステップできます。

EDBDataAdapter

EDBDataAdapter オブジェクトは、結果セットを EDB Postgres Advanced Server データベースにリンクします。値を変更し、 EDBDataAdapter クラスを使用して、 EDB Postgres Advanced Server データベースに保存されているデータを更新できます。

第4章 Installing and configuring the .NET Connector

4.1 .NET コネクタのインストール

EDB .NET Connector インストーラー（[from the EDB website](#) 入手可能）を使用して、システムに.NET Connectorを追加できます。インストーラーをダウンロードしたら、インストーラーアイコンを右クリックし、コンテキストメニューから **Run As Administrator** を選択します。プロンプトが表示されたら、インストール言語を選択し、OK をクリックして Setup ウィンドウに進みます。

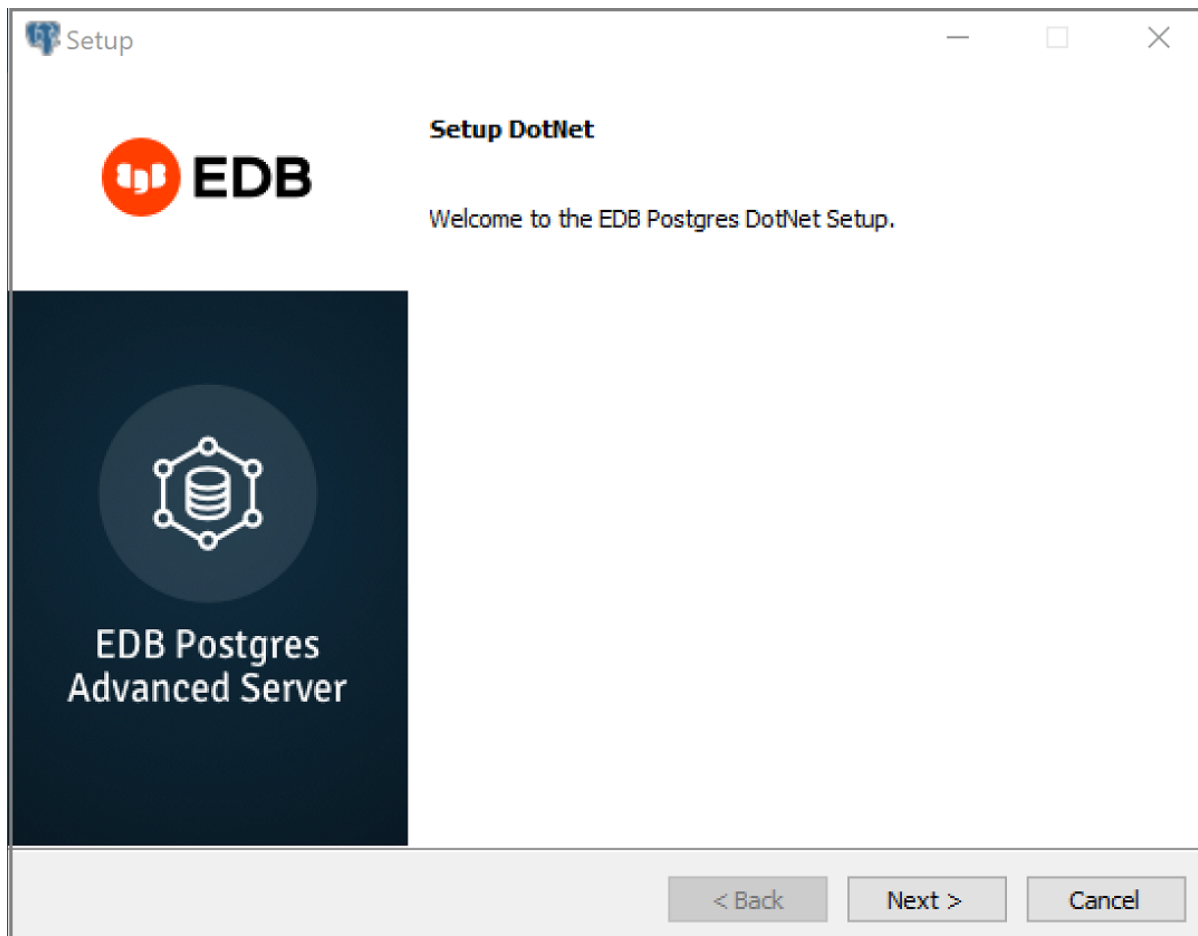


図 1: The .NET Connector Installation wizard

.NET コネクタのインストールウィザード

Next をクリックして続行します。

インストールダイアログ

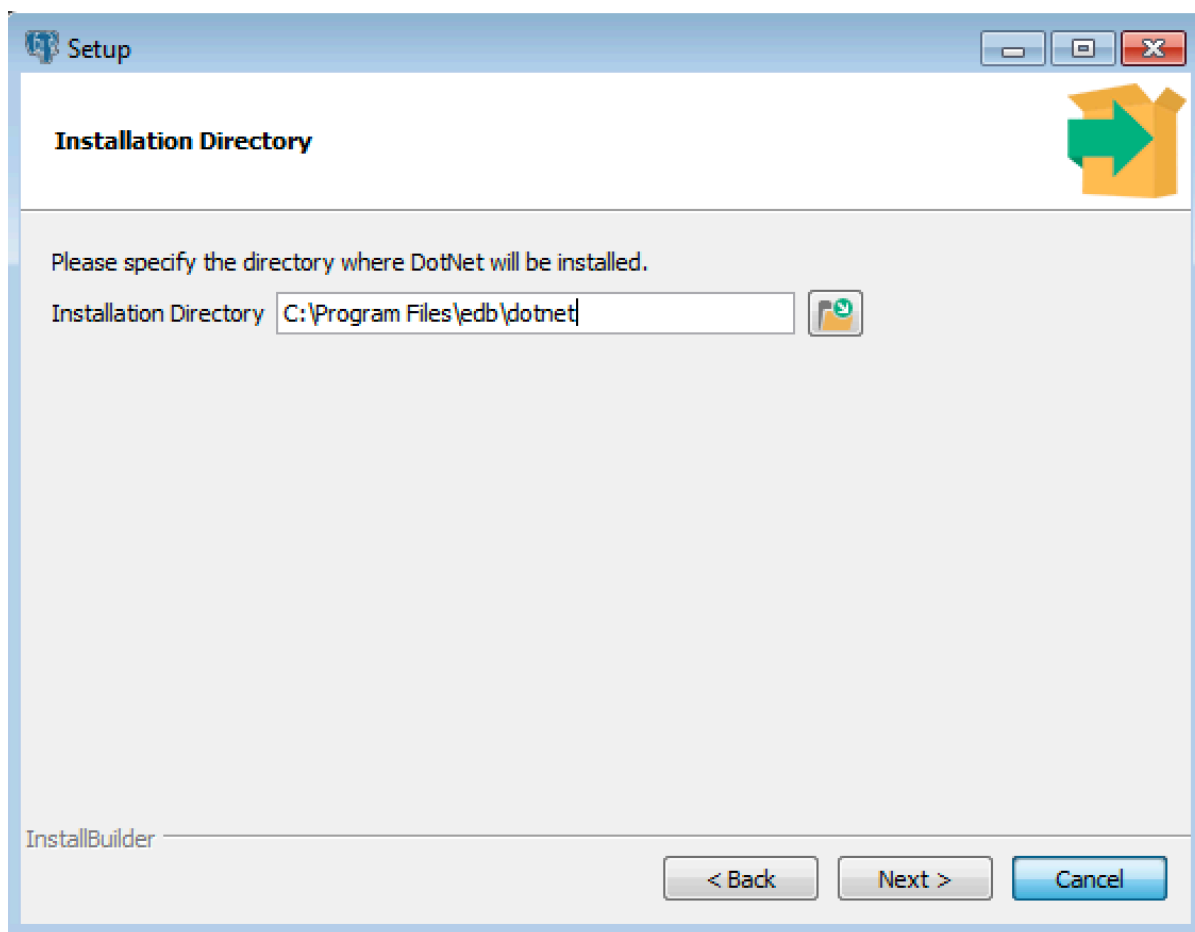


図 2: The Installation dialog

Installation Directory ダイアログを使用して、コネクタをインストールするディレクトリを指定し、Next をクリックして続行します。

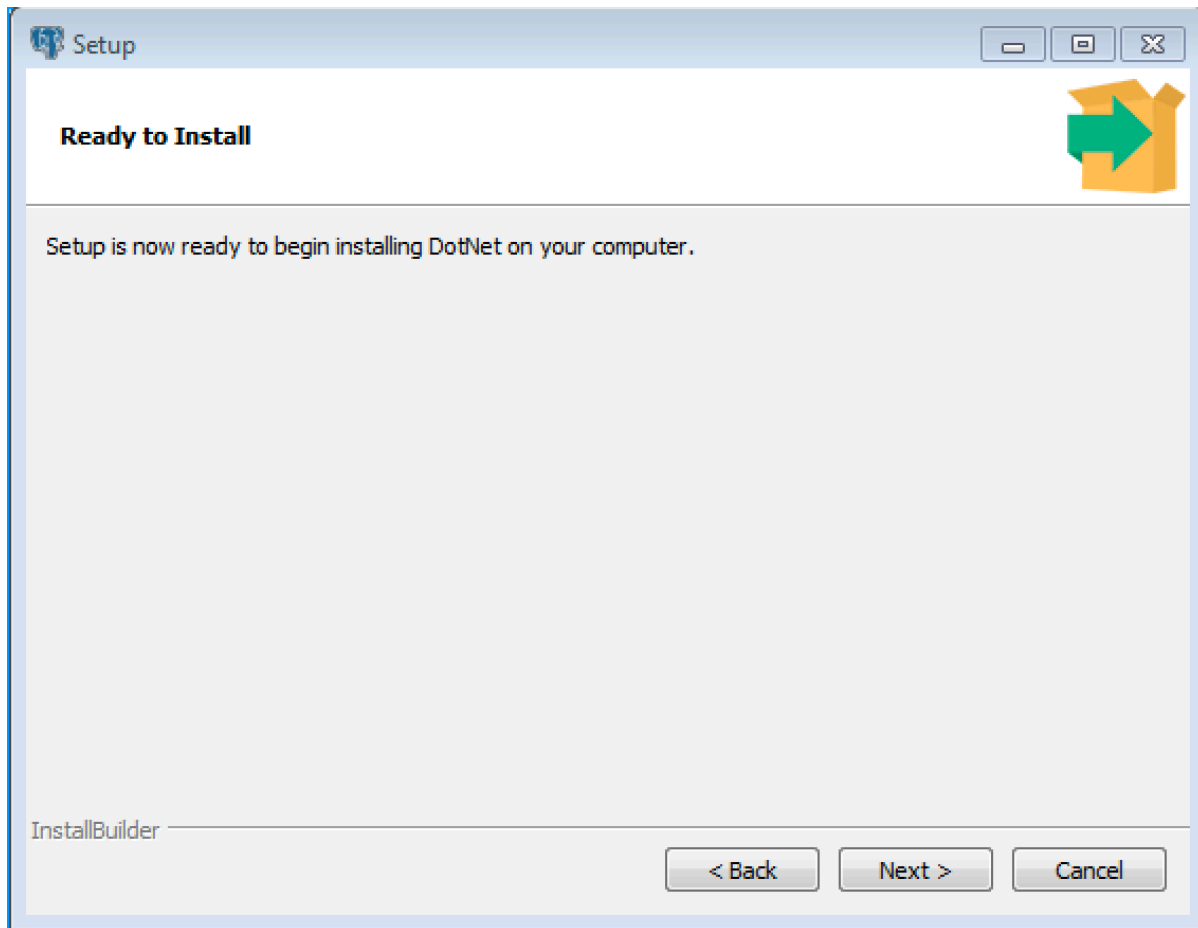


図 3: The Ready to Install dialog

インストールの準備完了ダイアログ

Ready to Install ダイアログで Next をクリックしてインストールを開始します。ポップアップダイアログは、インストールウィザードの進行状況を確認します。

インストールは完了です

ウィザードがセットアップを完了したことを通知したら、Finish ボタンをクリックしてダイアログを終了します。

StackBuilder Plus を使用して、既存の Advanced Server インストールにコネクタを追加または更新することもできます。StackBuilder Plus を開くには、Windows の Apps メニューから StackBuilder Plus を選択します。

StackBuilder Plus を起動する

StackBuilder Plus が開いたら、画面の指示に従います。

ツリーコントロールの Database Drivers ノードから EnterpriseDB.Net Connector オプションを選択します。

コネクタインストーラーの選択

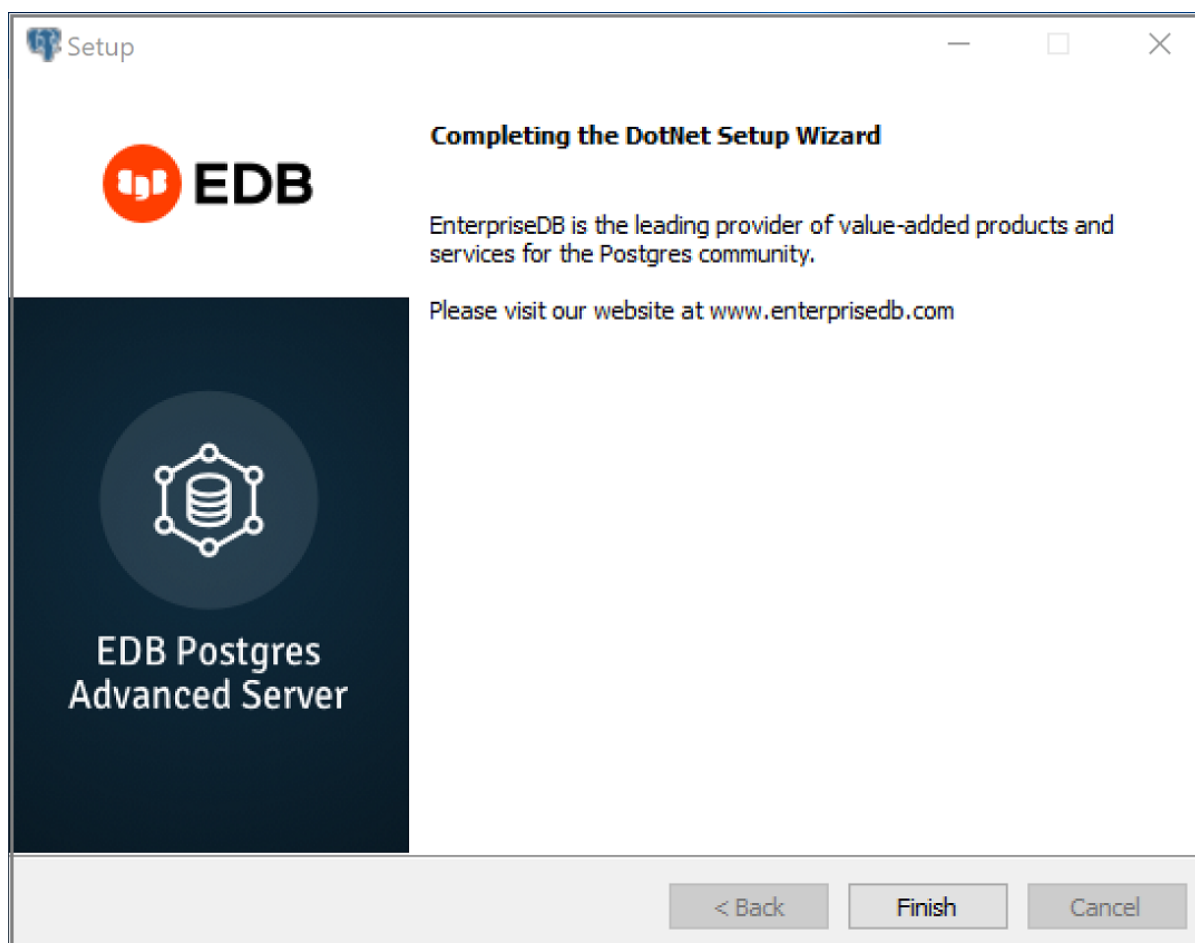


図 4: The installation is complete

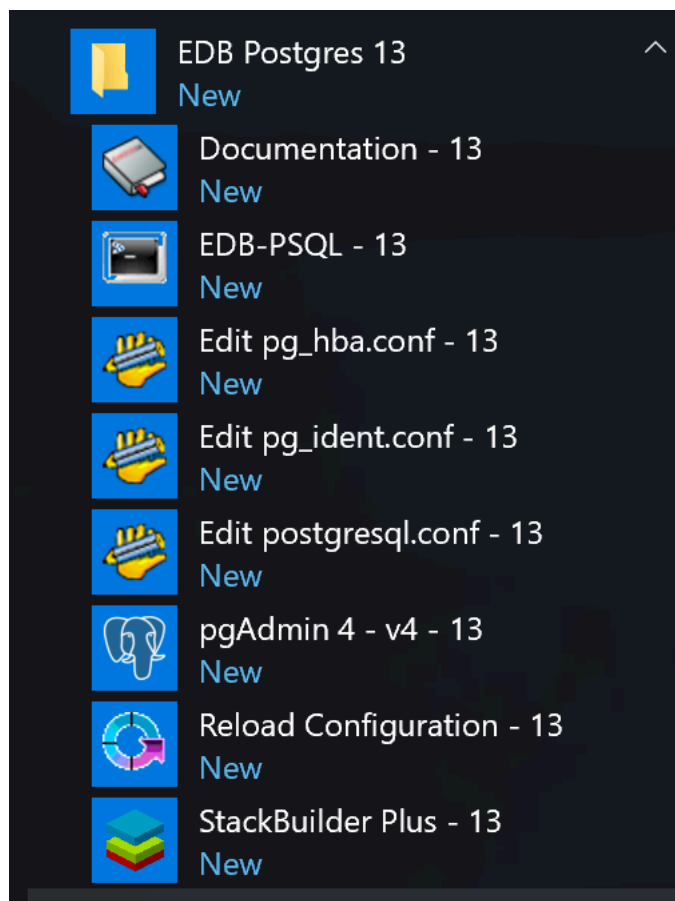


図 5: Starting StackBuilder Plus

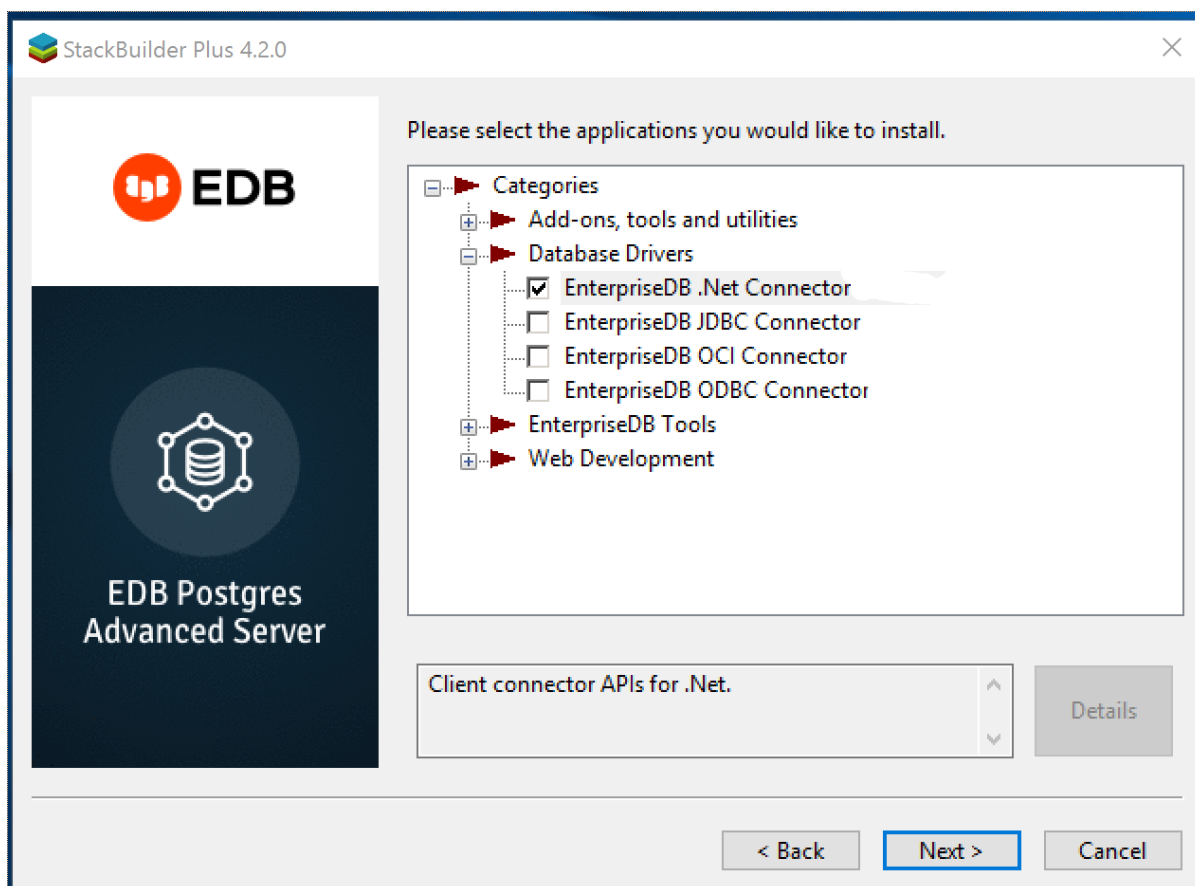


図 6: Selecting the Connectors installer

画面上のウィザードの指示に従って、EDB Connector のインストールを追加または更新します。

4.2 .NET コネクタの構成

.NET コネクタの構成については、次の環境固有のセクションを参照してください。

- **ライブラリファイルの参照。**
General configuration information はすべてのコンポーネントに適用されます。
- **.NET6.0** *.NET 6.0* で使用するための構成手順。
- **.NET5.0** *.NET 5.0* で使用するための構成手順。
- **.NETCore3.1** *.NET Core 3.1* で使用するための構成手順。
- **.NETStandard2.0** *.NET Standard 2.0* で使用するための構成手順。
- **.NETStandard2.1** *.NET Standard 2.1* で使用するための構成手順。
- **.NETEntityFrameworkCore** *.NET EntityFramework コア* で使用するための構成手順。

4.2.1 ライブラリファイルの参照

Microsoft Visual Studio でライブラリファイルを参照するには：

1. Solution Explorer でプロジェクトを選択します。
2. Project メニューから Add Reference を選択します。
3. Add Reference ダイアログボックスが開いたら、適切なライブラリファイルを参照して選択します。

オプションで、ライブラリファイルを指定した場所にコピーできます。

EDB .NET クラスを使用する前に、名前空間をプログラムにインポートする必要があります。名前空間をインポートすると、コンパイラーは名前空間内で使用可能なクラスを認識します。名前空間は次のとおりです。

`EnterpriseDB.EDBClient`

名前空間を含めるために使用する方法は、作成しているアプリケーションの種類によって異なります。たとえば、次のコマンドは名前空間を ASP.NET ページにインポートします。

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

名前空間を C#アプリケーションにインポートするには、次のように記述します。

```
using EnterpriseDB.EDBClient;
```

4.2.2 .NET フレームワークのセットアップ

次のセクションでは、さまざまな.NET バージョンのセットアップについて説明します。

4.2.3 .NET 6.0

.NET 6.0 の場合、データプロバイダーのインストールパスは次のとおりです。

`C:\Program Files\edb\dotnet\net6.0\`

次の依存関係をプロジェクトに追加する必要があります。

`EnterpriseDB.EDBClient.dll`

使用するアプリケーションの種類によっては、名前空間をソースコードにインポートする必要があります。これおよびライブラリファイルの参照に関するその他の情報については、[ライブラリファイルの参照](#)を参照してください。

4.2.4 .NET 5.0

.NET 5.0 の場合、データプロバイダーのインストールパスは次のとおりです。

`C:\Program Files\edb\dotnet\net5.0\`

次の依存関係をプロジェクトに追加する必要があります。

`EnterpriseDB.EDBClient.dll`

使用するアプリケーションの種類によっては、名前空間をソースコードにインポートする必要があります。これおよびライブラリファイルの参照に関するその他の情報については、[ライブラリファイルの参照](#)を参照してください。

4.2.5 .NET Core 3.1

.NET Core 3.1 を使用している場合、データプロバイダーのインストールパスは次のとおりです。

`C:\Program Files\edb\dotnet\netcoreapp3.1\`

次の共有ライブラリファイルが必要です。

`EnterpriseDB.EDBClient.dll`

`System.Threading.Tasks.Extensions.dll`

`System.Runtime.CompilerServices.Unsafe.dll`

`System.ValueTuple.dll`

`System.Memory.dll`

使用するアプリケーションの種類によっては、名前空間をソースコードにインポートする必要があります。これおよびライブラリファイルの参照に関するその他の情報については、[ライブラリファイルの参照](#)を参照してください。

4.2.6 .NET Standard 2.0

.NET Standard Framework 2.0 の場合、データプロバイダーのインストールパスは次のとおりです。

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

次の依存関係をプロジェクトに追加する必要があります。

EnterpriseDB.EDBClient.dll

System.Threading.Tasks.Extensions.dll

System.Runtime.CompilerServices.Unsafe.dll

System.ValueTuple.dll

注釈: ターゲットフレームワークが .Net Core 2.0 の場合、プロジェクトに次のファイルを含めます。

System.Threading.Tasks.Extensions.dll

使用するアプリケーションの種類によっては、名前空間をソースコードにインポートする必要があります。これおよびライブラリファイルの参照に関するその他の情報については、[ライブラリファイルの参照](#)を参照してください。

4.2.7 .NET Standard 2.1

.NET Standard Framework 2.1 の場合、データプロバイダーのインストールパスは次のとおりです。

C:\Program Files\edb\dotnet\netstandard2.1\

次の共有ライブラリファイルが必要です。

EnterpriseDB.EDBClient.dll

System.Memory.dll

System.Runtime.CompilerServices.Unsafe.dll

System.Text.Json.dll

System.Threading.Tasks.Extensions.dll

System.ValueTuple.dll

使用するアプリケーションの種類によっては、名前空間をソースコードにインポートする必要があります。これおよびライブラリファイルの参照に関するその他の情報については、[ライブラリファイルの参照](#)を参照してください。

4.2.8 .NET EntityFramework コア

Entity Framework Core で使用するよう.NET Connector を構成するには、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

注釈: Entity Framework Core は、net5.0 および netcoreapp3.1 サブディレクトリで使用可能な EnterpriseDB.EDBClient.dll ライブラリで使えます。

ライブラリファイルの参照については、[ライブラリファイルの参照](#) を参照してください。

次の NuGet パッケージが必要です。

```
Microsoft.EntityFrameworkCore.Design
```

```
Microsoft.EntityFrameworkCore.Relational
```

```
Microsoft.EntityFrameworkCore.Abstractions
```

NetCoreApp3.1 の場合、上記のパッケージに加えて次の NuGet パッケージを追加する必要があります。

```
System.Text.Json
```

Entity Framework Core の使用方法については、Microsoft のドキュメントを参照してください。

前提条件 :

dotnet-ef をインストールする (コマンドプロンプトを使用)

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

サンプルプロジェクト

.NET 5.0 または NetCoreApp3.1 に基づいて新しいコンソールアプリケーションを作成します

次の EDB アセンブリへの参照を追加します。

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

```
EnterpriseDB.EDBClient.dll
```

次の NuGet パッケージを追加します。

```
Microsoft.EntityFrameworkCore.Design
```

```
Microsoft.EntityFrameworkCore.Relational
```

```
Microsoft.EntityFrameworkCore.Abstractions
```

NetCoreApp3.1 の場合、上記のパッケージに加えて次の NuGet パッケージを追加します。

```
System.Text.Json
```

コマンドプロンプト

Open Tools → Command Line → Developer Command Prompt

データベース最初のシナリオ

次のコマンドを発行して、指定されたデータベース内のすべてのオブジェクトに対応するモデルクラスを作成します。

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

コードファーストのシナリオ

DbContext を定義し、操作を作成、読み取り、更新、削除するためのコードを追加します。

詳細については、Microsoft のドキュメントを参照してください。次のコマンドを発行して、初期データベースとテーブルを作成します。

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

```
dotnet ef database update --context BloggingContext
```


第5章 Using the .NET Connector

これらの例は、.NET アプリケーションが EDB Postgres Advanced Server データベースに接続して対話できるようにする EDB .NET Connector が提供する EDB オブジェクトクラスを使用する方法を示しています。

これらの例を使用するには、.NET ライブラリファイルをアプリケーションのコンパイル済み形式と同じディレクトリに配置します。これらの例はすべて C# で記述されており、それぞれが ASP.NET ページに埋め込まれています。同じロジックとコードが他の .NET アプリケーション (WinForm やコンソール アプリケーションなど) に適用されます。

サンプルコードと同じディレクトリに以下の `web.config` ファイルを作成して保存します。例では、この構成ファイルの `DB_CONN_STRING` キーを利用して、EDB Postgres Advanced Server ホストから接続文字列を返します。

```
<?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>
```

ASP.NET Web アプリケーションの EDB Postgres Advanced Server 接続文字列は、`web.config` ファイルに保存されています。ASP.NET を使用しないアプリケーションを作成している場合は、`app.config` などのアプリケーション構成ファイルで接続情報を提供します。

第6章 Opening a database connection

EDBConnection オブジェクトは、EDB Postgres Advanced Server のインスタンスと .NET アプリケーション間の通信を処理します。EDB Postgres Advanced Server データベースに保存されているデータにアクセスする前に、EDBConnection オブジェクトを作成して開く必要があります。

以下の例は、Advanced Server のインスタンスに接続するための基本的な手順を示しています。次のことを行う必要があります。

1. 名前空間 `EnterpriseDB.EDBClient` をインポートします。
2. EDBConnection のインスタンスを作成します。
3. 接続文字列をパラメーターとして EDBConnection クラスのコンストラクターに渡して、EDBConnection オブジェクトを初期化します。
4. EDBConnection オブジェクトの `Open` メソッドを呼び出して、接続を開きます。

6.1 接続文字列パラメーター

有効な接続文字列は、EDB Postgres Advanced Server インスタンスの場所と認証情報を指定します。接続を開く前に、接続文字列を指定する必要があります。接続文字列には以下が含まれている必要があります。

- サーバーの名前または IP アドレス
- EDB Postgres Advanced Server データベースの名前
- EDB Postgres Advanced Server ユーザーの名前
- そのユーザーに関連付けられたパスワード

接続文字列には次のパラメーターを含めることができます。

CommandTimeout

CommandTimeout は、例外をスローする前にコマンドの実行が完了するのを待機する時間（秒）を指定します。デフォルト値は 20 です。

ConnectionLifeTime

ConnectionLifeTime を使用して、プール内の未使用の接続を閉じるまでの待機時間（秒）を指定します。デフォルト値は 15 です。

Database

Database パラメーターを使用して、アプリケーションが接続するデータベースの名前を指定します。デフォルトは接続しているユーザーの名前です。

Encoding

Encoding パラメーターは廃止されました。このパラメーターは常に文字列 **unicode** を返し、それを設定しようとしても黙って無視します。

Integrated Security

Windows 統合セキュリティを使用するには、値 **true** を指定します。デフォルトでは、**Integrated Security** は **false** に設定され、Windows 統合セキュリティは無効になっています。

Load Role Based Tables

Load Role Based Tables を使用して、ロールに基づいてテーブル OID をロードします。この変更は、テーブル型 OID のロードにのみ影響し、複合型には影響しません。このパラメーターを **true** に設定すると、新しい機能がトリガーされます。デフォルト値は **false** です。

MaxPoolSize

MaxPoolSize は、プールが指定された接続数を超えた場合に、プールされた接続を破棄するように指示します。デフォルト値は **20** です。

MinPoolSize

MinPoolSize は、サーバーとの指定された数の接続を事前に割り当てるように指示します。デフォルト値は **1** です。

Password

クリアテキスト認証を使用する場合、サーバーとの接続を確立するために使用するパスワードを指定します。

Pooling

接続プーリングを無効にするには、**false** の値を指定します。デフォルトでは、**Pooling** は **true** に設定され、接続プーリングを有効にします。

Port

Port パラメーターは、アプリケーションの接続先のポートを指定します。

Protocol

(自動ではなく) 使用する特定のプロトコルバージョン。2 または 3 の整数値を指定します。

SearchPath

SearchPath パラメーターを使用して、検索パスを名前付きスキーマおよびパブリックスキーマに変更します。

Server

EDB Postgres Advanced Server ホストの名前または IP アドレス。

SSL

値 **true** を指定して、安全な接続を試みます。デフォルトでは、**SSL** は **false** に設定されます。

sslmode

sslmode を使用して、SSL 接続制御設定を指定します。sslmode は次のとおりです。

prefer —可能であれば SSL を使用してください。

require — SSL 接続を確立できない場合は例外をスローします。

allow —SSL なしで接続します。このパラメーターはサポートされていません。

disable —SSL 接続を試行しないでください。これがデフォルトの動作です。

SyncNotification

SyncNotification パラメーターを使用して、EDBDataprovider が同期通知を使用するように指定します。デフォルト値は **false** です。

Timeout

Timeout は、開いている接続を待機する時間（秒）を指定します。デフォルト値は 15 です。

User Id

User Id パラメーターは、接続に使用するユーザー名を指定します。

6.2 例：ASP.NET を使用してデータベース接続を開く

この例では、EDB Postgres Advanced Server のインスタンスへの接続を開き、接続を閉じる方法を示します。接続は、DB_CONN_STRING 構成パラメーターで指定された資格情報を使用して確立されます。接続情報の概要については、

Using the .NET Connector を参照してください。接続パラメーターについては、[接続文字列パラメーター](#) も参照してください。

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

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

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

    EDBConnection conn = new EDBConnection(strConnectionString);

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

    catch(EDBException exp)
```

(次のページに続く)

(前のページからの続き)

```
{
    exp.ToString();
}

finally
{
    conn.Close();
}
}

</script>
```

接続が成功すると、接続が正常に開かれたことを示すメッセージが表示されます。

6.3 例：コンソールアプリケーションからデータベース接続を開く

この例では、コンソールベースのアプリケーションを使用して EDB Postgres Advanced Server データベースとの接続を開きます。

コンソールアプリケーションのコードを記述する前に、データベースへの接続文字列を格納する `app.config` ファイルを作成します。構成ファイルを使用すると、情報が変更された場合に接続文字列を更新すると便利です。

```
<?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>
```

次のコードサンプルをファイルに入力します。

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

namespace EnterpriseDB
{
    class EDB
    {
```

(次のページに続く)

(前のページからの続き)

```

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

    EDBConnection conn = new EDBConnection(strConnectionString);

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

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

    finally
    {
        conn.Close();
    }
}
}

```

ファイルを EDBConnection-Sample.cs として保存し、次のコマンドでコンパイルします。

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

サンプルをコンパイルすると、Console.exe ファイルが生成されます。Console.exe と入力してサンプルコードを実行できます。実行すると、コンソールは正常に開いたことを確認します。

6.4 例：Windows フォームアプリケーションからデータベース接続を開く

この例では、.NET WinForm アプリケーションを使用してデータベース接続を開きます。この例を使用するには、ライブラリファイルが含まれるディレクトリに次のコードを WinForm-Example.cs として保存します。

```

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

```

(次のページに続く)

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

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

            Application.Run(frmMain);
        }

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

接続するデータベースを指すようにデータベース接続文字列を変更します。次に、次のコマンドでファイ

ルをコンパイルします。

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

このコマンドは、実行可能ファイルがコンパイルされたのと同じフォルダーに WinForm.exe ファイルを生成します。実行可能ファイルを呼び出すと、接続が成功したことを示すメッセージが表示されます。

第7章 Retrieving database records

SELECT ステートメントを使用して、SELECT コマンドを使用してデータベースからレコードを取得できます。SELECT ステートメントを実行するには、次のことを行う必要があります。

- データベース接続を作成して開きます。
- SELECT ステートメントを表す EDBCommand オブジェクトを作成します。
- EDBDataReader を返す EDBCommand オブジェクトの ExecuteReader() メソッドでコマンドを実行します。
- EDBDataReader をループし、結果を表示するか、EDBDataReader をコントロールにバインドします。

EDBDataReader オブジェクトは、データベースレコードの順方向専用および読み取り専用のストリームを表し、一度に1つのレコードが表示されます。ストリーム内の後続のレコードを表示するには、EDBDataReader オブジェクトの Read() メソッドを呼び出す必要があります。

次の例：

1. EDB Postgres Advanced Server の名前空間 EnterpriseDB.EDBClient をインポートします。
2. SELECT ステートメントで EDBCommand オブジェクトを初期化します。
3. データベースへの接続を開きます。
4. EDBCommand オブジェクトの ExecuteReader メソッドを呼び出して、EDBCommand を実行します。

SQL ステートメントの結果は、EDBDataReader オブジェクトに取得されます。

EDBDataReader オブジェクトの内容をループして、WHILE ループのクエリによって返されたレコードを表示します。

Read() メソッドは、次のレコード（レコードが存在する場合）に進み、レコードが存在する場合は true を返します。EDBDataReader が結果セットの最後に到達した場合、false を返します。

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

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

private void Page_Load(object sender, System.EventArgs e)
{
    string strConnectionString = ConfigurationManager.AppSettings
```

(次のページに続く)

(前のページからの続き)

```

["DB_CONN_STRING"];
EDBConnection conn = new EDBConnection(strConnectionString);

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

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

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

```

サンプルコードを実行するには、コードをデフォルトの Web ルートディレクトリの `selectEmployees.aspx` . 次に、プログラムを呼び出すには、ブラウザーに次の URL を入力します。

7.1 単一のデータベースレコードを取得する

クエリから単一の結果を取得するには、`EDBCommand` オブジェクトの `ExecuteScalar()` メソッドを使用します。`ExecuteScalar()` メソッドは、指定されたクエリによって生成された `DataSet` の最初の行の最初の列の最初の値を返します。

```

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

```

(次のページに続く)

(前のページからの続き)

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

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

        cmd.CommandType = CommandType.Text;

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

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

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

サンプルコードを Web ルートディレクトリの `selectscalar.aspx` という名前のファイルに保存します。サンプルコードを呼び出すには、ブラウザーに次を入力します。

このサンプルには、`ExecuteScalar()` メソッドによって返される値の明示的な変換が含まれています。`ExecuteScalar()` メソッドはオブジェクトを返します。オブジェクトを表示するには、`Convert.ToInt32` メソッドを使用して整数値に変換する必要があります。

第8章 Parameterized queries

_パラメーター化されたクエリ_は、SQL ステートメントに 1 つ以上のパラメーターマーカーが埋め込まれたクエリです。パラメーター化されたクエリを実行する前に、SQL ステートメントのテキストにある各マーカーの値を指定する必要があります。

パラメーター化されたクエリは、コードを記述するときにクエリの完全なテキストがわからない場合に役立ちます。たとえば、WHERE 句で参照される値は、ユーザー入力から計算できます。

次の例に示すように、EDBParameter オブジェクトを作成し、そのオブジェクトをコマンドのパラメーターコレクションに追加することにより、パラメーター化されたクエリで指定された各パラメーターのデータ型を宣言する必要があります。次に、パラメーターの Value() 関数を呼び出して、各パラメーターの値を指定する必要があります。

この例は、従業員の給与を引き上げる UPDATE ステートメントでパラメーター化されたクエリを使用する方法を示しています。

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

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

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

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

    try {
        conn.Open();

        EDBCommand cmdUpdate = new EDBCommand(updateQuery,conn);

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

        cmdUpdate.Parameters[0].Value = 7788;
```

(次のページに続く)

(前のページからの続き)

```
cmdUpdate.ExecuteNonQuery();

Response.Write("Record Updated");

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

サンプルコードを Web ルートディレクトリの `updateSalary.aspx` という名前のファイルに保存します。

サンプルコードを呼び出すには、ブラウザーに次を入力します。

第9章 Inserting records in a database

EDBCommand の ExecuteNonQuery() メソッドを使用して、INSERT コマンドで EDB Postgres Advanced Server ホストに保存されているデータベースにレコードを追加できます。

次の例では、INSERT コマンドは変数 cmd に保存されます。コロンが付いた値(:) は EDBParameters のプレースホルダーであり、インスタンス化され、値が割り当てられ、以降のステートメントで INSERT コマンドのパラメーターコレクションに追加されます。INSERT コマンドは、cmdInsert オブジェクトの ExecuteNonQuery() メソッドによって実行されます。

この例では、従業員を emp テーブルに追加します。

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

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

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

    try
    {
        conn.Open();

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

        EDBCommand cmdInsert = new EDBCommand(cmd,conn);

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

        cmdInsert.Parameters[0].Value = 1234;

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

(次のページに続く)

(前のページからの続き)

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

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

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

サンプルコードを Web ルートディレクトリの `insertEmployee.aspx` という名前のファイルに保存します。

サンプルコードを呼び出すには、ブラウザに次を入力します。

第10章 Deleting records in a database

EDBCommand の ExecuteNonQuery() メソッドを使用して、DELETE ステートメントで EDB Postgres Advanced Server ホストに保存されているデータベースからレコードを削除できます。

次の例では、DELETE コマンドは変数 strDeleteQuery に保存されます。このコードは、従業員番号を DELETE コマンド (EmpNo で指定) に渡します。次に、ExecuteNonQuery() メソッドを使用してコマンドが実行されます。

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

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

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

    EDBConnection conn = new EDBConnection(strConnectionString);

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

    try
    {
        conn.Open();

        EDBCommand deleteCommand = new EDBCommand(strDeleteQuery,conn);

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

        deleteCommand.Parameters[0].Value = 1234;

        deleteCommand.ExecuteNonQuery();

        Response.Write("Record Deleted");
    }
}
```

(次のページに続く)

(前のページからの続き)

```
    }

    catch(Exception exp)
    {

        Response.Write(exp.ToString());

    }

    finally
    {

        conn.Close();

    }
}
</script>
```

サンプルコードを Web ルートディレクトリの `deleteEmployee.aspx` という名前のファイルに保存します。

サンプルコードを呼び出すには、ブラウザに次を入力します。

第11章 Using SPL stored procedures in your .NET application

アプリケーションに SQL ステートメントを含めるには、次の 2 つの方法があります。

- .NET アプリケーションコードに SQL ステートメントを直接追加する。
- SQL 文をストアードプロシージャにパッケージ化し、.NET アプリケーションからストアードプロシージャを実行する。

場合によっては、ストアードプロシージャは埋め込み SQL ステートメントよりも優れています。ストアードプロシージャは、アプリケーションに直接埋め込まれた SQL ステートメントでは複製が困難な複雑な条件付きおよびループ構造をサポートします。

ストアードプロシージャを使用しても、パフォーマンスが向上します。ストアードプロシージャは、サーバー側で 1 回だけ解析、コンパイル、および最適化する必要があります。アプリケーションに含まれる SQL ステートメントは、.NET アプリケーションから実行されるたびに解析、コンパイル、および最適化される場合があります。

.NET アプリケーションでストアードプロシージャを使用するには、次のことを行う必要があります。

1. EDB Postgres Advanced Server ホストで SPL ストアードプロシージャを作成します。
2. EnterpriseDB.EDBClient 名前空間をインポートします。
3. ストアードプロシージャの名前を EDBCommand のインスタンスに渡します。
 4. EDBCommand.CommandType を CommandType.StoredProcedure に変更します。
 5. コマンドを Prepare() します。
 6. コマンドを実行します。

11.1 例：パラメーターなしでストアードプロシージャを実行する

このサンプルプロシージャは、部門 10 の名前を出力します。このプロシージャはパラメーターをとらず、パラメーターを返しません。サンプルプロシージャを作成するには、EDB-PSQL を呼び出して EDB Postgres Advanced Server ホストデータベースに接続します。コマンドラインで次の SPL コードを入力します。

```
CREATE OR REPLACE PROCEDURE list_dept10
IS
    v_deptname VARCHAR2(30);
BEGIN
```

(次のページに続く)

(前のページからの続き)

```

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;

```

EDB Postgres Advanced Server がストアードプロシージャを検証すると、CREATE PROCEDURE をエコーします。

11.1.1 EDBCommand オブジェクトを使用してストアードプロシージャを実行する

EDBCommand オブジェクトの CommandType プロパティは、実行されているコマンドの種類を示します。CommandType プロパティは、3つの可能な CommandType 列挙値のいずれかに設定されます。

- 実行のために SQL 文字列を渡すときは、デフォルトの Text 値を使用します。
- StoredProcedure 値を使用し、実行するストアードプロシージャの名前を渡します。
- テーブル名を渡すときに TableDirect 値を使用します。この値は、指定されたテーブルのすべてのレコードを返します。

CommandText プロパティには、CommandType プロパティの値に応じて、SQL 文字列、ストアードプロシージャ名、またはテーブル名が含まれている必要があります。

次の例では、ストアードプロシージャを実行します。

```

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

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

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

    try
    {
        conn.Open();

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

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

```

(次のページに続く)

(前のページからの続き)

```

        Response.Write("Stored Procedure Executed Successfully");
    }
    catch(Exception exp)
    {
        Response.Write(exp.ToString());
    }
    finally
    {
        conn.Close();
    }
}
</script>

```

サンプルコードを Web ルートディレクトリの `storedProc.aspx` という名前のファイルに保存します。

サンプルコードを呼び出すには、ブラウザに次を入力します。

11.2 例：IN パラメーターを使用したストアードプロシージャの実行

次の例は、IN パラメーターを含むストアードプロシージャの呼び出しを示しています。サンプルプロシージャを作成するには、EDB-PSQL を呼び出し、EDB Postgres Advanced Server ホストデータベースに接続します。コマンドラインで次の SPL コードを入力します。

```

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)

```

(次のページに続く)

(前のページからの続き)

```
VALUES (MAX_EMPNO+1,pENAME,pJOB,pSAL,pCOMM,pDEPTNO,pMgr);
CLOSE testcur;
END;
```

EDB Postgres Advanced Server がストアードプロシージャを検証すると、CREATE PROCEDURE をエコーします。

11.2.1 入力値をストアードプロシージャに渡す

パラメーターを含むストアードプロシージャの呼び出しは、パラメーターなしのストアードプロシージャの実行に似ています。主な違いは、パラメーター化されたストアードプロシージャを呼び出すときは、EDBCommand オブジェクトの EDBParameter コレクションを使用する必要があることです。EDBParameter が EDBCommand コレクションに追加されると、ParameterName、DbType、Direction、Size、Value などのプロパティが設定されます。

次の例は、C# スクリプトからパラメーター化されたストアードプロシージャを実行するプロセスを示しています。

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

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

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

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

    try
    {
        conn.Open();

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

(次のページに続く)

(前のページからの続き)

```
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;

cmdStoredProc.Prepare();

cmdStoredProc.ExecuteNonQuery();

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

Response.Write(empInfo);
}

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

(次のページに続く)

```
    }  
    finally  
{  
        conn.Close();  
    }  
}  
  
</script>  
  
</script>
```

サンプルコードを Web ルートディレクトリの `storedProcInParam.aspx` という名前のファイルに保存します。

サンプルコードを呼び出すには、ブラウザーに次を入力します。

この例では、`Page_Load` メソッドの本体で `EDBConnection` オブジェクトを宣言してインスタンス化します。このサンプルでは、ストアドプロシージャの実行に必要なプロパティを使用して `EDBCommand` オブジェクトを作成します。

この例では、`EDBCommand Parameter` コレクションの `Add` メソッドを使用して、6つの入力パラメーターを追加します。

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

`EMP_INSERT` ストアドプロシージャに渡す前に、各パラメーターに値を割り当てます。

`Prepare()` メソッドは、`ExecuteNonQuery()` メソッドを呼び出す前にステートメントを準備します。

`EDBCommand` オブジェクトの `ExecuteNonQuery` メソッドは、ストアドプロシージャを実行します。ストアドプロシージャが実行されると、テストレコードが `emp` テーブルに挿入され、挿入された値が Web ページに表示されます。

11.3 例：IN、OUT、および INOUT パラメーターを使用したストアドプロシージャの実行

前の例では、`IN` パラメーターをストアドプロシージャに渡す方法を示しました。次の例は、ストアドプロシージャから `IN` 値を渡し、`OUT` 値を返す方法を示しています。

11.3.1 ストアドプロシージャの作成

次のストアドプロシージャは、部門番号を渡し、対応する場所と部門名を返します。サンプルプロシージャを作成するには、EDB-PSQL コマンドラインを開き、EDB Postgres Advanced Server ホストデータベースに接続します。コマンドラインで次の SPL コードを入力します。

```
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;
```

EDB Postgres Advanced Server がストアドプロシージャを検証すると、CREATE PROCEDURE をエコーします。

11.3.2 ストアドプロシージャからの出力値の受信

OUT パラメーターから値を取得する場合、それらのパラメーターの方向を **Output** として明示的に指定する必要があります。2つの方法で **Output** パラメーターから値を取得できます。

- EDBCommand の **ExecuteReader** メソッドを呼び出し、返された **EDBDataReader** を明示的にループし、OUT パラメーターの値を検索します。
- EDBCommand の **ExecuteNonQuery** メソッドを呼び出し、その **EDBParameter** 値プロパティを呼び出すことにより、宣言された **Output** パラメーターの値を明示的に取得します。

各メソッドでは、パラメーターの方向を示す各パラメーター (**ParameterDirection.Input** 、 **ParameterDirection.Output** 、または **ParameterDirection.InputOutput**) を宣言する必要があります。プロシージャーを呼び出す前に、各 IN および INOUT パラメーターの値を指定する必要があります。プロシージャーが戻ったら、**command.Parameters[]** 配列から OUT および INOUT パラメーター値を取得できます。

次のコードは、**ExecuteReader** メソッドを使用して結果セットを取得する方法を示しています。

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

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

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

    try
    {

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

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

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

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

        command.Prepare();

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

        int fc = result.FieldCount;

        for (int i = 0; i < 3; i++)
        {
```

(次のページに続く)

(前のページからの続き)

```

        Console.WriteLine("RESULT[" + i + "]=" + Convert.ToString(command.
↪Parameters[i].Value));
        Console.WriteLine("\n");
    }

}

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

</script>

```

次のコードは、ExecuteNonQuery メソッドを使用して結果セットを取得する方法を示しています。

```

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

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

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

    try
    {

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

        command.Parameters.Add(new EDBParameter("pDEPTNO",
            EDBTypes.EDBDbType.Integer,10,"pDEPTNO",
            ParameterDirection.Input,false ,2,2,

```

(次のページに続く)

```
        System.Data.DataRowVersion.Current,1));

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

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

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

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

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

第12章 Using advanced queueing

EDB Postgres Advanced Server アドバンストキューイングは、EDB Postgres Advanced Server データベースのメッセージキューとメッセージ処理を提供します。ユーザー定義メッセージはキューに格納されます。キューのコレクションは、キュー表に格納されます。キュー表に依存するキューを作成する前に作成します。

サーバー側では、DBMS_AQADM パッケージのプロシージャによってメッセージキューとキューテーブルが作成および管理されます。DBMS_AQ パッケージを使用して、キューにメッセージを追加または削除したり、PL/SQL コールバックプロシージャを登録または登録解除します。DBMS_AQ および DBMS_AQADM の詳細については、DBMS_AQ を参照してください。

クライアント側では、アプリケーションは EDB.NET ドライバーを使用してメッセージをエンキューおよびデキューします。

12.1 メッセージのエンキューまたはデキュー

EDB Postgres Advanced Server の高度なキューイング機能の使用の詳細については、Database Compatibility for Oracle Developers Built-in Package Guide を参照してください。

12.1.1 サーバーサイドのセットアップ

.NET アプリケーションでアドバンストキューイング機能を使用するには、最初にユーザー定義型、キューテーブル、およびキューを作成し、データベースサーバーでキューを起動する必要があります。EDB-PSQL を呼び出し、EDB Postgres Advanced Server ホストデータベースに接続します。コマンドラインで次の SPL コマンドを使用します。

ユーザー定義型の作成

RAW データ型を指定するには、ユーザー定義型を作成します。次の例は、myxml という名前のユーザー定義型の作成を示しています。

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

キューテーブルの作成

キュー表は、同じペイロードタイプの複数のキューを保持できます。次の例は、MSG_QUEUE_TABLE という名前のテーブルの作成を示しています。

```
EXEC DBMS_AQADM.CREATE_QUEUE_TABLE
(queue_table => MSG_QUEUE_TABLE,
queue_payload_type => myxml,
```

(次のページに続く)

(前のページからの続き)

```
comment => Message queue table);
END;
```

キューを作成しています

次の例は、テーブル MSG_QUEUE_TABLE に MSG_QUEUE という名前のキューを作成する方法を示しています。

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

キューを開始しています

キューが作成されたら、コマンドラインで次の SPL コードを呼び出して、EDB データベースでキューを開始します。

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

12.1.2 クライアントサイドの例

ユーザー定義型を作成したら、続いてキューテーブルとキューを作成します。次に、EDB .Net ドライバーを使用してメッセージをエンキューまたはデキューできます。

メッセージをエンキューする

.NET アプリケーションでメッセージをエンキューするには、次のことを行う必要があります。

1. EnterpriseDB.EDBClient 名前空間をインポートします。
2. キューの名前を渡し、EDBAQueue のインスタンスを作成します。
3. エンキューメッセージを作成し、ペイロードを定義します。
4. queue.Enqueue メソッドを呼び出します。

次のコードは、queue.Enqueue メソッドの使用を示しています。

注釈: 次のコードはメッセージを作成してシリアル化します。これはサンプルコードであり、そのままコピーするとコンパイルされません。メッセージを XML としてシリアル化する必要があります。

```
using EnterpriseDB.EDBClient;
using System;
using System.Collections.Generic;
```

(次のページに続く)

(前のページからの続き)

```

using System.Linq;
using System.Text;
using System.Threading.Tasks;

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

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

        private static string ByteArrayToString(byte[] byteArray)
        {
            // Sanity check if its null so we dont incur overhead of an exception
            if (byteArray == null)
            {
                return string.Empty;
            }
            try
            {
                StringBuilder hex = new StringBuilder(byteArray.Length * 2);
                foreach (byte b in byteArray)

```

(次のページに続く)

```

        {
            hex.AppendFormat("{0:x2}", b);
        }

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

private static bool EnqueueMsg(string msg)
{
    EDBConnection con = GetConnection();
    using (EDBAQueue queue = new EDBAQueue("MSG_QUEUE", con))
    {
        queue.MessageType = EDBAQueueMessageType.Xml;
        EDBTransaction txn = queue.Connection.BeginTransaction();
        QueuedEntities.Message queuedMessage = new QueuedEntities.Message() {
→MessageText = msg };

        try
        {
            string rootElementName = queuedMessage.GetType().Name;
            if (rootElementName.IndexOf(".") != -1)
            {
                rootElementName = rootElementName.Split('.')[0];
            }

            string xml = new Utils.XmlFragmentSerializer<QueuedEntities.
→Message>().Serialize(queuedMessage);
            EDBAQueueMessage queMsg = new EDBAQueueMessage();
            queMsg.Payload = new MyXML { value = xml };
            queue.MessageType = EDBAQueueMessageType.Udt;
            queue.UdtTypeName = "myxml";
            EDBConnection.GlobalTypeMapper.MapComposite<MyXML>("myxml");
            con.ReloadTypes();
            queue.Enqueue(queMsg);
            var messageId = ByteArrayToString((byte[])queMsg.MessageId);
            Console.WriteLine("MessageID: " + messageId);
            txn.Commit();
            queMsg = null;
            xml = null;
            rootElementName = null;

```

(前のページからの続き)

```

        return true;
    }
    catch (Exception ex)
    {
        txn?.Rollback();
        Console.WriteLine("Failed to enqueue message.");
        Console.WriteLine(ex.ToString());
        return false;
    }
    finally
    {
        queue?.Connection?.Dispose();
    }
}
}
}
}
}

```

メッセージのデキュー

.NET アプリケーションでメッセージをデキューするには、次のことを行う必要があります。

1. EnterpriseDB.EDBClient 名前空間をインポートします。
2. キューの名前を渡し、EDBAQueue のインスタンスを作成します。
3. queue.Dequeue メソッドを呼び出します。

注釈: 次のコードはメッセージを作成してシリアル化します。これはサンプルコードであり、そのままコピーするとコンパイルされません。メッセージを XML としてシリアル化する必要があります。

```

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

namespace DequeueXML
{
    class MyXML
    {
        public string value { get; set; }
    }
}

```

(次のページに続く)

```

class Program
{
    static void Main(string[] args)
    {
        DequeueMsg();
    }

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

    private static string ByteArrayToString(byte[] byteArray)
    {
        // Sanity check if its null so we dont incur overhead of an exception
        if (byteArray == null)
        {
            return string.Empty;
        }
        try
        {
            StringBuilder hex = new StringBuilder(byteArray.Length * 2);
            foreach (byte b in byteArray)
            {
                hex.AppendFormat("{0:x2}", b);
            }

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

    public static void DequeueMsg(int waitTime = 10)
    {
        EDBConnection con = GetConnection();
        using (EDBAQueue queueListen = new EDBAQueue("MSG_QUEUE", con))
        {
            queueListen.UdtTypeName = "myxml";
            queueListen.DequeueOptions.Navigation = EDBAQueueNavigationMode.FIRST_
→MESSAGE;

```

(次のページに続く)

(前のページからの続き)

```

queueListen.DequeueOptions.Visibility = EDBAQVisibility.ON_COMMIT;
queueListen.DequeueOptions.Wait = 1;
EDBTransaction txn = null;

while (1 == 1)
{
    if (queueListen.Connection.State == System.Data.ConnectionState.
→Closed)
    {
        queueListen.Connection.Open();
    }

    string messageId = "Unknown";
    try
    {
        // the listen function is a blocking function. It will Wait
→the specified waitTime or until a
        // message is received.
        Console.WriteLine("Listening...");
        string v = queueListen.Listen(null, waitTime);
        // If we are waiting for a message and we specify a Wait time,
        // then if there are no more messages, we want to just bounce
→out.

        if (waitTime > -1 && v == null)
        {
            Console.WriteLine("No message received during Wait period.
→");

            Console.WriteLine();
            continue;
        }

        // once were here that means a message has been detected in
→the queue. Lets deal with it.
        txn = queueListen.Connection.BeginTransaction();

        Console.WriteLine("Attempting to dequeue message...");
        // dequeue the message
        EDBAQMessage deqMsg;
        try
        {
            deqMsg = queueListen.Dequeue();
        }
        catch (Exception ex)
        {

```

(次のページに続く)

(前のページからの続き)

```

        if (ex.Message.Contains("ORA-25228"))
        {
            Console.WriteLine("Message was not there.  Another
→process must have picked it up.");
            Console.WriteLine();
            txn.Rollback();
            continue;
        }
        else
        {
            throw;
        }
    }

    messageId = ByteArrayToString((byte[])deqMsg.MessageId);
    if (deqMsg != null)
    {
        Console.WriteLine("Processing received message...");
        // process the message payload
        MyXML obj = (MyXML) deqMsg.Payload;

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

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

        txn.Commit();

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

    }
    else
    {
        Console.WriteLine("Message was not dequeued.");
    }
}
catch (Exception ex)
{
    Console.WriteLine("Failed To dequeue or process the dequeued
→message.");
}

```

(次のページに続く)

(前のページからの続き)

```
        Console.WriteLine(ex.ToString());  
        Console.WriteLine();  
        if (txn != null)  
        {  
            txn.Rollback();  
            if (txn != null)  
            {  
                txn.Dispose();  
            }  
        }  
    }  
}  
  
}
```

12.2 EDBAQ クラス

このアプリケーションでは、次の EDBAQ クラスが使用されます。

EDBAQDequeueMode

EDBAQDequeueMode クラスには、使用可能なすべてのデキューアモードがリストされています。

Value	Description
Browse	ロックせずにメッセージを読み取ります。
Locked	メッセージを読み取り、書き込みロックを取得します。
Remove	読んだらメッセージを削除します。これがデフォルト値です。
Remove_NoData	メッセージの受信を確認します。

EDBAQDequeueOptions

EDBAQDequeueOptions クラスは、メッセージをデキューするときに使用できるオプションをリストします。

Property	Description
ConsumerName	メッセージをデキューするコンシューマの名前。
DequeueMode	これは `EDBAQDequeueMode` から設定されます。 <code>dequeue</code> オプションにリンクされたロック動作を表します。
Navigation	これは `EDBAQNavigationMode` から設定されます。取得するメッセージの位置を表します。

次のページに続く

表 2 – 前のページからの続き

Property	Description
Visibility	これは `EDBAQVisibility` から設定されます。これは、新しいメッセージが現在のトランザクションの一部としてデキューされるかどうかを表します。
Wait	検索条件によるメッセージの待機時間。
Msgid	メッセージ識別子。
Correlation	相関識別子。
DeqCondition	デキューの状態。ブール式です。
Transformation	メッセージをデキューする前に適用する変換。
DeliveryMode	デキューされたメッセージの配信モード。

EDBAQEnqueueOptions

EDBAQEnqueueOptions クラスは、メッセージをエンキューするときに使用できるオプションをリストします。

Property	Description
Visibility	これは `EDBAQVisibility` から設定されます。新しいメッセージが現在のトランザクションの一部としてエンキューされるかどうかを表します。
RelativeMsgid	相対メッセージ識別子。
SequenceDeviation	メッセージをデキューする順序。
Transformation	メッセージをエンキューする前に適用する変換。
DeliveryMode	エンキューされたメッセージの配信モード。

EDBAQ メッセージ

EDBAQMessage クラスは、エンキュー/デキューするメッセージをリストします。

Property	Description
Payload	キューに入れられる実際のメッセージ。
MessageId	キューに入れられたメッセージの ID。

EDBAQMessageProperties

EDBAQMessageProperties は、使用可能なメッセージプロパティをリストします。

Property	Description
Priority	メッセージの優先度。
Delay	メッセージをデキューできる投稿時間（秒単位）。
Expiration	メッセージをデキューできる時間（秒）。
Correlation	相関識別子。
Attempts	メッセージのデキューの試行回数。
RecipientList	デフォルトのキューサブスクライバーを無効にする受信者リスト。
ExceptionQueue	未処理のメッセージの移動先のキューの名前。
EnqueueTime	メッセージがエンキューされた時刻。

次のページに続く

表 5 – 前のページからの続き

Property	Description
State	デキュー中のメッセージの状態。
OriginalMsgid	最後のキューのメッセージ識別子。
TransactionGroup	デキューされたメッセージのトランザクショングループ。
DeliveryMode	デキューされたメッセージの配信モード。

EDBAQMessageState

EDBAQMessageState クラスは、デキュー中のメッセージの状態を表します。

Value	Description
Expired	メッセージは例外キューに移動されます。
Processed	メッセージは処理されて保持されます。
Ready	メッセージを処理する準備ができました。
Waiting	メッセージは待機状態です。遅延に達していません。

EDBAQMessageType

EDBAQMessageType クラスは、ペイロードのタイプを表します。

Value	Description
Raw	生のメッセージタイプ。 注: 現在、このペイロードタイプはサポートされていません。
UDT	ユーザー定義型のメッセージ。
XML	XML タイプのメッセージ。 注: 現在、このペイロードタイプはサポートされていません。

EDBAQNavigationMode

EDBAQNavigationMode クラスは、使用可能なさまざまなタイプのナビゲーションモードを表します。

Value	Description
First_Message	検索語に一致する最初の使用可能なメッセージを返します。
Next_Message	検索項目に一致する次の利用可能なメッセージを返します。
Next_Transaction	次のトランザクショングループの最初のメッセージを返します。

EDBAQQueue

EDBAQQueue クラスは、PostgreSQL データベースで DMBS_AQ 機能を実行する SQL ステートメントを表します。

Property	Description
Connection	使用する接続
Name	キューの名前

次のページに続く

表 9 – 前のページからの続き

Property	Description
MessageType	このキューからエンキュー/デキューされるメッセージタイプ (例: <i>EDBAQMessageType.Udt</i>)
UdtTypeName	メッセージ型のユーザー定義型名
EnqueueOptions	使用するエンキューオプション
DequeueOptions	使用するデキューオプション
MessageProperties	使用するメッセージのプロパティ

EDBAQ 可視性

EDBAQVisibility クラスは、使用可能な可視性オプションを表します。

Value	Description
Immediate	エンキュー/デキューは、進行中のトランザクションの一部ではありません。
On_Commit	エンキュー/デキューは現在のトランザクションの一部です。

注釈:

- これらのパラメーターのデフォルトオプションを確認するには、DBMS_AQ を参照してください。 - EDB アドバンストキューイング機能は、ユーザー定義型を使用してエンキュー/デキュー操作を呼び出します。 *NoTypeLoading* はユーザー定義型をロードしないため、*Server Compatibility Mode=NoTypeLoading* はアドバンストキューイングでは使用できません。

第13章 Using a ref cursor in a .NET application

`ref cursor` は、クエリ結果セットへのポインターを含むカーソル変数です。結果セットは、カーソル変数を使用して `OPEN FOR` ステートメントを実行することにより決定されます。カーソル変数は、静的カーソルのような特定のクエリに関連付けられていません。異なるクエリを含む `OPEN FOR` ステートメントを使用して、毎回同じカーソル変数を開くことができます。そのクエリの新しい結果セットが作成され、カーソル変数を介して利用できるようになります。カーソル変数は、次の2つの方法で宣言できます。

- `SYS_REFCURSOR` 組み込みデータ型を使用して、弱く型指定された参照カーソルを宣言します。
- その型の変数を宣言する厳密に型指定された参照カーソルを定義します。

`SYS_REFCURSOR` は、任意の結果セットに関連付けることができる参照カーソルタイプです。これは、弱く型指定された参照カーソルと呼ばれます。次の例は、弱く型指定された参照カーソルの宣言です。

```
name SYS_REFCURSOR;
```

以下は、厳密に型指定された参照カーソルの例です。

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

13.1 ストアドプロシージャの作成

次のサンプルコードでは、`refcur_inout_callee` というストアドプロシージャを作成します。OUT パラメーターとして渡される参照カーソルのデータ型を指定します。サンプルプロシージャを作成するには、EDB-PSQL を呼び出して EDB Postgres Advanced Server ホストデータベースに接続します。コマンドラインで次の SPL コードを入力します。

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

次の C#コードでは、ストアドプロシージャを使用して、`emp` テーブルから従業員の名前を取得します。

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

(次のページに続く)

```
namespace EDBRefCursor
{
    class EmpRefcursor
    {
        [STAThread]
        static void Main(string[] args)
        {
            string strConnectionString =
                ConfigurationManager.AppSettings["DB_CONN_STRING"];
            EDBConnection conn = new EDBConnection(strConnectionString);
            conn.Open();
            try
            {
                EDBTransaction tran = conn.BeginTransaction();
                EDBCommand command = new EDBCommand("refcur_inout_callee",
                    conn);
                command.CommandType = CommandType.StoredProcedure;
                command.Transaction = tran;
                command.Parameters.Add(new EDBParameter("refCursor",
                    EDBTypes.EDBDbType.Refcursor, 10, "refCursor",
                    ParameterDirection.Output, false, 2, 2,
                    System.Data.DataRowVersion.Current, null));

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

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

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

(前のページからの続き)

```
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message.ToString());
        }
    }
}
```

次の .NET コード スニペットは、コンソールに結果を表示します。

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

ref カーソルパラメーターを使用している場合は、EDBParameter() に EDBDbType.RefCursor 型をバインドする必要があります。

第14章 Using plugins

EDB .Net ドライバープラグインは、.Net では利用できないさまざまなデータ型の拡張機能をサポートしています。利用可能なさまざまなプラグインがサポートしています：

- GeoJSON
- Json.NET
- LegacyPostGIS
- NetTopologySuite
- NodaTime
- RawPostGIS

プラグインは、空間、データ/時間、および JSON タイプの使用をサポートしています。以下は、これらのプラグインでサポートされているフレームワークとデータプロバイダーのインストールパスです。

14.1 GeoJSON

.NET Standard 2.0 で GeoJSON プラグインを使用している場合、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

GeoJSON プラグインの使用の詳細については、[Npgsql documentation](#) を参照してください。

14.2 Json.NET

.NET Standard 2.0 で Json.NET プラグインを使用している場合、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

Json.NET プラグインの使用の詳細については、[Npgsql documentation](#) を参照してください。

14.3 LegacyPostGIS

.Net Standard 2.0 で LegacyPostGIS プラグインを使用している場合、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

LegacyPostGIS プラグインの使用の詳細については、[Npgsql documentation](#) を参照してください。

14.4 NetTopologySuite

.Net Standard 2.0 で NetTopologySuite プラグインを使用している場合、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

NetTopologySuite タイプのプラグインの使用の詳細については、[Npgsql documentation](#) を参照してください。

14.5 NodaTime

.Net Standard 2.0 で NodaTime プラグインを使用している場合、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

NodaTime プラグインの使用の詳細については、[Npgsql documentation](#) を参照してください。

14.6 RawPostGIS

.Net Standard 2.0 で RawPostGIS タイプのプラグインを使用している場合、データプロバイダーのインストールパスは次のとおりです。

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

次の共有ライブラリファイルが必要です。

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

RawPostGIS プラグインの使用の詳細については、[documentation](#) を参照してください。

第15章 Using object types in .NET

SQL CREATE TYPE コマンドは、EDB Postgres Advanced Server データベースに保存されるユーザー定義のオブジェクトタイプを作成します。

その後、これらのユーザー定義型を SPL プロシージャ、SPL ファンクション、および .NET プログラムで参照できます。

基本的なオブジェクトタイプは、CREATE TYPE AS OBJECT コマンドとオプションで CREATE TYPE BODY コマンドを使用して作成されます。

15.1 オブジェクトタイプの使用

オブジェクトタイプを使用するには、最初に EDB Postgres Advanced Server データベースにオブジェクトタイプを作成する必要があります。オブジェクトタイプ `addr_object_type` は、アドレスの属性を定義します。

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

オブジェクトタイプ `emp_obj_typ` は、従業員の属性を定義します。これらの属性の 1 つは、前述のオブジェクト型 `ADDR_OBJECT_TYPE` です。オブジェクトタイプの本文には、従業員情報を表示するメソッドが含まれています。

```
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
```

(次のページに続く)

(前のページからの続き)

```

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;

```

次のリストは、これらのユーザー定義オブジェクトタイプを使用する完全な.NET プログラムです。

```

namespace ObjectTypesSample
{
    class Program
    {
        static void Main(string[] args)
        {
            EDBConnection.GlobalTypeMapper.MapComposite<addr_object_type>(
↪ "enterprisedb.addr_object_type");
            EDBConnection.GlobalTypeMapper.MapComposite<emp_obj_typ>("enterprisedb.
↪ emp_obj_typ");
            EDBConnection conn = new EDBConnection("Server=localhost;Port=5444;
↪ database=test;User ID=enterprisedb;password=<PASSWORD>");

            try
            {
                conn.Open();

                EDBCommand cmd = new EDBCommand("emp_obj_typ.display_emp", conn);

                cmd.CommandType = System.Data.CommandType.StoredProcedure;
                EDBCommandBuilder.DeriveParameters(cmd);

                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",

```

(次のページに続く)

(前のページからの続き)

```

        addr = address
    };
    cmd.Parameters[0].Value = emp;
    cmd.Prepare();
    cmd.ExecuteNonQuery();

    emp_obj_typ 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;
}
}

```

次の.NET タイプは、 EDB Postgres Advanced Server のタイプにマッピングされます。

```
public class addr_object_type
```

(次のページに続く)

(前のページからの続き)

```
{
    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;
}
```

EDBConnection.GlobalTypeMapper.MapComposite への呼び出しは、.NET タイプを EDB Postgres Advanced Server タイプにマップします。

```
EDBConnection.GlobalTypeMapper.MapComposite<addr_object_type>("enterprisedb.addr_
↪object_type");
EDBConnection.GlobalTypeMapper.MapComposite<emp_obj_typ>("enterprisedb.emp_obj_typ
↪");
```

EDBCommandBuilder.DeriveParameters() の呼び出しは、ストアドプロシージャのパラメーター情報を取得します。これにより、パラメーター値を設定してストアドプロシージャを呼び出すことができます。

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

パラメーターの値は、.NET タイプのオブジェクトを作成し、それをパラメーターの Value プロパティに割り当てることにより設定します。

```
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;
```

`cmd.ExecuteNonQuery()` を呼び出すと、`display_emp()` メソッドの呼び出しが実行されます。

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


第16章 Scram compatibility

EDB .NET ドライバーは、EDB Postgres Advanced Server バージョン 10 以降の SCRAM-SHA-256 サポートを提供します。このサポートは、EDB .NET 4.0.2.1 リリース以降で利用できます。

第17章 EDB .NET Connector logging

EDB .NET Connector は、アプリケーションで使用するときに.NET Connector の問題を解決するのに役立つログの使用をサポートしています。コネクタは、ロギングに `EnterpriseDB.EDBClient.Logging` 名前空間のクラスを使用します。

17.1 標準エラーへのコンソールロギング

`EnterpriseDB.EDBClient.Logging.ConsoleLoggingProvider` は、STDERR にエラーメッセージを出力するクラスです。このクラスを使用するには、EDB Postgres Advanced Server .NET Connector API を使用する前にアプリケーションに次の行を含めます。

```
EnterpriseDB.EDBClient.Logging.EDBLogManager.Provider = new EnterpriseDB.EDBClient.  
Logging.ConsoleLoggingProvider(EDBLogLevel.Debug, true, true);
```

次のログレベルを使用できます。

- トレース
- デバッグ
- 情報
- 警告する
- エラー
- 致命的

17.2 カスタムロガーを書く

コンソールロギングプロバイダーがニーズを満たさない場合、`EnterpriseDB.EDBClient.Logging.IEDBLoggingProvider` インターフェイスを実装し、`EnterpriseDB.EDBClient.Logging.EDBLogger` クラスを拡張することにより、たとえば、ログをファイルに書き込むことにより、カスタムロガーを作成できます。以下は、カスタムロガーを作成する方法の簡単な例です。

```
public class MyLoggingProvider : IEDBLoggingProvider  
{  
    string _logFile;  
    readonly EDBLogLevel _minLevel;  
    readonly bool _printLevel;
```

(次のページに続く)

```

        readonly bool _printConnectorId;

        public MyLoggingProvider(string logFile, EDBLogLevel minLevel = EDBLogLevel.
→Info, bool printLevel = false, bool printConnectorId = false)
        {
            _logFile = logFile;
            _minLevel = minLevel;
            _printLevel = printLevel;
            _printConnectorId = printConnectorId;
        }

        public EDBLogger CreateLogger(string name)
        {
            return new MyLogger(_logFile, _minLevel, _printLevel, _printConnectorId);
        }
    }

class MyLogger : EDBLogger
{
    string _logFile;
    readonly EDBLogLevel _minLevel;
    readonly bool _printLevel;
    readonly bool _printConnectorId;

    internal MyLogger(string logFile, EDBLogLevel minLevel, bool printLevel, bool
→printConnectorId)
    {
        _logFile = logFile;
        _minLevel = minLevel;
        _printLevel = printLevel;
        _printConnectorId = printConnectorId;
    }

    public override bool IsEnabled(EDBLogLevel level) => level >= _minLevel;

    public override void Log(EDBLogLevel level, int connectorId, string msg,
→Exception exception = null)
    {
        if (!IsEnabled(level))
            return;

        using (StreamWriter writer = new StreamWriter(_logFile, true))
        {
            var sb = new StringBuilder();
            if (_printLevel)

```

(前のページからの続き)

```
        {
            sb.Append(level.ToString().ToUpper());
            sb.Append( );
        }

        if (_printConnectorId && connectorId != 0)
        {
            sb.Append("[");
            sb.Append(connectorId);
            sb.Append("] ");
        }

        sb.AppendLine(msg);

        if (exception != null)
            sb.AppendLine(exception.ToString());

        writer.Write(sb.ToString());
    }

}
```

このカスタムロガーを使用するには、EDB .NET Connector API を使用する前にアプリケーションに次の行を追加します。

```
EDBLogManager.Provider = new MyLoggingProvider(filepath, EDBLogLevel.Debug, true, true);
```


第18章 API reference

API の使用については、[Npgsql documentation](#) を参照してください。

使用上の注意：

- API を使用する場合、Npgsql への参照を `EnterpriseDB.EDBClient` に置き換えます。
- クラスを参照する場合は、Npgsql を EDB に置き換えます。たとえば、`NpgsqlBinaryExporter` クラスの代わりに `EDBBinaryExporter` クラスを使用します。