



.Net Connector

リリース ***4.0.10.1***

EDB Postgres™ Advanced Server .NET Connector Guide

2019 年 10 月 09 日

目次

第 1 章	要件の概要	2
1.1	サポートされているサーバーのバージョン	2
1.2	サポートされているプラットフォーム	2
第 2 章	Advanced Server .NET コネクタ-概要	4
2.1	.NET クラス階層	4
第 3 章	.NET コネクタのインストールと構成	6
3.1	.NET コネクタのインストール	6
3.2	.NET コネクタの構成	12
3.2.1	ライブラリファイルの参照	13
3.2.2	.NET Framework のセットアップ	14
3.2.3	Entity Framework 5/6	18
3.2.4	Enterprise Studio VSIX for Visual Studio 2015/2017/2019	20
第 4 章	.NET コネクタの使用	24
第 5 章	データベース接続を開く	25
5.1	接続文字列パラメーター	25
5.2	データベース接続を開く	28
5.3	データベース接続を開く	30
5.4	データベース接続を開く	32
第 6 章	データベースレコードの取得	34
6.1	単一のデータベースレコードの取得	37
第 7 章	パラメータ化されたクエリ	39
第 8 章	データベースへのレコードの挿入	41

第 9 章	データベースのレコードを削除する	43
第 10 章	.NET アプリケーションでの SPL ストアドプロシージャの使用	45
10.1	例-パラメータなしでストアドプロシージャを実行する	46
10.2	例-IN パラメータを使用したストアドプロシージャの実行	48
10.3	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	52
第 11 章	アドバンスドキューイングの使用	56
11.1	メッセージをエンキューまたはデキューする	57
11.1.1	サービス側のセットアップ	57
11.1.2	クライアント側のサンプル	58
11.2	EDBAQ クラス	65
第 12 章	.NET アプリケーションでの参照カーソルの使用	69
第 13 章	プラグインを使用する	72
13.1	GeoJSON	73
13.2	Json.NET	74
13.3	LegacyPostGIS	75
13.4	NetTopologySuite	76
13.5	野田タイム	77
13.6	RawPostGIS	78
第 14 章	API リファレンス	79
第 15 章	おわりに	80
	索引	81

EDB Postgres™Advanced Server（Advanced Server）と共に配布される EDB Postgres™Advanced Server .NET コネクタは、.NET クライアントアプリケーションと Advanced Server データベースサーバー間の接続を提供します。このガイドでは、インストール手順、使用手順、および Advanced Server .NET Connector の機能を示す例について説明します。

- Advanced Server のインスタンスに接続する方法。
- Advanced Server データベースから情報を取得する方法。
- Advanced Server データベースに保存されている情報を更新する方法。

このドキュメントは、C #と.NET の両方の実用的な知識があることを前提としています。 Advanced Server .NET Connector の機能は、Npgsql オープンソースプロジェクトのコア機能の上に構築されています。 * Npgsql User's Manual *はオンラインで利用可能です<<http://www.npgsql.org/doc/index.html>> ‘_。

Advanced Server .NET Connector“4.0.10.1”を作成するために、次の機能が追加されました。

- EDB .NET コネクタが EDB Postgres Advanced Server 12 をサポートするようになりました。
- EDB .NET コネクタは、Windows Server 2019 プラットフォームでもサポートされるようになりました。
- アップストリームコミュニティドライババージョン 4.0.10 と統合されました。
- Visual Studio 2019 の VSIX をサポートします。

要件の概要

次のセクションでは、Advanced Server .NET Connector でサポートされているプラットフォームについて詳しく説明します。

1.1 サポートされているサーバーのバージョン

Advanced Server .NET コネクタは、Advanced Server バージョン 9.4 以上で認定されています。

1.2 サポートされているプラットフォーム

Advanced Server .NET Connector の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 Server 2016
- Windows Server 2012 R2

32 ビット Windows :

- Windows 10

- Windows 8
- Windows 7

Advanced Server .NET コネクタ-概要

Advanced Server .NET コネクタは、クライアントアプリケーションが 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し、クライアントアプリケーションの最適なパフォーマンス、広範な機能、および Advanced Server 機能へのアクセスを可能にします。

.NET コネクタは、.NET Framework バージョン 4.0 および 4.5.1、Entity Framework 5/6、および .Net Standard 2.0 をサポートしています。

2.1 .NET クラス階層

.NET クラス階層には、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オブジェクトを作成するために使用できる多くの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 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 Advanced Server データベースへの接続方法を指示する “`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 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、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 行ずつ順方向にステップスルーできます。

EDBDataAdapter

“EDBDataAdapter”オブジェクトは、結果セットを 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、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。

.NET コネクタのインストールと構成

この章では、Advanced Server .NET Connector のインストールおよび構成方法について説明します。

3.1 .NET コネクタのインストール

EnterpriseDB .NET Connector インストーラーを使用できます（EnterpriseDB Web サイトから入手可能） <<https://www.enterprisedb.com/advanced-downloads>> ‘_）.NET コネクタをシステムに追加します。インストーラーをダウンロードした後、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“管理者として実行”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“セットアップ”ウィンドウに進みます。

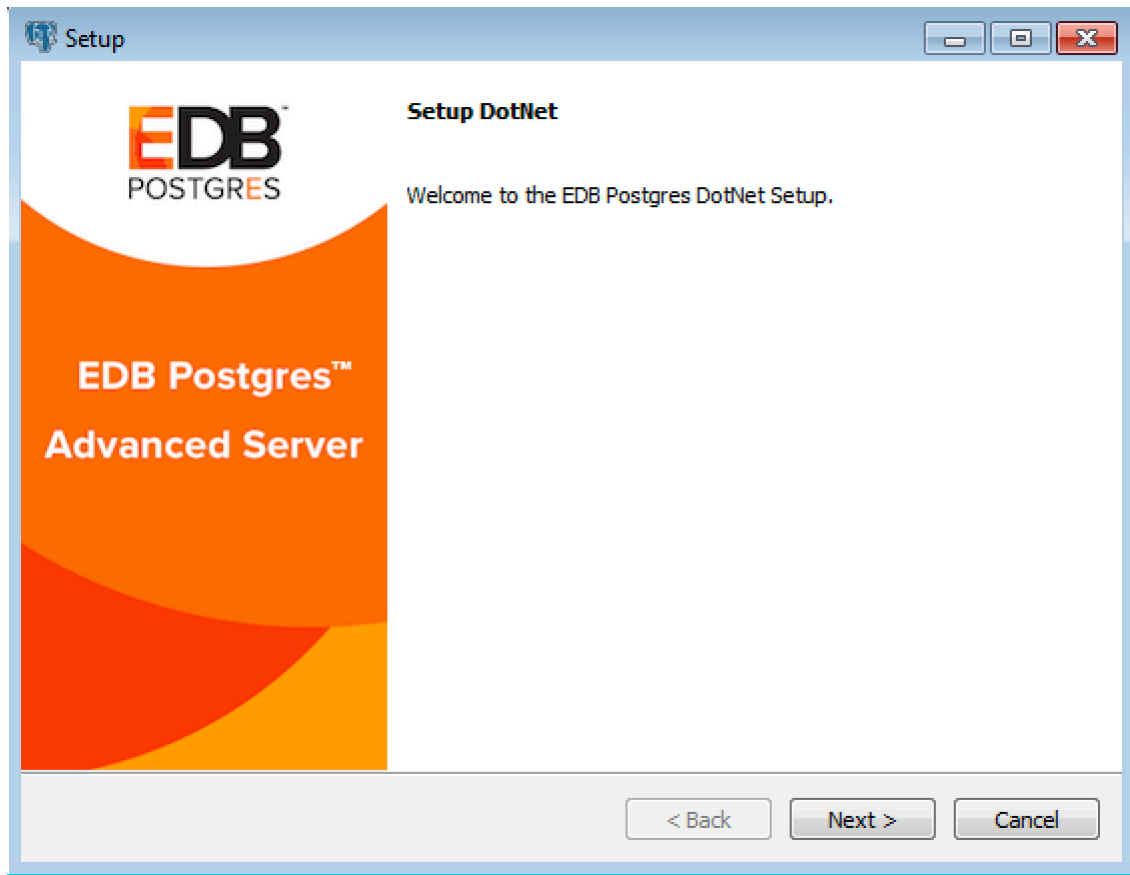


図 1 The .NET Connector Installation wizard

“次へ”をクリックして続行します。

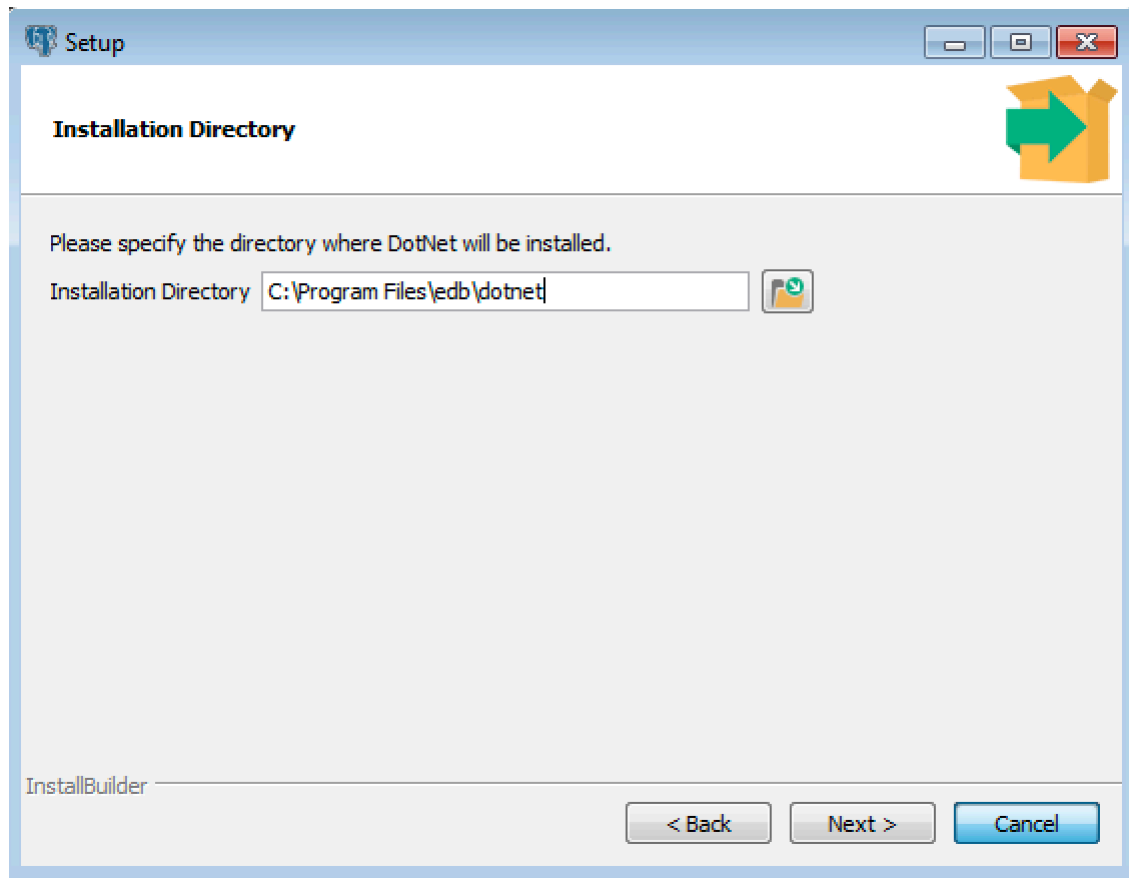


図 2 インストールダイアログ

“インストールディレクトリ”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、“次”をクリックして続行します。

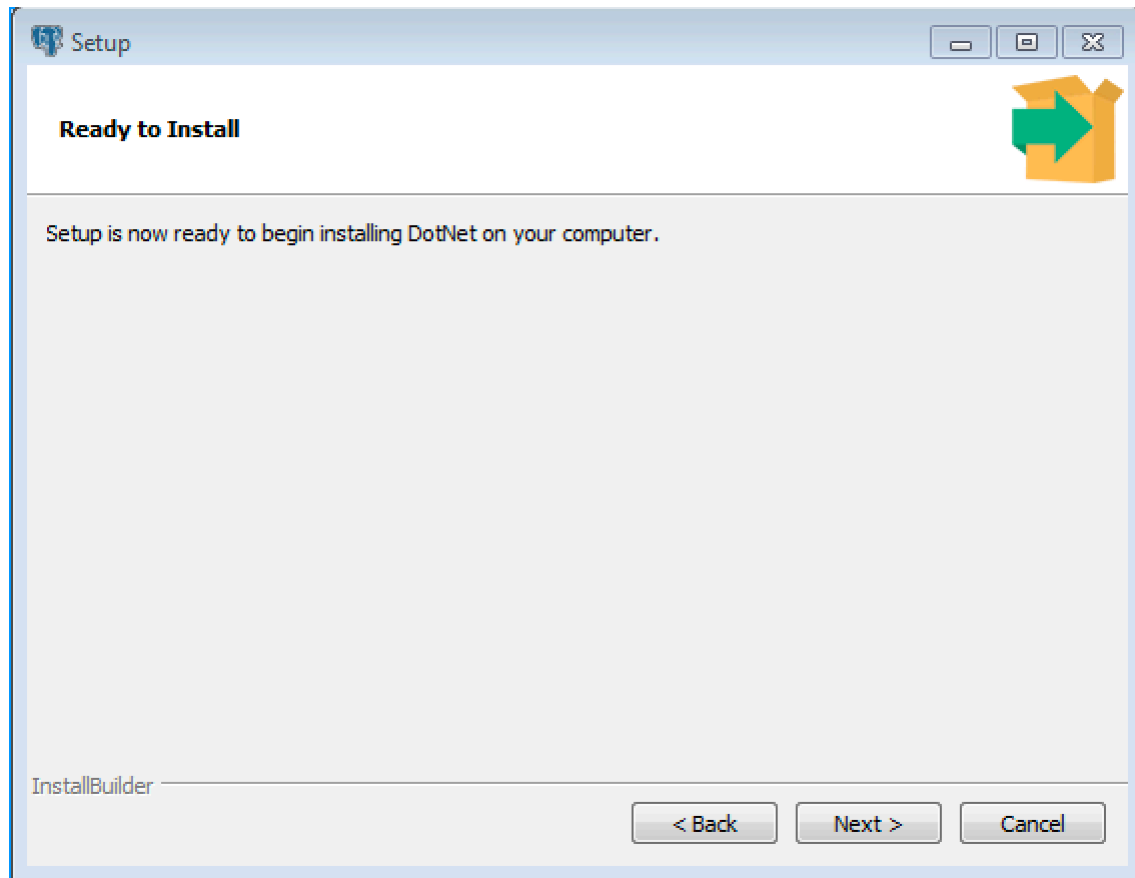


図 3 インストールの準備ダイアログ

“インストールの準備完了”ダイアログで“次へ”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がインストールウィザードの進行状況を確認します。

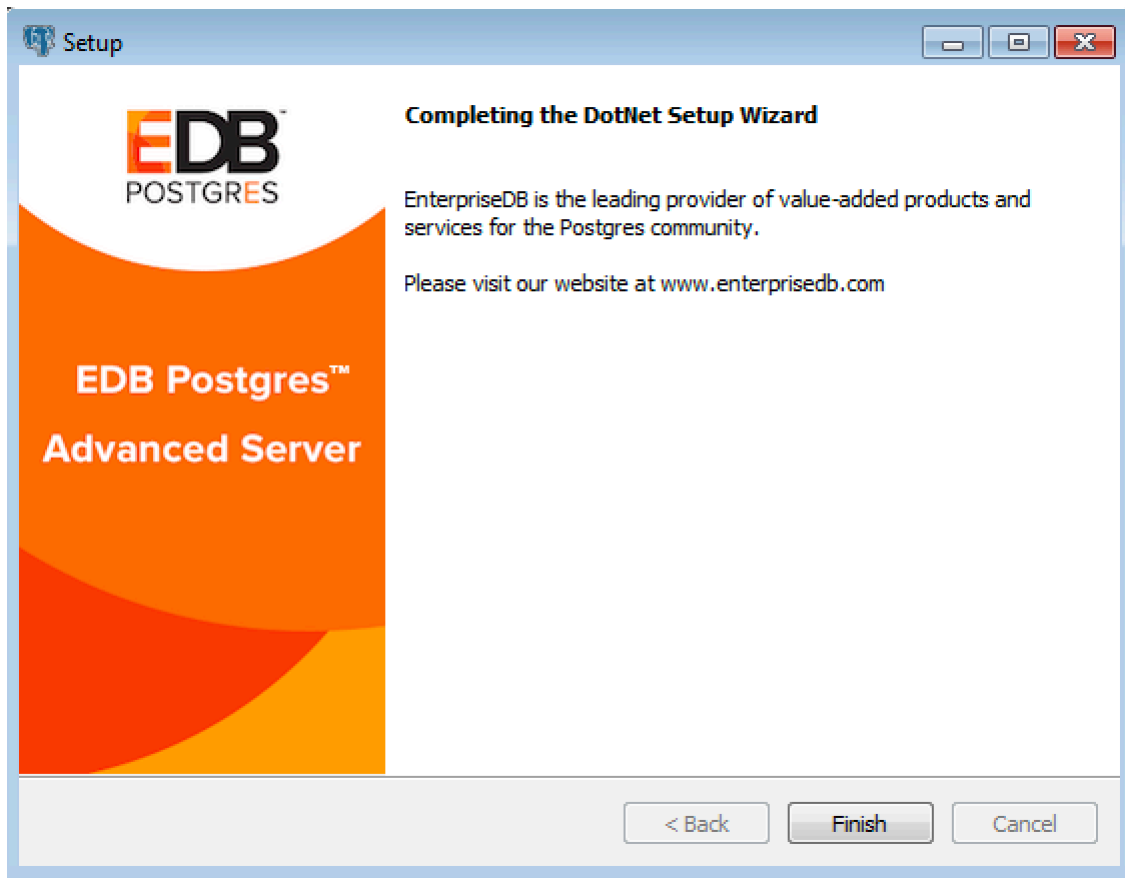


図 4 インストールが完了しました

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、“終了”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 インストールで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 “アプリ” メニューから StackBuilder Plus を選択します。

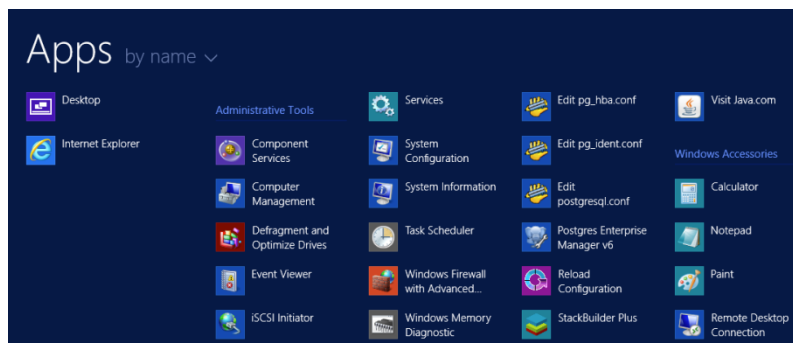


図 5 Starting StackBuilder Plus

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 “データベースドライバ” ノードから “EnterpriseDB.Net Connector” オプションを選択します。

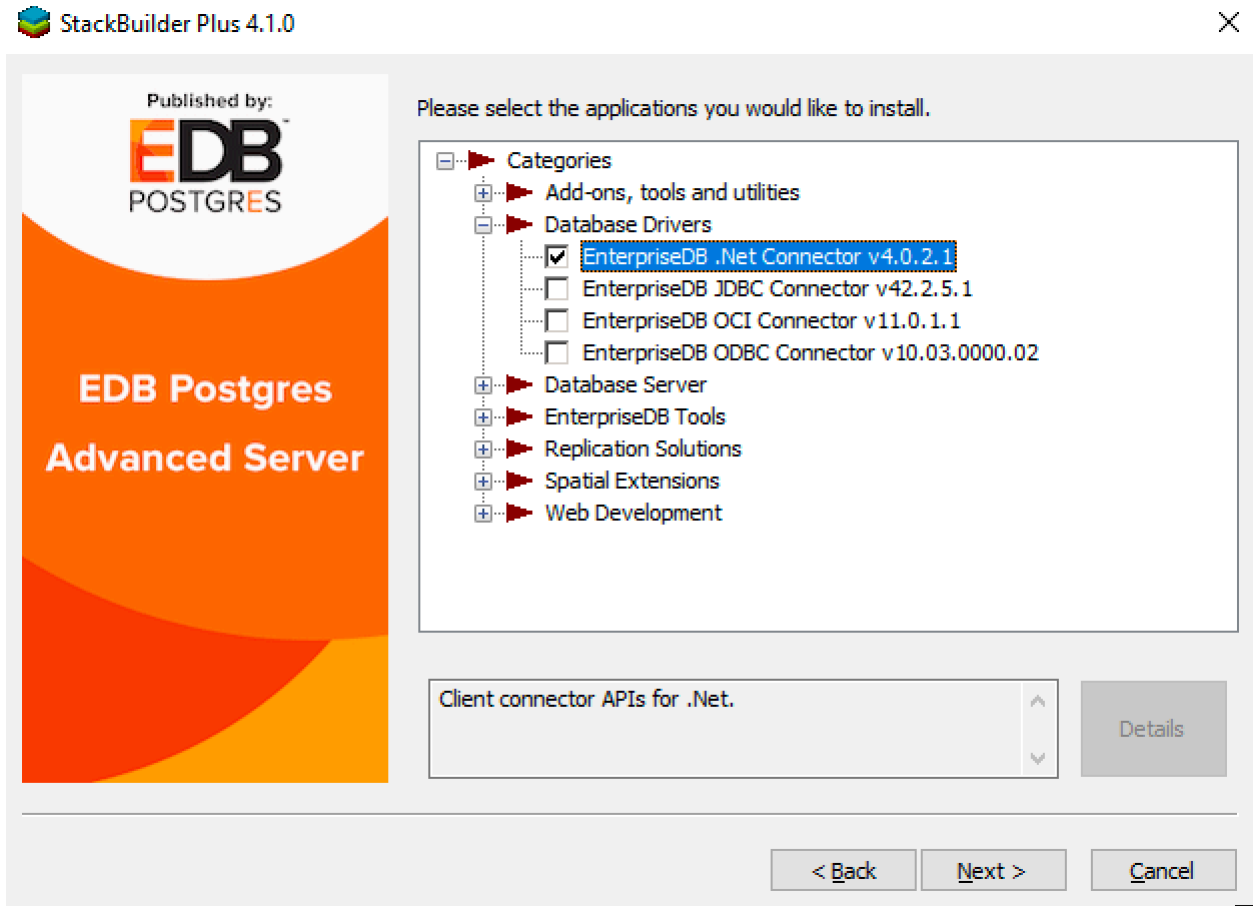


図 6 コネクタインストーラーの選択

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

3.2 .NET コネクタの構成

.NET コネクタの構成については、環境固有の次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 : ref : ‘一般的な構成情報<referencing_the_library_files>’すべてのコンポーネントに適用可能。
- 。NET Framework 4.0。 : ref : ‘.NET Framework 4.0 で使用するための設定手順<framework_setup_4>’。
- 。NET Framework 4.5。 : ref : ‘.NET Framework 4.5 で使用するための設定手順<framework_setup_4_5>’。
- 。NET Framework 4.5.1。 : ref : ‘.NET Framework 4.5.1 で使用するための設定手順<framework_setup_4_5_1>’。
- 。NET Standard 2.0。 : ref : ‘.NET Standard 2.0 で使用するための設定手順<standard_setup_2>’。
- Entity Framework 5/6。 : ref : ‘Entity Framework で使用するための設定手順<entity_setup_5_6>’。
- EnterpriseDB VSIX。 : ref : ‘EnterpriseDB VSIX で使用するための設定手順<vsix_setup>’。

3.2.1 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. “ソリューションエクスプローラー”でプロジェクトを選択します。
2. “プロジェクト”メニューから“参照の追加”を選択します。
3. “参照の追加”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。

Advanced Server .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
```

Entity Framework 6 を使用している場合、次の追加の名前空間が必要です。

```
EntityFramework6.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タイプ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。

```
EnterpriseDB.EDBClient を使用;
```

3.2.2 .NET Framework のセットアップ

以下のセクションでは、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。

.NET Framework 4.0

.NET Framework バージョン 4.0 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 / net40 /
```

次の共有ライブラリファイルが必要です。

```
EDBDataProvider.2.0.2.dll
```

```
Mono.Security.dll
```

: ref: [ライブラリファイルの参照を参照してください](#)。 <referencing_the_library_files> [ライブラリファイルの参照に関する情報](#)。

使用するアプリケーションのタイプによっては、名前空間をソースコードにインポートする必要がある場合があります（: ref: [ライブラリファイルの参照を参照](#)） <referencing_the_library_files> ）。

.NET Framework 4.5

.NET Framework バージョン 4.5 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 / net45 /
```

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
```

また、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。

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

: ref : ‘ライブラリファイルの参照を参照してください。 <referencing_the_library_files> ‘ライブラリファイルの参照に関する情報。

使用するアプリケーションのタイプによっては、名前空間をソースコードにインポートする必要がある場合があります (: ref : ‘ライブラリファイルの参照を参照) <referencing_the_library_files> ‘)。

.NET Framework 4.5.1

.NET Framework バージョン 4.5.1 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 / net451 /
```

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
```

また、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。

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

: ref: ‘ライブラリファイルの参照を参照してください。 <referencing_the_library_files> ‘ライブラリファイルの参照に関する情報。

使用するアプリケーションのタイプによっては、名前空間をソースコードにインポートする必要がある場合があります（: ref: ‘ライブラリファイルの参照を参照） <referencing_the_library_files> ‘）。

.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
```

また、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。

```
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
```

: ref: ‘ライブラリファイルの参照を参照してください。 <referencing_the_library_files> ‘ライブラリファイルに関する情報。

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る場合があります（参照：ライブラリファイルの参照） <referencing_the_library_files> ‘)。

3.2.3 Entity Framework 5/6

Entity Framework で使用するために.NET コネクタをセットアップするには、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 / EF /
```

次の共有ライブラリファイルが必要です。

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

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

注釈: Entity Framework は、“net45”および“net451”サブディレクトリにある“EnterpriseDB.EDBClient.dll”で使えます。

: ref: ‘ライブラリファイルの参照を参照してください。 <referencing_the_library_files> ‘ライブラリファイルの参照に関する情報。

を追加<DbProviderFactories>““app.config”ファイルへの Postgres 用の “ADO.NET”ドライバーのエントリ。次のエントリを追加します。

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

プロジェクトの“app.config”ファイルで、EntityFramework / providers タグの下にプロバイダーサービスの次のエントリを追加します。

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

以下は“app.config”ファイルの例です：

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

(次のページに続く)

(前のページからの続き)

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

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

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

注釈: の同じエントリ<providers>そして<DbProviderFactories>“web.config”ファイルと “app.config”ファイルに有効です。

使用してい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る場合があります（: ref: ‘ライブラリファイルの参照を参照） <referencing_the_library_files> ‘）。

Entity Framework の使用情報については、Microsoft のドキュメントを参照してください。

3.2.4 Enterprise Studio VSIX for Visual Studio 2015/2017/2019

“EDB Data Designer Extensibility Provider (EnterpriseDB VSIX)”は、Advanced Server データベースアクセスを Visual Studio に統合するコンポーネントであり、Visual Studio に統合された機能を提供します。

Visual Studio のサーバーエクスプローラーから Advanced Server に接続したり、既存のデータベースからモデルを作成したりすることができます。したがって、Visual Studio の機能が必要な場合は、EnterpriseDB VSIX を使用する必要があります。

EnterpriseDB VSIX は次のディレクトリにあります。

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

上記の場所で利用可能なファイルは次のとおりです。

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

Visual Studio 2015/2017/2019 のインストールと構成

以下は、EnterpriseDB VSIX をインストールおよび構成する手順です。

ステップ 1：次の場所にある “EnterpriseDB.vsix” インストーラーを使用して、Visual Studio の目的のバージョンに EnterpriseDB VSIX をインストールします。

```
C : / Program Files / edb / dotnet / vsix / EnterpriseDB.vsix
```

VSIX の以前のバージョンが既にインストールされている場合は、競合を避けるためにそれらをアンインストールすることを強くお勧めします。

グローバルアセンブリキャッシュ（GAC）に “EnterpriseDB.EDBClient” を含める必要はなくなりました。

ステップ 2：Visual Studio を再起動し、“ツール > 拡張機能と更新…” メニューから EnterpriseDB 拡張機能がインストールされていることを確認します。

ステップ 3：次の場所から Visual Studio Developers コマンドラインを使用して、“gacutil” ユーティリティを使用してグローバルアセンブリキャッシュ（GAC）に “System.ValueTuple.dll” アセンブリを追加します。

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

例：

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

ステップ 4：サーバーエクスプローラーから、“データ接続” を右クリックし、“接続を追加” をクリックして、“Enterisedb Postgres データベース” データソースが利用可能であることを確認します。

モデル優先およびデータベース優先使用

ステップ 1: Visual Studio Developers コマンドラインを使用して、“gacutil”ユーティリティを使用して、グローバルアセンブリキャッシュ (GAC) に “EntityFramework5.EnterpriseDB.EDBClient.dll” アセンブリを追加します。

例:

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

ステップ 2: を追加<DbProviderFactories> “machine.config” ファイル内の EDB Postgres の ADO.NET ドライバーのエントリ。エントリは次のとおりです。

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

属性と値のペアでは、二重引用符で囲まれた文字列に余分な空白文字を含めることはできませんが、1 行で構成する必要があります。このセクションに示されている例は、明確にするために複数の行に分割されています場合がありますが、実際には次のような 1 行で構成する必要があります。

```
description = "EnterpriseDB PostgreSQL 用の .NET データプロバイダー";
```

64 ビット Windows の場合、“machine.config” ファイルは次の場所にあります。

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

32 ビット Windows の場合、“machine.config” ファイルは次の場所にあります。

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

ステップ 3: 次の例のように、DDL 生成テンプレート “SSDLToPgSQL.tt” を Visual Studio の “EntityFramework Tools / DBGen” フォルダーに配置します。

```
C:\Program Files (x86)\Microsoft Visual Studio 14.
0\Common7\IDE\Extensions\Microsoft\EntityFramework Tools\DBGen\
```

注釈: EDMX ファイルのプロパティでこのテンプレート “SSDLToPgSQL.tt” を選択します。

ステップ 4: ファイル “EnterpriseDB.EDBClient.dll” および “EntityFramework6.EnterpriseDB.EDBClient.dll” をプロジェクト参照に追加します。 :ref: ライブラリファイルの参照を参照してください。 <referencing_the_library_files> ライブラリファイルの参照に関する情報。

ステップ 5: プロジェクトの “app.config” ファイルで、EntityFramework / providers タグの下にプロバイダーサービスの次のエントリを追加します。

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

以下は “app.config” ファイルの例です。

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

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

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

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

.NET コネクタの使用

以下のセクションでは、.NET アプリケーションが Advanced Server データベースに接続して対話できる Advanced Server .NET Connector によって提供される EDB オブジェクトクラスの使用を示す例を示します。

このガイドの例を使用するには、アプリケーションのコンパイル済みフォームと同じディレクトリに .NET ライブラリファイルを配置します。すべての例は 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は、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）にも適用できます。

サンプルコードと同じディレクトリに次の “web.config” ファイルを作成して保存してください。例では、この設定ファイルの “DB_CONN_STRING” キーを使用して、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 Advanced Server 接続文字列は “web.config” ファイルに保存されます。ASP.NET を使用しないアプリケーションを作成する場合は、アプリケーション構成ファイル (“app.config” など) で接続情報を提供します。

第 5

データベース接続を開く

“EDBConnection”オブジェクトは、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。Advanced Server データベースに保存されたデータにアクセスする前に、“EDBConnection”を作成して開く必要があります。

次の例は、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。あなたがする必要があります：

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すことで、“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。

5.1 接続文字列パラメーター

有効な接続文字列では、Advanced Server インスタンスの場所と認証情報を指定する必要があります。接続を開く前に、接続文字列を指定する必要があります。接続文字列には次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
- Advanced Server データベースの名前
- 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

次のパラメーターが接続文字列に含まれている場合があります。

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まで待機する時間の長さ（秒単位）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待機する時間の長さ（秒単位）を指定します。デフォルト値は “15” です。

データベース

“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。データベース名が指定されていない場合、データベース名はデフォルトで接続ユーザーの名前になります。

エンコード

“Encoding”パラメーターは廃止されました。パラメータは常に文字列 `unicode` を返し、設定の試行を静かに無視します。

統合セキュリティ

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

MaxPoolSize

“MaxPoolSize”は、プールが指定された接続数を超えると、プールされた接続を破棄するよう “EDBConnection”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 “EDBConnection”に指示します。デフォルト値は “1” です。

パスワード

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

プール

デフォルトでは、接続プーリングを有効にするために “Pooling”は “true” に設定されます。接続プールを無効にするには、“false”の値を指定します。

ポート

“Port”パラメーターは、アプリケーションが接続するポートを指定します。

プロトコル

使用する特定のプロトコルバージョン（自動ではなく）。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とパブリックスキーマに変更します。

サーバー

Advanced Server ホストの名前または IP アドレス。

SSL

デフォルトでは、“SSL”は “false”に設定されています。安全な接続を試みるには “true”の値を指定します。

sslmode

“sslmode”を使用して、SSL 接続制御設定を指定します。“sslmode”は次のいずれかです：

prefer-可能であれば SSL を使用します。

require-SSL 接続を確立できない場合に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 接続を試みません。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。

タイムアウト

“タイムアウト”は、開いている接続を待機する時間の長さ（秒単位）を指定します。デフォルト値は “15”です。

ユーザー ID

“User Id”パラメーターは、接続に使用されるユーザー名を指定します。

5.2 データベース接続を開く

次の例は、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ています。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（:ref: ‘章-.Net コネクターの使用を参照） <using_the_net_connector> ‘接続情報の概要については、:ref: ‘セクション-接続文字列パラメーターも参照してください。 <connection_string_parameters> ‘接続パラメータ用）。

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

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

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

    EDBConnection conn = new EDBConnection(strConnectionString);

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

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

    finally
    {
        conn.Close();
    }
}

</script>
```

接続が成功すると、ブラウザに次が表示されます。

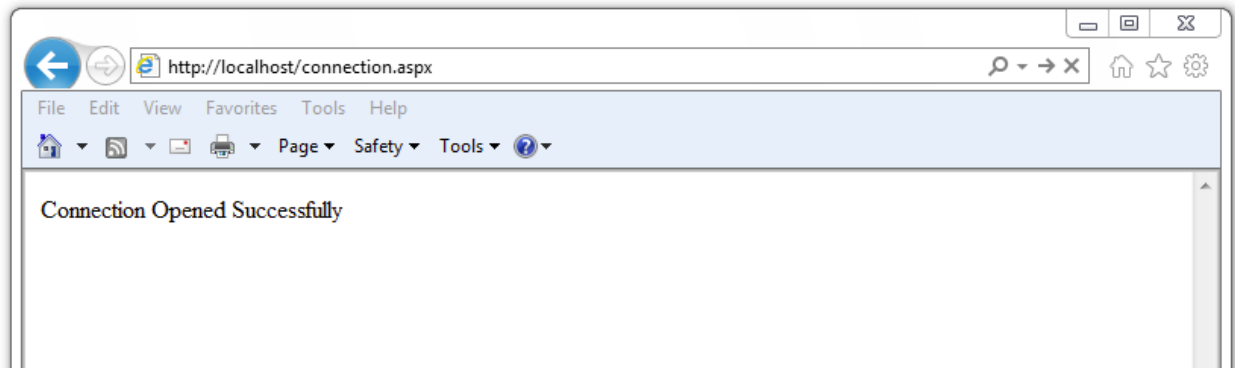


図 1 接続が正常にオープンしました

5.3 データベース接続を開く

次の例では、コンソールベースのアプリケーションを使用して 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書く前に、データベースへの接続文字列を保存する ‘‘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のに便利です。

```
<?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>
```

選択したテキストエディターを使用して、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
    {
        static void Main(string[] args)
        {
            string strConnectionString = ConfigurationSettings.AppSettings
                ["DB_CONN_STRING"];

            EDBConnection conn = new EDBConnection(strConnectionString);

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

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

            finally
            {

```

(次のページに続く)

(前のページからの続き)

```
        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:EDBDataProvider.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。実行されると、コンソールは次のことを確認する必要があります。

接続が正常に開かれました

5.4 データベース接続を開く

次の例は、.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く方法を示しています。この例を使用するには、次のコードを ‘WinForm-Example.cs’ としてライブラリファイルを含むディレクトリに保存します。

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

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

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

            Application.Run(frmMain);
        }

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

(次のページに続く)

(前のページからの続き)

```
}  
  
}  
  
}  
  
}
```

次のコマンドでファイルをコンパイルする前に、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する必要があることに注意してください。

```
csc /r:EDBDataProvider.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同じフォルダー内に“WinForm.exe”ファイルを生成する必要があります。実行可能ファイルを呼び出すと、次が表示されます。

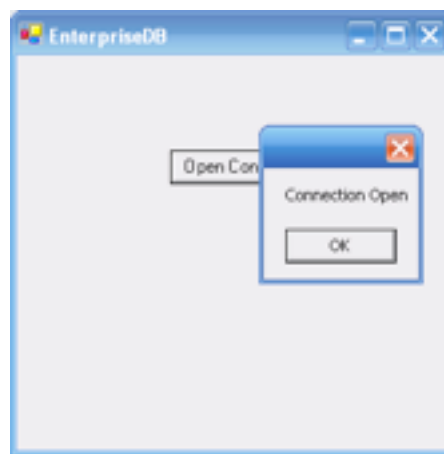


図 2 正常な接続

データベースレコードの取得

“SELECT”ステートメントを使用して、“SELECT”コマンドを介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、以下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。
- “EDBData”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
- 結果を表示するか、“EDBDataReader”を何らかのコントロールにバインドする “EDBDataReader”をループします。

“EDBDataReader”オブジェクトは、一度に 1 レコードずつ提示される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ストリームを表します。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. Advanced Server 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”メソッドを呼び出すことで “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。

5. “EDBDataReader” オブジェクトの内容をループして、クエリによって返されたレコードを “WHILE” ループ内に表示します。

“Read ()” メソッドは次のレコードに進み (レコードが存在する場合)、レコードが存在する場合は “true” を返し、“EDBDataReader” が到達したことを示す場合は “false” を返します結果セットの終わり。

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

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

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

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

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

    }

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

サンプルコードを実行するには、デフォルトの **Web** ルートディレクトリの次の名前のファイルにコードを保存します。

```
selectEmployees.aspx
```

プログラムを呼び出すには、**Web** ブラウザーを開き、次を参照します。

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

6.1 単一のデータベースレコードの取得

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" %>

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

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

        cmd.CommandType = CommandType.Text;

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

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

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

次の名前の Web ルートディレクトリ内のファイルにサンプルコードを保存します。

```
selectscalar.aspx
```

サンプルコードを呼び出すには、Web ブラウザーを開き、次を参照します。

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

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ることに注意してください。“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オブジェクトを整数値に変換する必要があります。

第 7

パラメータ化されたクエリ

“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で見つかった各マーカーの値を指定する必要があります。

パラメータ化されたクエリは、コードの記述時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して、パラメーター化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、各パラメーターに“value”を指定する必要があります。

この例は、従業員の給与を増やす“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 = ConfigurationSettings.AppSettings
        ["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);
```

(次のページに続く)

(前のページからの続き)

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

try {
    conn.Open();

    EDBCommand cmdUpdate = new EDBCommand(updateQuery, conn);

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

    cmdUpdate.Parameters[0].Value = 7788;

    cmdUpdate.ExecuteNonQuery();

    Response.Write("Record Updated");

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

次の名前の **Web** ルートディレクトリ内のファイルにサンプルコードを保存します。

updateSalary.aspx

サンプルコードを呼び出すには、**Web** ブラウザーを開き、次を参照します。

http:// localhost / updateSalary.aspx

第 8

データベースへのレコードの挿入

EDBCommand の ExecuteNonQuery () メソッドを使用して、INSERT コマンドで 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 (:) で始まる値は、インスタンス化され、値が割り当てられ、その後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る EDBParameters のプレースホルダーです。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。

この例は、新しい従業員を ‘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 = ConfigurationSettings.AppSettings
        ["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);

    try
    {
        conn.Open();
```

(次のページに続く)

(前のページからの続き)

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

EDBCommand cmdInsert = new EDBCommand(cmd, conn);

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

cmdInsert.Parameters[0].Value = 1234;

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

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

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

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

次の名前の **Web** ルートディレクトリ内のファイルにサンプルコードを保存します。

insertEmployee.aspx

サンプルコードを呼び出すには、**Web** ブラウザーを開き、次を参照します。

http:// localhost / insertEmployee.aspx

第 9

データベースのレコードを削除する

EDBCommand の ExecuteNonQuery () メソッドを使用して、DELETE ステートメントで 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” に格納されています。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。その後、コマンドは “ExecuteNonQuery ()” メソッドを使用して実行されます。次の例では、前の例で挿入された従業員を削除します。

```
<% @ 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 = ConfigurationSettings.AppSettings
        ["DB_CONN_STRING"];

    EDBConnection conn = new EDBConnection(strConnectionString);

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

    try
    {
        conn.Open();
```

(次のページに続く)

(前のページからの続き)

```
        EDBCommand deleteCommand = new EDBCommand(strDeleteQuery, conn);

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

        deleteCommand.Parameters[0].Value = 1234;

        deleteCommand.ExecuteNonQuery();

        Response.Write("Record Deleted");

    }

    catch(Exception exp)
    {

        Response.Write(exp.ToString());

    }

    finally
    {

        conn.Close();

    }
}
</script>
```

次の名前の Web ルートディレクトリ内のファイルにサンプルコードを保存します。

deleteEmployee.aspx

サンプルコードを呼び出すには、Web ブラウザーを開き、次を参照します。

http:// localhost / deleteEmployee.aspx

第 10

.NET アプリケーションでの SPL ストアドプロシージャの使用

次の 2 つの方法で、アプリケーションに SQL ステートメントを含めることができます。

- .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 ステートメントをストア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ドプロシージャを実行する。

場合によっては、ストアドプロシージャを使用すると、埋め込み SQL ステートメントよりも利点があります。ストアドプロシージャは、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およびループ構造をサポートします。

ストアドプロシージャを使用すると、パフォーマンスが大幅に向上することもわかります。ストアドプロシージャは、サーバー側で一度だけ解析、コンパイル、最適化する必要がありますが、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するたびに解析、コンパイル、最適化できます。

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

1. Advanced Server ホストで SPL ストア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. “Prepare ()” コマンド。
6. コマンドを実行します。

10.1 例-パラメータなしでストアードプロシージャを実行する

サンプルプロシージャは、部門 10 の名前を出力します。プロシージャはパラメータを受け取らず、パラメータを返しません。サンプルプロシージャを作成するには、EDB-PSQL を呼び出し、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
    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;
```

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。

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タイプを示すために使用されます。“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 値を使用して、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内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、またはテーブル名を含める必要があります。

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 = ConfigurationSettings.AppSettings
        ["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);
```

(次のページに続く)

(前のページからの続き)

```
try
{
    conn.Open();

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

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

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

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

次の名前の Web ルートディレクトリ内のファイルにサンプルコードを保存します。

storedProc.aspx

サンプルコードを呼び出すには、Web ブラウザーを開き、次を参照します。

http:// localhost / storedProc.aspx

10.2 例-IN パラメータ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、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)
        VALUES (MAX_EMPNO+1, pENAME, pJOB, pSAL, pCOMM, pDEPTNO, pMgr);
    CLOSE testcur;
END;
```

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。

ストアードプロシージャへの入力値の受け渡し

パラメータを含むストアードプロシージャの呼び出しは、パラメータなしのストアードプロシージャの実行に非常に似ています。主な違いは、パラメータ化されたストアードプロシージャを呼び出すときに、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 #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" >
```

(次のページに続く)

(前のページからの続き)

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

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

    try
    {
        conn.Open();

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

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

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

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

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

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

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

        cmdStoredProc.Prepare();
    }
}
```

(次のページに続く)

(前のページからの続き)

```
cmdStoredProc.ExecuteNonQuery();

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

Response.Write(empInfo);

}

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

</script>

</script>
```

次の名前の **Web** ルートディレクトリ内のファイルにサンプルコードを保存します。

storedProcInParam.aspx

サンプルコードを呼び出すには、**Web** ブラウザーを開き、次を参照します。

http:// localhost / storedProcInParam.aspx

この例では、“**Page_Load**”メソッドの本体が“**EDBConnection**”オブジェクトを宣言してインスタンス化します。サンプルは、ストアドプロシージャを実行するために必要なプロパティを持つ“**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” “ストアー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、ストアードプロシージャを実行します。ストアードプロシージャが実行された後、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。

10.3 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

前の例では、ストアードプロシージャに ‘IN’ パラメータを渡す方法を示しました。次の例は、‘IN’ 値を渡し、ストアードプロシージャから ‘OUT’ 値を返す方法を示しています。

ストアードプロシージャの作成

次のストアードプロシージャは、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、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;
```

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。

ストアー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、出力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て、‘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して、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、パラメーターの ‘方向’ を示す各パラメーターを宣言する必要があります。

ります (ParameterDirection.Input、ParameterDirection.Output、または “ParameterDirection.InputOutput”)。プロシージャを呼び出す前に、各 “IN” および “INOUT” パラメータに値を指定する必要があります。プロシージャが戻った後、“command.Parameters []” 配列から “OUT” および “INOUT” パラメータ値を取得できます。

次のコードリストは、“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 =
        ConfigurationSettings.AppSettings["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);

    try
    {

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

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

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

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

        command.Prepare();

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

(次のページに続く)

(前のページからの続き)

```
int fc = result.FieldCount;

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

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

</script>
```

次のコードリストは、“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 = ConfigurationSettings.AppSettings
        ["DB_CONN_STRING"];
    EDBConnection conn = new EDBConnection(strConnectionString);

    try
    {
        conn.Open();
        EDBCommand command = new EDBCommand("DEPT_SELECT
            (:pDEPTNO, :pDNAME, :pLOC)", conn);
```

(次のページに続く)

(前のページからの続き)

```
command.CommandType = CommandType.StoredProcedure;

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

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

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

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

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

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

アドバンスドキューイングの使用

EDB Postgres Advanced Server Advanced Queuing は、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。ユーザー定義のメッセージはキューに保存されます。キューのコレクションはキューテーブルに格納されます。キューテーブルを作成してから、それに依存するキューを作成する必要があります。

サーバー側では、“DBMS_AQADM”パッケージのプロシージャがメッセージキューとキューテーブルを作成および管理します。“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、“[ここをクリック](https://www.enterprisedb.com/docs/en/11.0/EPAS_BIP_Guide_v11/Database_Compatibility_for_Oracle_Developers_Built-in_Package_Guide.1.14.html#ID0E01HG0HA)”<https://www.enterprisedb.com/docs/en/11.0/EPAS_BIP_Guide_v11/Database_Compatibility_for_Oracle_Developers_Built-in_Package_Guide.1.14.html#ID0E01HG0HA>’_。

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/デキューします。

11.1 メッセージをエンキューまたはデキューする

11.1.1 サービス側のセットアップ

.NET アプリケーションで Advanced Queuing 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てから、データベースサーバーでキューを開始する必要があります。EDB-PSQL を呼び出して、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。

ユーザー定義タイプの作成

RAW データ型を指定するには、ユーザー定義型を作成する必要があります。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

キューテーブルは、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',
     comment => 'Message queue table');
END;
```

キューの作成

次の例は、テーブル “MSG_QUEUE_TABLE” 内に “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;
```

開始キュー

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;
```

11.1.2 クライアント側のサンプル

ユーザー定義タイプを作成し、続いてキュー表とキューを作成したら、キューを開始します。次に、EDB .Net ドライバーを使用してメッセージをキューに登録またはデキューできます。

メッセージをキューに入れる：

.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。

次のコードリストは、“Queue.enqueue”メソッドの使用法を示しています。

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

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

            private static EDBConnection GetConnection()
            {
```

(次のページに続く)

(前のページからの続き)

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

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

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

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

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

(次のページに続く)

(前のページからの続き)

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

メッセージをデキュー

.NET アプリケーションでメッセージをデキューするには、次の手順を実行する必要があります。

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、“EDBAQQueue”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。

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

(次のページに続く)

(前のページからの続き)

```
using EnterpriseDB.EDBClient;

namespace DequeueXML
{
    class MyXML
    {
        public string value { get; set; }
    }
    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 it's null so we don't incur overhead of an exception
            if (byteArray == null)
            {
                return string.Empty;
            }
            try
            {
                StringBuilder hex = new StringBuilder(byteArray.Length * 2);
                foreach (byte b in byteArray)
                {
                    hex.AppendFormat("{0:x2}", b);
                }

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

(次のページに続く)

(前のページからの続き)

```
public static void DequeueMsg(int waitTime = 10)
{
    EDBConnection con = GetConnection();
    using (EDBAQQueue queueListen = new EDBAQQueue("MSG_QUEUE", con))
    {
        queueListen.UdtTypeName = "myxml";
        queueListen.DequeueOptions.Navigation = EDBAQNavigationMode.FIRST_
MESSAGE;
        queueListen.DequeueOptions.Visibility = EDBAQVisibility.ON_COMMIT;
        queueListen.DequeueOptions.Wait = 1;
        EDBTransaction txn = null;

        while (1 == 1)
        {

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

            string messageId = "Unknown";
            try
            {
                // the listen function is a blocking function. It will Wait
the specified waitTime or until a
                // message is received.
                Console.WriteLine("Listening...");
                string v = queueListen.Listen(null, waitTime);
                // If we are waiting for a message and we specify a Wait time,
                // then if there are no more messages, we want to just bounce
out.
                if (waitTime > -1 && v == null)
                {
                    Console.WriteLine("No message received during Wait period.
");
                    Console.WriteLine();
                    continue;
                }

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

                Console.WriteLine("Attempting to dequeue message...");
                // dequeue the message
                EDBAQMessage deqMsg;
                try
```

(次のページに続く)

(前のページからの続き)

```
{
    deqMsg = queueListen.Dequeue();
}
catch (Exception ex)
{
    if (ex.Message.Contains("ORA-25228"))
    {
        Console.WriteLine("Message was not there.  Another
↪process must have picked it up.");
        Console.WriteLine();
        txn.Rollback();
        continue;
    }
    else
    {
        throw;
    }
}

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

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

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

    txn.Commit();

    Console.WriteLine("Finished processing message");
    Console.WriteLine();
}
else
{
    Console.WriteLine("Message was not dequeued.");
}
}
catch (Exception ex)
```

(次のページに続く)

(前のページからの続き)

```
        {
            Console.WriteLine("Failed To dequeue or process the dequeued_
↪message.");

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

11.2 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”クラスは利用可能なすべてのデキューモードをリストします。

値	説明
閲覧する	ロックせずにメッセージを読みます。
ロック済み	メッセージを読み取り、書き込みロックを取得します。
削除する	読み取り後に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利用可能なオプションをリストします。

プロパティ	説明
Consumer-Name	メッセージをデキューするコンシューマの名前。
Dequeue-Mode	これは、EDBAQDequeueMode から設定されます。デキューオプションにリンクされたロック動作を表します。
ナビゲーション	これは、EDBAQNavigationMode から設定されます。取得されるメッセージの位置を表します。
可視性	これは EDBAQVisibility から設定されます。現在のトランザクションの一部として、新しいメッセージがデキューされるかどうかを表します。
待って	検索条件ごとのメッセージの待機時間。
Msgid	メッセージ識別子。
相関関係	相関識別子。
DeqCondition	デキュー条件。これはブール式です。
変換	メッセージをデキューする前に適用される変換。
Delivery-Mode	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利用可能なオプションをリストします。

プロパティ	説明
可視性	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シーケンス。
変換	メッセージをエンキューする前に適用される変換。
Delivery-Mode	エンキューされたメッセージの配信モード。

****EDBAQMessage ****

“EDBAQMessage”クラスは、エンキュー/デキューされるメッセージをリストします。

プロパティ	説明
ペイロード	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”は利用可能なメッセージプロパティをリストします。

プロパティ	説明
優先順位	メッセージの優先度。
遅延	メッセージがデキューに使用できる期間の投稿。これは秒単位で指定されます。
有効期限	メッセージがデキューに使用できる期間。これは秒単位で指定されます。
相関関係	相関識別子。
試行	メッセージのデキューを試行した回数。
RecipientList	デフォルトのキューサブスクリバを無効にする受信者リスト。
ExceptionQueue	未処理のメッセージを移動するキューの名前。
EnqueueTime	メッセージがエンキューされた時間。
都道府県	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。

値	説明
期限切れ	メッセージは例外キューに移動されます。
処理済み	メッセージは処理され、保持されます。
準備完了	メッセージを処理する準備ができました。
待っています	メッセージは待機状態です。遅延に達していない。

****EDBAQMessageType ****

“EDBAQMessageType”クラスはペイロードのタイプを表します。

値	説明
生	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	ユーザー定義タイプのメッセージ。
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、利用可能なさまざまな種類のナビゲーションモードを表します。

値	説明
First_Message	検索語に一致する最初の利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。

プロパティ	説明
接続	使用する接続。
お名前	キューの名前。
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。たとえば、EDBAQMessageType.Udt。
UdtTypeName	メッセージタイプのユーザー定義タイプ名。
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プロパティ。

****EDBAQVisibility ****

“EDBAQVisibility”クラスは利用可能な可視性オプションを表します。

値	説明
すぐに	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:

- 上記のパラメータのデフォルトオプションを確認するには、“ここをクリック”<https://www.enterprisedb.com/docs/en/11.0/EPAS_BIP_Guide_v11/Database_Compatibility_for_Oracle_Developers_Built-in_Package_Guide.1.14.html#pID0E01HG0HA>’_。
 - EDBAQ 機能は、ユーザー定義タイプを使用してエンキュー/デキュー操作を呼び出します。“NoTypeLoading”はユーザー定義タイプをロードしないため、“Server Compatibility Mode = NoTypeLoading”は EDBAQ で使用できません。
-

第 12

.NET アプリケーションでの参照カーソルの使用

“`ref cursor`”はクエリ結果セットへのポインタ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た `OPEN FOR` ステートメントの実行によって決定されます。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`” ステートメントで同じカーソル変数を何度も開くことができ、そのたびにそのクエリに対して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可能になります。カーソル変数を宣言するには、2 つの方法があります。

- 弱いデータ型の `ref` カーソルを宣言するには、“`SYS_REFCURSOR`”組み込みデータ型を使用します。
- 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 `ref` カーソルタイプです。これは、弱い型指定の参照カーソルとして知られています。次の例は、弱く型付けされた参照カーソルの宣言です。

```
name SYS_REFCURSOR;
```

以下は、強く型付けされた参照カーソルの例です。

```
タイプ<cursor_type_name> REF CURSOR RETURN emp % ROWTYPE;
```

ストアドプロシージャの作成

次のサンプルコードは、“`refcur_inout_callee`”というストアドプロシージャを作成します。サンプルプロシージャを作成するには、`EDB-PSQL` を呼び出し、`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 IN OUT SYS_REFCURSOR)
```

(次のページに続く)

(前のページからの続き)

```
IS
BEGIN
    OPEN v_refcur FOR SELECT ename FROM emp;
END;
```

.NET コードから上記の定義されたプロシージャを使用するには、上記のスクリプトに示すように、“IN”パラメータとして渡される ref カーソルのデータ型を指定する必要があります。

次の C #コードは、ストアードプロシージャを使用して emp テーブルから従業員名を取得します。

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

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

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

                command.ExecuteNonQuery();
                String cursorName = command.Parameters[0].Value.ToString();
                command.CommandText = "fetch all in \" + cursorName + "\"";
                command.CommandType = CommandType.Text;
            }
            catch { }
            tran.Commit();
            conn.Close();
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

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

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を使用している場合は、“EDBParameter ()”の“EDBDbType.RefCursor”タイプをバインドする必要があることに注意してください。

プラグインを使用する

EDB .Net ドライバープラグインは、さまざまなデータ型の拡張機能をサポートするために導入されています。これは、それ以外では .Net では利用できません。利用可能なさまざまなプラグインのサポート：

- GeoJSON
- Json.NET
- レガシー PostGIS
- NetTopologySuite
- 野田タイム
- Rawpostgis

プラグインは、空間、データ/時間、Json タイプの使用をサポートします。以下のセクションでは、これらのプラグインでサポートされ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について詳しく説明します。

13.1 GeoJSON

.NET Framework 4.5 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/net45
```

次の共有ライブラリファイルが必要です。

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

```
GeoJSON.Net.dll
```

```
Newtonsoft.Json.dll
```

.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 プラグインの使用に関する詳細については、[‘Npgsql のドキュメントを参照してください<http://www.npgsql.org/doc/types/geojson.html/>’](http://www.npgsql.org/doc/types/geojson.html)。

13.2 Json.NET

.NET Framework 4.5 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/net45
```

次の共有ライブラリファイルが必要です。

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

```
Newtonsoft.Json.dll
```

.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 プラグインの使用に関する詳細については、**Npgsql** のドキュメントを参照してください<<http://www.npgsql.org/doc/types/jsonnet.html>> ‘_。

13.3 LegacyPostGIS

.Net Framework 4.5 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 / net45
```

次の共有ライブラリファイルが必要です。

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

.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 プラグインの使用に関する詳細については、[Npgsql](http://www.npgsql.org/doc/types/legacy-postgis.html) のドキュメントを参照してください<<http://www.npgsql.org/doc/types/legacy-postgis.html>> ‘[_](http://www.npgsql.org/doc/types/legacy-postgis.html)’。

13.4 NetTopologySuite

.Net Framework 4.5 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/net45
```

次の共有ライブラリファイルが必要です。

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

.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 タイププラグインの使用に関する詳細については、[Npgsql](http://www.npgsql.org/doc/types/nts.html) のドキュメントを参照してください。 <<http://www.npgsql.org/doc/types/nts.html>> _。

13.5 野田タイム

.Net Framework 4.5 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 / net45
```

次の共有ライブラリファイルが必要です。

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

.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 プラグインの使用に関する詳細については、**‘Npgsql** のドキュメントを参照してください<<http://www.npgsql.org/doc/types/nodatime.html>>**‘**。

13.6 RawPostGIS

.Net Framework 4.5 で RawPostGIS 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 / net45
```

次の共有ライブラリファイルが必要です。

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

.Net Standard 2.0 で RawPostGIS タイプ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 プラグインの使用に関する詳細については、“ドキュメント”<<https://www.nuget.org/packages/Npgsql.RawPostgis/>> ‘_。

第 14

API リファレンス

API の使用については、`Npgsql` のドキュメントを参照してください<<http://www.npgsql.org/doc/api/Npgsql.html>>
’。

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。
- クラスを参照するときは、`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。

おわりに

EDB Postgres .NET コネクタガイド

Copyright©2007-2019 EnterpriseDB Corporation。無断複写・転載を禁じます。

EnterpriseDB®Corporation

34 Crosby Drive, Suite 201, Bedford, MA 01730, アメリカ

T +1 781 357 3390 F +1 978 467 1307 E

info@enterprisedb.com

Web の “www.enterprisedb.com” で EnterpriseDB にアクセスします<<https://www.enterprisedb.com/>> ‘_’。

- EDB は、EDB ソフトウェアのログオン UI が存在する場合、その設計、コーディングのベストプラクティスの確立、レビュー、および入力検証の検証を行います。EDB は追加の入力コンポーネントに対して同じアプローチに従いますが、製品の性質上、限定された検証が可能な信頼できるユーザーによって入力および送信されるフリーフォーム SQL、WMI またはその他の文字列を受け入れる必要がある場合があります。このような場合、ユーザーが誤った入力や危険な入力を禁止することはできません。
- EDB は、将来、認証された信頼できるユーザーからのフリーフォーム SQL、WMI またはその他の潜在的に危険な入力を受け入れる製品に機能を追加する権利を留保しますが、そのような機能はすべて、可能な限り最小のリスクを提供するように設計およびテストされます可能であれば、スーパーユーザーまたは同等の特権が必要です。
- EDB は、すべての潜在的な脅威を予測できることを保証しません。したがって、すべての潜在的な脆弱性が対処または検討されたことをプロセスが完全に保証することはできません。

.NET framework setup, 14

API Reference, 78

Conclusion, 79

configuring the .NET connector, 12

Deleting Records in a Database, 42

dequeue message, 57

edbaq classes, 65

enqueue message, 57

executing stored procedure with in, 52

executing stored procedure with in parameters, 48

executing stored procedure without parameters, 46

Inserting Records in a Database, 40

Installing and Configuring the .NET Connector, 5, 13–17, 19

Opening a Database Connection, 24, 25

opening database connection from console application, 30

opening database connection from windows form application, 32

opening database connection using asp.net, 28

out and inout parameters, 52

Parameterized Queries, 38

referencing the library files, 13

Requirements Overview, 1

Retrieving Database Records, 33

Supported platforms, 2

Supported server versions, 2

the .net class hierarchy, 4

The Advanced Server .NET Connector - Overview, 3

Using a Ref Cursor in a .NET Application, 68

Using Advanced Queueing, 55

Using Plugins, 71

Using SPL Stored Procedures in your .NET Application, 44

Using the .NET Connector, 23

What' s New, 1