|  |
| --- |
| かずきのBlog@hatena |
| Enterprise Library 5.0入門 |
| Enterprise Libraryの基本的な使用方法 |
|  |
| **okazuki** |
| **2012/04/03** |

|  |
| --- |
| 2011年5月にリリースされたEnterprise Library 5.0の入門になります。 |

改版履歴

|  |  |
| --- | --- |
| **日時** | **改版内容** |
| 2012/04/04 | 初版作成 |

目次

[1. はじめに 3](#_Toc321344590)

[1.1. 前提条件 3](#_Toc321344591)

[2. Enterprise Libraryとは 3](#_Toc321344592)

[1.2. Enterprise Libraryの構成 3](#_Toc321344593)

[3. Hello world 4](#_Toc321344594)

# はじめに

ここでは、Microsoftのpattern & practiceチームが2011年5月にリリースしたEnterprise Library 5.0について著者の学習もかねて記載したものになります。記載内容に誤りを見つけた方はお手数をおかけしますが下記メールアドレスまたは、Twitterアカウントまで連絡ください。

* メール：[k\_ota28@hotmail.com](mailto:k_ota28@hotmail.com)
* Twitter: okazuki

## 前提条件

本書では、Visual Studio 2010を使用して.NET Framework 4(Client Profileではない)を使用しています。

## Enterprise Libraryとは

Enterprise Libraryとは、Microsoftのpattern & practiceチームが開発しているエンタープライズアプリケーションの開発のベストプラクティスを集めたライブラリです。このライブラリの特徴は、UnityというDIコンテナを軸にしてEnterprise Libraryが提供している機能(Application Blockと呼ばれる)を組み合わせて使用できるという点です。そのためEnterprise Libraryが提供している機能のうちのごく一部を使うことも、全機能使うことも問題なく出来ます。

また、Unityを使用せずに自分のプログラムから直接使用することも可能なように作成されています。本書では、UnityをDIコンテナとして使用して各Application Blockの機能を利用する方法を説明します。

### Application Blockとは

Application Blockとは、Enterprise Libraryを構成するある程度まとまった機能のことです。Application Block単体でも動作するように作成されていますが、一部のApplication Blockは他のApplication Blockに依存しているものもあります。以下にEnterprise LibraryのApplication Blockを示します。

* Caching Application Block  
  アプリケーション内でキャッシュ機能を提供します。
* Cryptography Application Block  
  データの暗号化・複合とハッシュを生成する機能を提供します。
* Data Access Application Block  
  データベースへのアクセス機能を提供します。
* Exception Handling Application Block  
  例外処理の機能を提供します。
* Logging Application Block  
  ログ出力機能を提供します。
* Policy Injection Application Block  
  機能横断的なPolicyをアプリケーションに適用する機能を提供します。
* Security Application Block  
  承認の規則（操作の許可や拒否など）を構成・管理する機能を提供します。
* Validation Application Block  
  値の妥当性検証の機能を提供します。

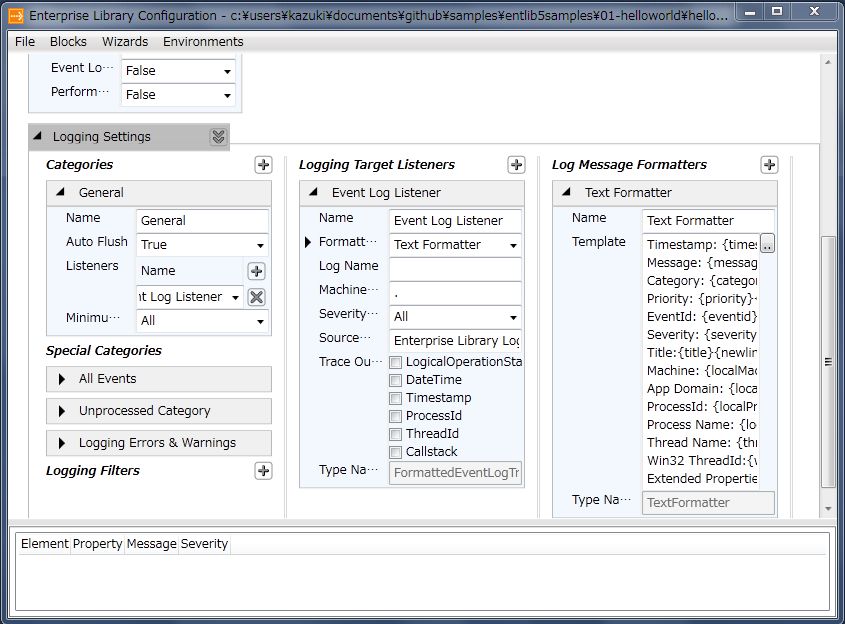
説明からもわかるように、Application Blockは特別な機能を提供しているわけではありません。一般的なアプリケーションであれば実装している機能になります。これらの機能の使い方を理解し適切に選択して使用できるようになると、アプリケーションを効率よく開発することが出来るようになると思います。（もしくは、どのような機能セットを提供しているのか、どのような点に留意して作成されているのかという参考にすることも出来ます）

## Enterprise Libraryの構成

Enterprise Libraryの基本的な使い方の流れは以下のようになります。

1. Enterprise Libraryで使用するApplication Blockを選択します。
2. 関連するアセンブリを参照に追加します。
3. Enterprise Libraryの構成を行います。以下のような方法があります。
   1. 構成ファイル(app.configやweb.config)を使用する方法
   2. 任意のファイルを使用する方法
   3. コードで構成する方法
4. Enterprise Libraryのコンテナを初期化します。
5. Enterprise Libraryのコンテナから必要な機能を取り出して使用します。

Enterprise Libraryには下図のような構成ツールが付属していて、これによりグラフィカルにapp.configやweb.configを設定することが出来ます。ここで設定した内容によって各アプリケーションブロックの機能をカスタマイズすることが出来ます。

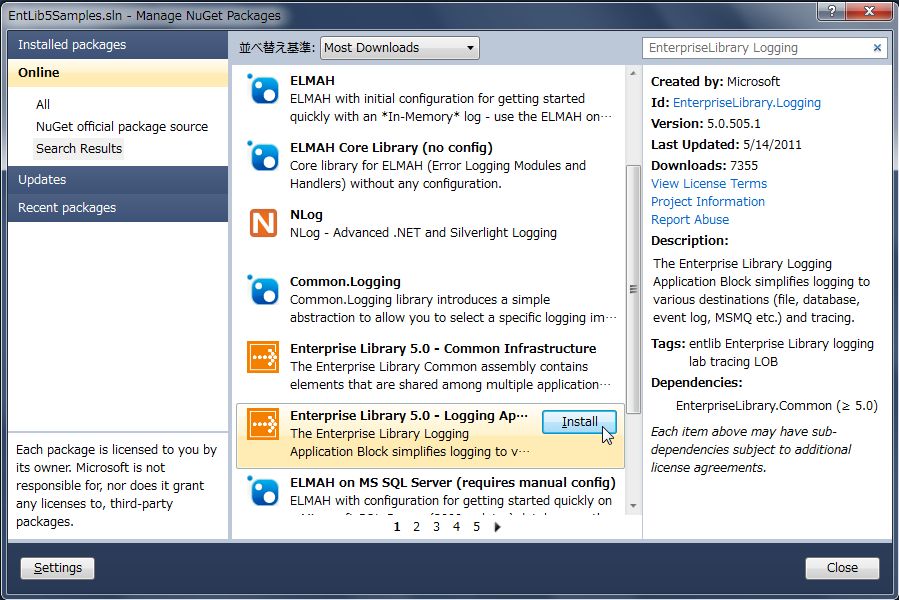


Enterprise Libraryでは、app.configなどの構成ファイル以外にもアプリケーションから構成情報を組み立てるFluent APIも提供しています。本書では、サンプルプログラムとしての見通しを優先するためFluent APIによる構成情報の組み立てを優先して使用します。

## Hello world

ここでは、Loggin Application Blockを使ってHello worldというログをファイルに出力する方法を通じて基本的なEnterprise Libraryの使い方の流れについて説明します。

HelloWorldという名前でコンソールアプリケーションを作成します。そしてNuGet Package Managerから「EnterpriseLibrary Logging」で検索をして「Enterprise Library 5.0 – Logging Application Block」をプロジェクトに追加します。

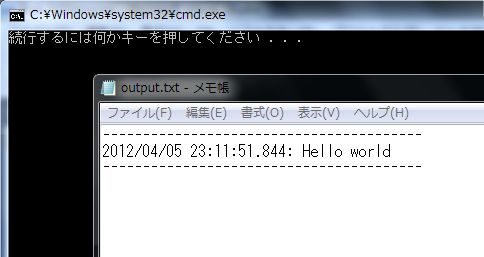


プロジェクトに参照が追加されたら、続けて参照の追加からSystem.Configurationを追加します。以上でプロジェクトの参照の設定は終了です。（昔はこの参照設定を整えるだけで一仕事だったのですが、いい時代になったものです…）Program.csを開いて下記のように編集します。

1. using System.Diagnostics;
2. using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
3. using Microsoft.Practices.EnterpriseLibrary.Logging;
4. namespace HelloWorld
5. {
6. class Program
7. {
8. static void Main(string[] args)
9. {
10. // 構成情報を組み立てる
11. var builder = new ConfigurationSourceBuilder();
12. builder.ConfigureLogging()
13. .SpecialSources
14. .AllEventsCategory
15. .SendTo
16. .FlatFile("FlatFileListener")
17. .FormatWith(
18. new FormatterBuilder()
19. .TextFormatterNamed("TextFormatter")
20. .UsingTemplate("{timestamp(local:yyyy/MM/dd HH:mm:ss.fff)}: {message}"))
21. .ToFile("output.txt");
22. // 組み立てた構成情報からConfigurationSourceを作成
23. var config = new DictionaryConfigurationSource();
24. builder.UpdateConfigurationWithReplace(config);
25. // 構成情報を元にEnterpriseLibraryのコンテナの初期化
26. EnterpriseLibraryContainer.Current = EnterpriseLibraryContainer.CreateDefaultContainer(config);
27. // EnterpriseLibraryのコンテナからLogging Application BlockのLog書き込み部品を取得
28. var logger = EnterpriseLibraryContainer.Current.GetInstance<LogWriter>();
29. // ログに出力する
30. logger.Write("Hello world");
31. // ログを表示
32. Process.Start("output.txt");
33. }
34. }
35. }

細かいAPIの利用方法は置いておいてコード全体の流れが、構成→コンテナの初期化→コンテナから部品の取り出し→使用という流れになっていることが確認できると思います。構成とコンテナの初期化をアプリケーションのエントリーポイントなどの初期化処理で行い、残りの部分で適時コンテナから必要な部品を取り出して使用するという流れになります。（もしくは、Dependency Injectionしてもらう）

このプログラムの実行結果は以下のようになります。ログが出力され、ログファイルが開かれます。



## 手順の簡略化

ここでは、ConfigurationSourceBuilderからEnterprise Libraryのコンテナを初期化するためのユーテリティコードを説明します。このコードは、Enterprise Libraryのコンテナの初期化の冗長なコードを簡略化することを目的としています。この後の、各Application Blockの説明では、ここで紹介したコードがあることを前提にサンプルを記載します。ConfigurationSourceBuilderからIServiceLocatorを作成するために下記の拡張メソッドを定義します。

1. namespace EntLib5Sample.Commons
2. {
3. using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
4. using Microsoft.Practices.ServiceLocation;
5. public static class ConfigurationSourceBuilderExtensions
6. {
7. /// <summary>
8. /// ConfigurationSourceBuilderの内容を元にIServiceProviderを作成します。
9. /// </summary>
10. /// <param name="self"></param>
11. /// <returns></returns>
12. public static IServiceLocator CreateContainer(this ConfigurationSourceBuilder self)
13. {
14. var configuration = new DictionaryConfigurationSource();
15. self.UpdateConfigurationWithReplace(configuration);
16. return EnterpriseLibraryContainer.CreateDefaultContainer(configuration);
17. }
18. }
19. }

このユーテリティを使用することで、前回のHello worldのコードは下記のようになります。

1. namespace HelloWorld
2. {
3. using System.Diagnostics;
4. using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
5. using Microsoft.Practices.EnterpriseLibrary.Logging;
6. using EntLib5Sample.Commons;
7. class Program
8. {
9. static void Main(string[] args)
10. {
11. // 構成情報を組み立てる
12. var builder = new ConfigurationSourceBuilder();
13. builder.ConfigureLogging()
14. .SpecialSources
15. .AllEventsCategory
16. .SendTo
17. .FlatFile("FlatFileListener")
18. .FormatWith(
19. new FormatterBuilder()
20. .TextFormatterNamed("TextFormatter")
21. .UsingTemplate("{timestamp(local:yyyy/MM/dd HH:mm:ss.fff)}: {message}"))
22. .ToFile("output.txt");
23. // 組み立てた構成情報からIServiceLocatorを作成
24. EnterpriseLibraryContainer.Current = builder.CreateContainer();
25. // EnterpriseLibraryのコンテナからLogging Application BlockのLog書き込み部品を取得
26. var logger = EnterpriseLibraryContainer.Current.GetInstance<LogWriter>();
27. // ログに出力する
28. logger.Write("Hello world");
29. // ログを表示
30. Process.Start("output.txt");
31. }
32. }
33. }

29行目で先ほど定義したメソッドを使用しています。本書では、この方法でEnterprise Libraryのコンテナの初期化を行います。

次章から、各Application Blockの使用方法を確認していきます。

# Application Block

ここでは、Enterprise Libraryで提供されるApplication Blockを１つずつ取り上げて説明します。

## Logging Application Block

ここではLogging Application Blockについて説明します。業務アプリケーションにおいて、ログは必須の構成要素です。地味な機能ですが、テスト時やリリース後の障害発生時の問題切り分けの重要な情報になります。また適切なログを出力して監視することで、障害の予兆を検知したり様々な利用方法があります。

.NETでは、System.Diagnostics.Traceを使ったログ出力機能が標準で備わっていますが、業務システムで必須の機能セットを備えているかと言われると力不足と言わざるを得ません。例えば、ファイルサイズが1000KBになったタイミングで今のログファイルを別名で退避するといったありがちな機能がありません。また、障害発生時に必要となるプロセスIDやアプリケーションドメイン名、コンピュータ名などの様々な情報を出力するような機能も用意されていません。

Enterprise LibraryのLogging Application Blockは、業務アプリケーション開発にひつような上記のSystem.Diagnostics.Traceに不足している機能セットを提供します。

### ログ出力機能の設定

ここでは、Fluent APIで設定できる設定項目について説明します。完全な設定内容は、リファレンスの下記のページから参照してください。ここでは代表的ないくつかを実際の記述例をもとに説明します。

* Microsoft.Practices.EnterpriseLibrary.Common.Configuration.Fluent Namespace  
  <http://msdn.microsoft.com/en-us/library/microsoft.practices.enterpriselibrary.common.configuration.fluent(v=pandp.50).aspx>

Fluent APIの難点は、APIドキュメントを見ただけでは何が出来るのかわからないという点にあると思います。そのため、インテリセンスとサンプルを見て記述のコツをつかむのが大事です。以下に、単純にログに出力するケースの設定コード例を示します。

1. var builder = new ConfigurationSourceBuilder();
2. builder.ConfigureLogging()
3. // 名前を付けてログの定義を開始
4. .LogToCategoryNamed("General")
5. // WithOptionsで追加オプション
6. // ここではGeneralをデフォルトのカテゴリとして設定
7. .WithOptions.SetAsDefaultCategory()
8. // フラットファイルに出力ファイル名はdefault.log
9. .SendTo.FlatFile("FlatFileListener").ToFile("default.log")
10. // フィルタリング（警告以上を表示する）
11. .Filter(SourceLevels.Warning)
12. // ログのフォーマットを指定
13. .FormatWith(new FormatterBuilder()
14. // フォーマッタの名前を指定
15. .TextFormatterNamed("LogFormatter")
16. // フォーマットを指定
17. .UsingTemplate("{timestamp(local:yyyy/MM/dd HH:mm:ss.fff)}: {severity}: {message}"));
18. ConfigureLoggingメソッド  
    ログの構成を開始するメソッド
19. LogToCategoryNamedメソッド  
    名前を付けてカテゴリを作成するメソッド。後続のメソッドチェインでカテゴリの設定を行う。再度、LogToCategoryNamedメソッドが呼ばれるまで、このメソッドで作成したカテゴリに対する設定になる。
20. WithOptionsプロパティ  
    ログにオプションを追加する。ここではデフォルトのカテゴリとして指定している。
21. SendToプロパティ  
    ログの出力先を指定する。ここではフラットファイルを指定している。このほかにもイベントログやMSMQやDatabaseなど様々な出力先に対応している。
22. Filterメソッド  
    ログのフィルタリングを行う条件を指定する。ここでは警告以上のログを出力するようにしている。
23. FormatWithメソッド  
    ログのフォーマットを指定するメソッド。フォーマットはFormatBuilderというヘルパークラスがあるので、それを使って作成する。フォーマットに指定できる名前は、LogEntryクラスのプロパティ名と大体同じ。（<http://msdn.microsoft.com/en-us/library/microsoft.practices.enterpriselibrary.logging.logentry_members(v=pandp.50).aspx>）

上記の構成をした状態で下記のようなログを出力するコードを記載します。

1. var l = EnterpriseLibraryContainer.Current.GetInstance<LogWriter>();
2. // Verbose～Criticalまでのログを出力
3. l.Write(new LogEntry { Message = "VerboseMessage", Severity = TraceEventType.Verbose});
4. l.Write(new LogEntry { Message = "InformationMessage", Severity = TraceEventType.Information});
5. l.Write(new LogEntry { Message = "WarningMessage", Severity = TraceEventType.Warning});
6. l.Write(new LogEntry { Message = "ErrorMessage", Severity = TraceEventType.Error});
7. l.Write(new LogEntry { Message = "CriticalMessage", Severity = TraceEventType.Critical});

Enterprise LibraryのコンテナからLogWriterを取得してログを出力します。LogWriterのWriteメソッドを使用してログを出力できます。ここでは、LogEntryというログに出力する情報を表すクラスを渡すオーバーロードを指定してVerbose～Criticalまでのログを出力しています。このプログラムを実行すると下記のログが出力されます。

----------------------------------------

2012/04/09 22:41:24.096: Warning: WarningMessage

----------------------------------------

----------------------------------------

2012/04/09 22:41:24.110: Error: ErrorMessage

----------------------------------------

----------------------------------------

2012/04/09 22:41:24.110: Critical: CriticalMessage

----------------------------------------

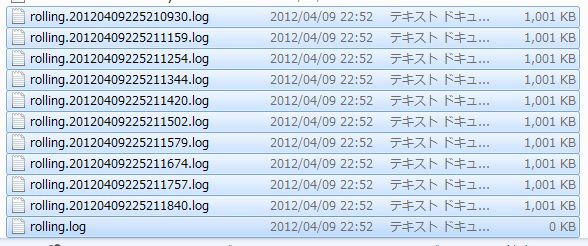
Filterメソッドで指定したとおり警告(Warning)以上のログしか出力していないことが確認できます。

### ローリングの設定

先ほどのサンプルでは単純なフラットファイルでのログの出力だったので、ここではファイルを一定の条件でローリングする設定方法について説明します。

1. var builder = new ConfigurationSourceBuilder();
2. builder.ConfigureLogging()
3. // 名前を付けてログの定義を開始
4. .LogToCategoryNamed("General")
5. // WithOptionsで追加オプション
6. // ここではGeneralをデフォルトのカテゴリとして設定
7. .WithOptions.SetAsDefaultCategory()
8. // フラットファイルに出力ファイル名はdefault.log
9. .SendTo.FlatFile("FlatFileListener").ToFile("default.log")
10. // フィルタリング（警告以上を表示する）
11. .Filter(SourceLevels.Warning)
12. // ログのフォーマットを指定
13. .FormatWith(new FormatterBuilder()
14. // フォーマッタの名前を指定
15. .TextFormatterNamed("LogFormatter")
16. // フォーマットを指定
17. .UsingTemplate("{timestamp(local:yyyy/MM/dd HH:mm:ss.fff)}: {severity}: {message}"))
18. // Rollingという名前でログの定義を開始
19. .LogToCategoryNamed("Rolling")
20. // SendTo.RollingFileで一定の条件を満たしたらローリング
21. .SendTo.RollingFile("RollingFileListener")
22. // 1000KBでローリング
23. .RollAfterSize(1000)
24. // 1分間隔でローリング
25. .RollEvery(RollInterval.Minute)
26. // ローリングしたファイルにタイムスタンプをつける
27. .UseTimeStampPattern("yyyyMMddHHmmssfff")
28. // 10世代管理
29. .CleanUpArchivedFilesWhenMoreThan(10)
30. // ファイル名はrolling.log
31. .ToFile("rolling.log");

上記コードの19行目からが、ローリングの設定です。SendToでRollingFileメソッドを使うことでローリングの設定が出来ます。その後に、ローリングするときの条件（サイズや時間）にローリングしたときのファイル名の命名規約や何世代までログを管理するか指定します。上記のような設定で、ログを大量に出力した結果を以下に示します。



1000KBのファイルが10世代、管理されていることが確認できます。今回のコードではローリングするログの設定をカテゴリ名”Rolling”で作成したため、ログ出力の際にこのカテゴリ名を指定する必要があります。（指定しない場合はデフォルトに設定しているGeneralが使用されます。ログ出力部分のコードは下記のようになります。

1. var l = EnterpriseLibraryContainer.Current.GetInstance<LogWriter>();
2. l.Write("ログメッセージ", "Rolling");
3. l.Write(new LogEntry { Message = "sample message", Categories = { "Rolling" } });

LogWriterのWriteメソッドの第二引数でカテゴリ名を設定します。LogEntryを使用する場合はCategoriesプロパティにカテゴリを文字列の配列で渡します。Categoriesプロパティに複数のカテゴリを指定することで、一度に複数個所にログを出力することも可能です。

### イベントログへの出力

最後にイベントログへの出力例を説明します。イベントログに出力するにはSendToのあとにEventLogメソッドを呼び出します。そして、UsingEventLogSourceメソッドで何処に出力するか指定します。コード例を以下に示します。

1. var builder = new ConfigurationSourceBuilder();
2. builder.ConfigureLogging()
3. // 名前を付けてログの定義を開始
4. .LogToCategoryNamed("General")
5. // WithOptionsで追加オプション
6. // ここではGeneralをデフォルトのカテゴリとして設定
7. .WithOptions.SetAsDefaultCategory()
8. // フラットファイルに出力ファイル名はdefault.log
9. .SendTo.FlatFile("FlatFileListener").ToFile("default.log")
10. // フィルタリング（警告以上を表示する）
11. .Filter(SourceLevels.Warning)
12. // ログのフォーマットを指定
13. .FormatWith(new FormatterBuilder()
14. // フォーマッタの名前を指定
15. .TextFormatterNamed("LogFormatter")
16. // フォーマットを指定
17. .UsingTemplate("{timestamp(local:yyyy/MM/dd HH:mm:ss.fff)}: {severity}: {message}"))
18. // Rollingという名前でログの定義を開始
19. .LogToCategoryNamed("Rolling")
20. // SendTo.RollingFileで一定の条件を満たしたらローリング
21. .SendTo.RollingFile("RollingFileListener")
22. // 1000KBでローリング
23. .RollAfterSize(1000)
24. // 1分間隔でローリング
25. .RollEvery(RollInterval.Minute)
26. // ローリングしたファイルにタイムスタンプをつける
27. .UseTimeStampPattern("yyyyMMddHHmmssfff")
28. // 10世代管理
29. .CleanUpArchivedFilesWhenMoreThan(10)
30. // ファイル名はrolling.log
31. .ToFile("rolling.log")
32. // EventLogという名前でログの定義を開始
33. .LogToCategoryNamed("EventLog")
34. // EventLogに送信するEventLogListener
35. .SendTo.EventLog("EventLogListener")
36. // ソースはApplication
37. .UsingEventLogSource("Application");

33行目からが追加したイベントログの定義になります。コメントにあるようにApplicationのログに出力するように設定しています。この状態で下記のようなコードを書くとイベントログにログが出力されます。

1. var l = EnterpriseLibraryContainer.Current.GetInstance<LogWriter>();
2. l.Write("EventLogMessage", "EventLog", 0, 0, TraceEventType.Information);

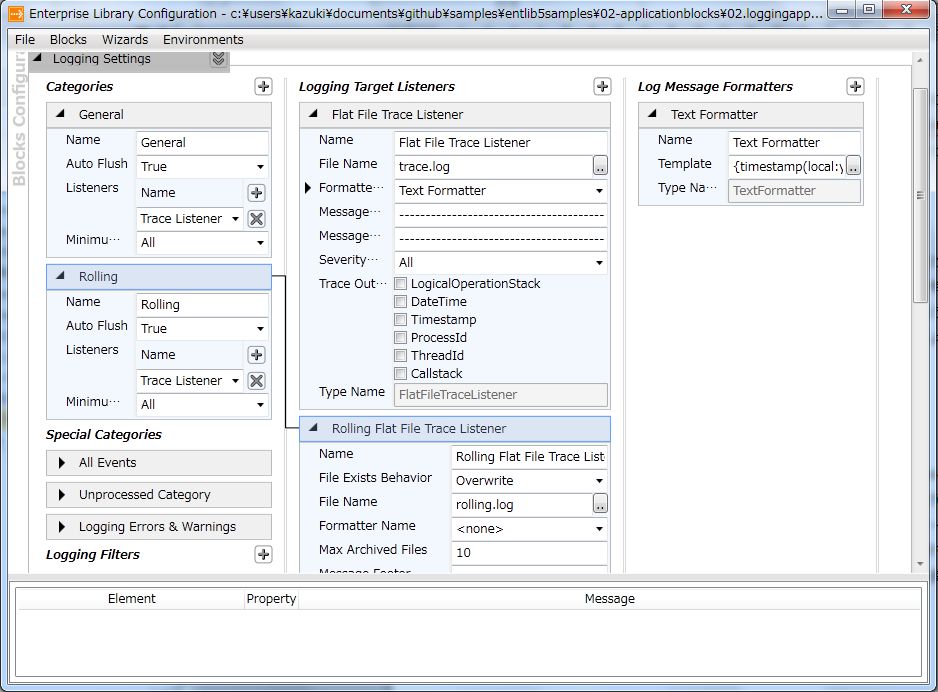
プログラムを実行して、イベントログを確認するとログが出力されていることがわかります。



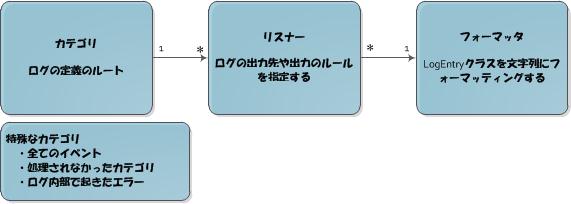
### 構成ファイルによるログの設定

ConfigurationSourceBuilderによるFluent APIでの構成はプログラムでログの設定が組み立てられて慣れるととてもいいのですが、ログに関しては障害の調査のために一時的にログをErrorからInformationに下げて取得したいといったことが考えられます。このように、再コンパイルを行わずに構成変更することが考えられるログは、本番では構成ファイルによって設定したほうが良いと考えられます。

Enterprise Libraryにはグラフィカルに構成ファイルを編集できるツールがついているのでそれを使用して構成を組み立てることができます。



Logging Application Blockの構成は下記のような構造になっています。

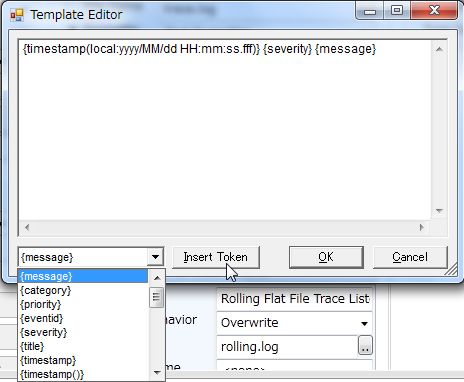


カテゴリを定義して、それに対してログの出力先やルール（ローリング等）を定義するリスナーを複数紐づけます。さらにリスナーにログの出力形式を指定するフォーマッタを紐づけます。カテゴリには全てのイベントを受け取るカテゴリと、処理されなかったカテゴリを受け取るものと、ログ内部でエラーが起きたときのための特殊なカテゴリも定義されています。

### フォーマッタで指定可能な書式

Logging Application BlockのTextFormatterでは{}で括られたプレースホルダでログのメッセージやタイムスタンプなどを任意の並びで表示するように構成可能です。代表的なものに、ログの文字列を指定する{message}や、タイムスタンプを指定する{timestamp}、ログの重要度を指定する{severity}などがあります。このフォーマットを組み立てるには、Enterprise Libraryの構成ツールのTemplate Editorで作成するのが楽なのでお勧めです。

Template Editorは、構成ツールのLog Message Fomattersの下にText Formatterを作成し、そのTemplateの横にある[…]というボタンをクリックすることで表示させることが可能です。



### その他の機能

この他にもLogging Application BlockにはデータベースやWCFへのログの出力やメールなど一般的な用途に使えそうなログ出力の機能が提供されています。また、LogWriterクラスのIsLoggingEnabledメソッドなどを使うことで不要な時はログを出力しないといったコードも作成することが可能です。

### まとめ

以上でLogging Application Blockの説明は終わりです。Logging Application Blockが、かなり多機能なログ出力の機能を持っていることが確認できたと思います。地味ながら業務システムに必須のログ出力機能を提供しています。また、構成を変更することでアプリケーションロジックのコードには手を入れることなくログの出力先やログのフィルタリングも行えます。特に採用するログ出力ライブラリが決まっていないときの選択肢の１つとして検討してみても良いと思います。

## Data Access Application Block

ここでは、データベースにアクセスするための機能を提供するData Access Application Blockについて説明します。Data Access Application Blockを使うと、データベースにアクセスするための定型的なコードを簡略化することができます。

Data Access Application Blockを使用するには、プロジェクトを作成してNuGetからEnterpriseLibrary.Dataをインストールします。



Data Access Application Blockを利用するには、Fluent APIを利用することも出来ますがログと同様に接続先のDBは構成ファイルで管理するのが一般的だと思うのでFluent APIは利用せずに構成ファイルで接続文字列を管理するようにします。

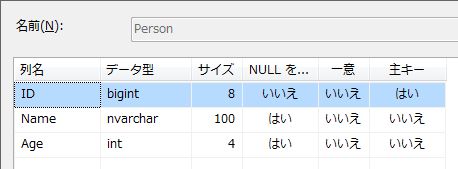
### 主な提供機能

Data Access Application Blockは、Databaseと呼ばれるクラスを使って各種データベースへのアクセスを行います。１メソッドでDataSetやDataTableへSQLを使ってデータを読み込むことや、DbReaderを取得することが出来ます。また、SQLからPOCOにデータをつめこむということも行えます。その他、ストアドプロシージャのDbCommandに対するサポート機能が用意されていますが、ここではその部分についての説明は省略します。詳しくはEnterprise Libraryのドキュメントを確認してください。

### データベースの作成

ここでは、SQL Server Compact Edition 4.0に簡単なテーブルを定義してSQLを使ってデータを読み書きする方法を示します。DataSetやDataTableを利用する方法は下位互換のための機能だと思うので、ここではSQLからPOCOへデータを詰め込む機能に絞って説明を行います。

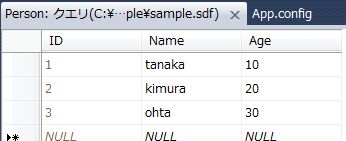
コンソールアプリケーションを新規作成しターゲットフレームワークを.NET Framework 4に変更します。そして、sample.sdfという名前でSQL Server Compact Edition 4.0のデータベースを作成します。作成したデータベースに下記の構造を持ったテーブルを作成します。



ID列は、下記のようにデータベースで自動的に採番されるようにします。

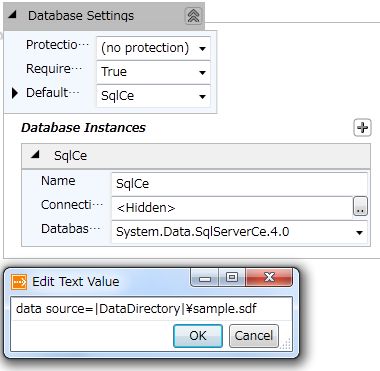


データは初期状態で3件登録しました。



### 構成ファイルの編集

構成ファイルは、Enterprise Libraryの構成ファイルを編集するツールを使用します。app.configをプロジェクトに追加して右クリックからEdit configuration fileを選択してツールを起動します。ツールを起動したらDatabase Settingsの箇所を下図のように、先ほど作成したSQL Server Compact Edition 4.0のデータベースに接続するようにSqlCeという名前（任意の名前で問題ありません）構成します。そして、デフォルトにSqlCeを設定して保存します。



### データベースへのアクセス

データベースにアクセスするにはEnterprise LibraryのコンテナからDatabaseクラスを取得します。そしてIEnumerable<T> ExecuteSqlStringAccessor<T>(string)メソッドを使ってSQL文を発行します。型引数のTはSQLの実行結果を格納するクラスを指定します。今回は”SELECT Id, Name, Age FROM PERSON ORDER BY Age DESC”というSQL文を実行するつもりなので、その結果を格納するプロパティを持ったPersonクラスを定義します。

1. class Person
2. {
3. public long Id { get; set; }
4. public string Name { get; set; }
5. public int Age { get; set; }
6. }

このクラスにデータを格納するコードを下記に示します。

1. // Enterprise LibraryのコンテナからDatabaseクラスのインスタンスを取得
2. var database = EnterpriseLibraryContainer.Current.GetInstance<Database>();
3. // SQL文を発行してデータを格納
4. var people = database.ExecuteSqlStringAccessor<Person>(
5. "SELECT Id, Name, Age FROM PERSON ORDER BY Age DESC");
6. // 結果を表示
7. foreach (var p in people)
8. {
9. Console.WriteLine("Id: {0}, Name: {1}, Age: {2}", p.Id, p.Name, p.Age);
10. }

4行目～5行目がデータを取得している箇所になります。このコードの実行結果を以下に示します。

Id: 3, Name: ohta, Age: 30

Id: 2, Name: kimura, Age: 20

Id: 1, Name: tanaka, Age: 10

パラメータつきのSQL文を実行する方法は下記のようになります。パラメータと引数を対応づけるには通常はDbCommandのParametersにDbParameterを追加しますが、SqlStringAccessorではIDbParameterMapperの実装クラスでパラメータと引数の対応付けを行います。Enterprise Libraryでは、特別な実装は用意されていないので利用者が要件にあった実装を行う必要があります。例えば@p1, @p2, @p3…のようなルールのパラメータに引数を当てはめるIDbParameterMapperは下記のような実装になります。

1. /// <summary>
2. /// @p1, @p2, @p3という名前の順番でパラメータをマッピングするパラメータマッパー
3. /// </summary>
4. class SequenceParameterMapper : IParameterMapper
5. {
6. /// <summary>
7. /// デフォルトインスタンス
8. /// </summary>
9. public static readonly IParameterMapper Default = new SequenceParameterMapper();
10. public void AssignParameters(DbCommand command, object[] parameterValues)
11. {
12. // 引数で渡された値をCommandParameterへ変換
13. var parameters = parameterValues
14. .Select((value, index) =>
15. {
16. var p = command.CreateParameter();
17. p.ParameterName = "p" + (index + 1);
18. p.Value = value;
19. return p;
20. })
21. .ToArray();
22. // コマンドにパラメータを追加
23. command.Parameters.AddRange(parameters);
24. }
25. }

AssignParametersメソッドでパラメータをDbCommandに設定しています。このクラスを使うとパラメータつきのSQL文は下記のように実行できます。

1. // Enterprise LibraryのコンテナからDatabaseクラスのインスタンスを取得
2. var database = EnterpriseLibraryContainer.Current.GetInstance<Database>();
3. // SQL文を発行してデータを格納
4. var accessor = database.CreateSqlStringAccessor<Person>(
5. // パラメータつきのSQL文
6. "SELECT Id, Name, Age FROM PERSON WHERE NAME LIKE @p1 ORDER BY ID DESC",
7. // パラメータのマッピングルール
8. SequenceParameterMapper.Default);
9. // パラメータを指定して実行
10. var people = accessor.Execute("%mu%");
11. // 結果を表示
12. foreach (var p in people)
13. {
14. Console.WriteLine("Id: {0}, Name: {1}, Age: {2}", p.Id, p.Name, p.Age);
15. }

CreateSqlStringAccessorが返すDataAccessor<T>型はExecute(params object[] parameters)というメソッドを持っているので、そこに必要な数のパラメータを渡して使用します。このコードの実行結果を以下に示します。

Id: 2, Name: kimura, Age: 20

Nameにmuを含むid:2のkimuraさんだけが抽出されていることが確認できます。

### データの更新と明示的なトランザクション管理

データの更新はDbCommandを使って行います。コードは以下のようになります。

1. // Enterprise LibraryのコンテナからDatabaseクラスのインスタンスを取得
2. var database = EnterpriseLibraryContainer.Current.GetInstance<Database>();
3. // コネクションを作成
4. using (var conn = database.CreateConnection())
5. {
6. conn.Open();
7. // トランザクションを開始
8. using (var tran = conn.BeginTransaction())
9. {
10. // 登録対象のデータ
11. var p = new Person { Name = "hanami", Age = 100 };
12. // コマンドをSQLから作成
13. var command = database.GetSqlStringCommand("INSERT INTO PERSON(NAME, AGE) VALUES(@p1, @p2)");
14. // パラメータを追加
15. database.AddInParameter(command, "p1", DbType.String, p.Name);
16. database.AddInParameter(command, "p2", DbType.Int32, p.Age);
17. // トランザクションを指定してコマンドを実行
18. var count = database.ExecuteNonQuery(command, tran);
19. // DB側でふられたIDを取得
20. var newId = database.ExecuteScalar(tran, CommandType.Text, "SELECT @@IDENTITY");
21. // 登録件数と、登録時にふられたIDを表示
22. Console.WriteLine("inserted: {0}, newId: {1}", count, newId);
23. // コミット
24. tran.Commit();
25. }
26. conn.Close();
27. }

上記のコードは単純にINSERT文を実行するだけではなく、Data Access Application Blockにおけるコネクションの明示的な管理方法とトランザクションの明示的な管理方法を示しています。このように、DBに非依存なAPIを使ってトランザクションやコネクションの管理が簡単にできるようになっています。

### TransactionScope

.NET Framework 2.0から導入されたTransactionScopeを使ったトランザクション管理にもData Access Application Blockは対応しています。何も考えずに使うと分散トランザクションに簡単に昇格してしまうことから嫌う人は多い機能([SQL Server 2008 と .NET Framework 2.0 SP1で緩和されてますが](http://blogs.msdn.com/b/nakama/archive/2008/10/23/transactionscope.aspx))ですが、Data Access Application BlockではTransactionScopeのトランザクションが存在している間は、Databaseクラスを経由して行った同じ接続文字列のコネクションをキャッシュしておいて、2重にコネクションが開かないように管理を行います。このためSQL Server 2008以降でなくても、分散トランザクションに昇格しにくくなっています。

まず、最初に分散トランザクションに対応していないSQL Server Compact Edition 4.0でTransactionScope内で2回コネクションを開いたときの挙動を下記に示します。

1. using (var tc = new TransactionScope())
2. {
3. // 単純にTransactionScope内で2つのコネクションを開いて閉じる
4. var conn1 = new SqlCeConnection(ConfigurationManager.ConnectionStrings["SqlCe"].ConnectionString);
5. conn1.Open();
6. conn1.Close();
7. var conn2 = new SqlCeConnection(ConfigurationManager.ConnectionStrings["SqlCe"].ConnectionString);
8. conn2.Open();
9. conn2.Close();
10. }

上記のコードを実行すると下記のように例外になります。

ハンドルされていない例外: System.InvalidOperationException: 接続オブジェクトをトランザクション スコープに参加させることができません。

場所 System.Data.SqlServerCe.SqlCeConnection.Enlist(Transaction tx)

場所 System.Data.SqlServerCe.SqlCeConnection.Open()

場所 以下省略

Databaseクラスを使って上記のように複数コネクションを開くように見えるコードを記載してみます。

1. // Enterprise LibraryのコンテナからDatabaseクラスのインスタンスを取得
2. using (var tc = new TransactionScope())
3. {
4. var database1 = EnterpriseLibraryContainer.Current.GetInstance<Database>();
5. Console.WriteLine("最初のdatabaseオブジェクト取得 HashCode: {0}", database1.GetHashCode());
6. for (int i = 0; i < 2; i++)
7. {
8. // 登録対象のデータ
9. var p = new Person { Name = "hanami", Age = 100 };
10. // コマンドをSQLから作成
11. var command = database1.GetSqlStringCommand("INSERT INTO PERSON(NAME, AGE) VALUES(@p1, @p2)");
12. // パラメータを追加
13. database1.AddInParameter(command, "p1", DbType.String, p.Name);
14. database1.AddInParameter(command, "p2", DbType.Int32, p.Age);
15. // トランザクションを指定してコマンドを実行
16. var count = database1.ExecuteNonQuery(command);
17. // DB側でふられたIDを取得
18. var newId = database1.ExecuteScalar(CommandType.Text, "SELECT @@IDENTITY");
19. // 登録件数と、登録時にふられたIDを表示
20. Console.WriteLine("database1: inserted: {0}, newId: {1}", count, newId);
21. }
22. var database2 = EnterpriseLibraryContainer.Current.GetInstance<Database>();
23. Console.WriteLine("2つ目のdatabaseオブジェクト取得 HashCode: {0}", database2.GetHashCode());
24. for (int i = 0; i < 2; i++)
25. {
26. // 登録対象のデータ
27. var p = new Person { Name = "sakurai", Age = 100 };
28. // コマンドをSQLから作成
29. var command = database2.GetSqlStringCommand("INSERT INTO PERSON(NAME, AGE) VALUES(@p1, @p2)");
30. // パラメータを追加
31. database2.AddInParameter(command, "p1", DbType.String, p.Name);
32. database2.AddInParameter(command, "p2", DbType.Int32, p.Age);
33. // トランザクションを指定してコマンドを実行
34. var count = database1.ExecuteNonQuery(command);
35. // DB側でふられたIDを取得
36. var newId = database2.ExecuteScalar(CommandType.Text, "SELECT @@IDENTITY");
37. // 登録件数と、登録時にふられたIDを表示
38. Console.WriteLine("database2: inserted: {0}, newId: {1}", count, newId);
39. }
40. tc.Complete();
41. }

複数のDatabaseオブジェクトを取得してTransactionScopeの中で複数のSQL文を発行しています。ここで複数個のコネクションが開かれていたら先ほどと同じように例外になるはずです。上記のコードの実行結果を以下に示します。

最初のdatabaseオブジェクト取得 HashCode: 58204539

database1: inserted: 1, newId: 11

database1: inserted: 1, newId: 12

2つ目のdatabaseオブジェクト取得 HashCode: 54078809

database2: inserted: 1, newId: 13

database2: inserted: 1, newId: 14

### ヘルパーメソッドのすゝめ

このようにData Access Application Blockを使うと、SELECT系のコードが簡潔に、そして更新系はTransactionScope内でのコネクションのキャッシュにより昇格が起こりにくくなる機能の恩恵を受けることができます。しかし、UPDATE文やINSERT文を発行するコードはお世辞にも直感的とは言い難いです。そのため、下記のようなヘルパーメソッドを作ることで完結に更新系のSQL文の発行も行えるようになります。

1. /// <summary>
2. /// Databaseクラスへの拡張メソッド
3. /// </summary>
4. static class DatabaseExtensions
5. {
6. // IParameterMapperに指定したパラメータマッピングルールでSQL文を実行する
7. public static int ExecuteUpdate(
8. this Database self,
9. string sql,
10. IParameterMapper mapper,
11. params object[] parameters)
12. {
13. using (var command = self.GetSqlStringCommand(sql))
14. {
15. mapper.AssignParameters(command, parameters);
16. return self.ExecuteNonQuery(command);
17. }
18. }
19. // SequenceParameterMapperのパラメータマッピングルールでSQL文を実行する
20. public static int ExecuteUpdate(
21. this Database self,
22. string sql,
23. params object[] parameters)
24. {
25. return self.ExecuteUpdate(sql, SequenceParameterMapper.Default, parameters);
26. }
27. }

上記のような拡張メソッドを定義すると、INSERT文などの更新系SQLを実行するコードは下記のようになります。

1. using (var tc = new TransactionScope())
2. {
3. // Enterprise LibraryのコンテナからDatabaseクラスのインスタンスを取得
4. var database = EnterpriseLibraryContainer.Current.GetInstance<Database>();
5. // 登録対象のデータ
6. var p = new Person { Name = "hanami", Age = 100 };
7. // 拡張メソッドを使って登録
8. var count = database.ExecuteUpdate(
9. "INSERT INTO PERSON(NAME, AGE) VALUES(@p1, @p2)",
10. p.Name, p.Age);
11. // DB側で割り振られたIDを取得
12. var newId = (decimal)database.ExecuteScalar(
13. CommandType.Text, "SELECT @@IDENTITY");
14. // 結果を表示
15. Console.WriteLine("inserted: {0}, newId {1}", count, newId);
16. tc.Complete();
17. }

実行結果は以下のようになります。

inserted: 1, newId 9

### Fluent APIでの設定

因みに、Data Access Application BlockをFluent APIで設定すると下記のようになります。

1. var builder = new ConfigurationSourceBuilder();
2. builder.ConfigureData()
3. // 使用する接続文字列を取得
4. .ForDatabaseNamed("SqlCe")
5. .AsDefault();
6. // Enterprise LibraryのコンテナからDatabaseクラスのインスタンスを取得
7. var database = EnterpriseLibraryContainer.Current.GetInstance<Database>();
8. // SQL文を発行してデータを格納
9. var people = database.ExecuteSqlStringAccessor<Person>(
10. "SELECT Id, Name, Age FROM PERSON ORDER BY ID DESC");
11. // 結果を表示
12. foreach (var p in people)
13. {
14. Console.WriteLine("Id: {0}, Name: {1}, Age: {2}", p.Id, p.Name, p.Age);
15. }

ForDatabaseNamedで接続文字列の名前を指定して、AsDefaultでデフォルトで使用するものに設定します。実行結果については割愛します。

## Caching Application Block

<http://msdn.microsoft.com/ja-jp/library/system.runtime.caching.memorycache.aspx>

<http://msdn.microsoft.com/ja-jp/library/dd997357.aspx>

<http://msdn.microsoft.com/ja-jp/library/system.runtime.caching.objectcache.aspx>