Professional Expertise Distilled

Microsoft Enterprise Library 5.0

Develop Enterprise applications using reusable software components of Microsoft Enterprise Library 5.0

Sachin Joshi

[PACKT] PUBLISHING enterprise professional expertise distilled

Microsoft Enterprise Library 5.0

Develop Enterprise applications using reusable software components of Microsoft Enterprise Library 5.0

Sachin Joshi

[PACKT] PUBLISHING enterprise professional expertise distilled

Microsoft Enterprise Library 5.0

First published: November 2010

Production Reference: 1041110

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 978-1-849680-90-5

www.packtpub.com

Cover Image by Dorota Feifer (dfeifer@hotmail.com)

Credits

Author
Sachin Joshi

Reviewers
Nikos Anastopoulos
Anand Narayanswamy

Acquisition Editor
Rashmi Phadnis

Development Editor
Reshma Sundaresan

Technical Editor
Neha Damle

Indexers
Monica Ajmera Mehta
Rekha Nair

Editorial Team Leader
Gagandeep Singh

Project Team Leader
Lata Basantani

Project Coordinator
Leena Purkait

Proofreader
Chris Smith

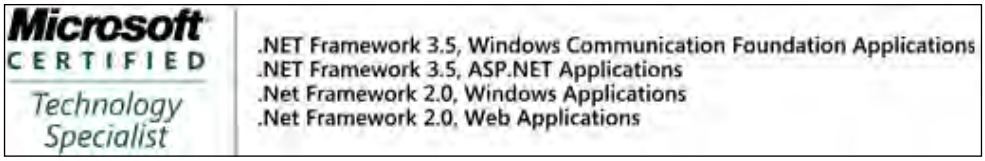
Graphics
Geetanjali Sawant

Production Coordinator
Arvindkumar Gupta

Cover Work
Arvindkumar Gupta

About the Author

Sachin Joshi holds a Master's Degree in Computer Applications and a Bachelor's Degree in Business Administration. He is a Microsoft Certified Professional Developer in ASP.NET and Microsoft Certified Technology Specialist in WCF and .NET 3.5 Windows. He has over five years of industry experience developing enterprise applications with Microsoft .NET and related technologies. Sachin was performing a juggling act between his college degree and running his own optical business for around five years before joining the IT industry.



Sachin is currently working as a Consultant in a well reputed software company in Hyderabad, India. He has several years of experience in designing and architecting solutions for various domains and he has been involved with several complex engagements. His technical strengths include C, C++, C#, VB.NET, Microsoft .NET, AJAX, Design Patterns, SQL Server, JavaScript, and so on. His current passion is Vala (`http://live.gnome.org/Vala`) a new programming language with C#-like syntax with the power of C.

Sachin blogs at `http://fuzzydev.com` and spends most of his time reading books and playing with different technologies. When not at work, Sachin spends time with his family, watching movies and playing video games. He and his wife have not only pledged but are working towards spreading awareness of the noble cause of organ donation. Sachin has a dream of opening a library for students who are economically disadvantaged, to enable them to grow. In the technology arena, Sachin is currently working on an open source project code named Apocalypse – *lifting of the veil*, a CMS based on ASP.NET 4.0.

> *Life is not about winning or losing, it's not about battles or competitions, it's not about mistakes or missed opportunities, it's about realizing the true meaning of life, it's about freeing your mind & soul of dust, life is about your own definition of success. YOUR LIFE IS BEAUTIFUL, YOU ARE ALWAYS SUCCESSFUL PERIOD*
>
> – *Sachin Joshi*

Acknowledgement

No book is the product of just the author, he just happens to be the one with his name on the cover.

A number of people contributed to the success of this book, and it would take more space than I have to thank each one individually.

I am greatly indebted to Rashmi Phadnis, Acquisition Editor at Packt Publishing, for accepting my proposal and also for her support and guidance from the beginning. I am thankful to Reshma Sundaresan, Development Editor at Packt Publishing, for the valuable advice at every stage through e-mails that encouraged me a lot. I am also thankful to Leena Purkait, Project Coordinator at Packt Publishing, for all the support and help in keeping me on schedule and also managing the schedule so professionally. I'm grateful to Nikos Anastopoulos and Anand Narayanswamy—thank you for reviewing the book and providing valuable and insightful feedback. Also, thanks to Dorota Feifer for the lovely cover image.

Special mention goes to David Barnes, Acquisition Editor at Packt Publishing, with whom I had the initial interaction. Thank you David for inspiring me during our interaction and with your blog posts (`http://davidbarneswork.posterous.com`). Also, a big thank you to the entire Packt Publishing team, for working so diligently to help bring out a high quality product.

I must also thank the talented team of developers who have contributed to the Enterprise Library project. This product truly helps in taking the complexity out of enterprise application development and allows developers to focus on the crux of the requirements.

About the Reviewers

Nikos Anastopoulos has a BSc and MSc in Software Engineering, from Coventry University, UK, and an MBA from Athens University of Economics and Business.

He has worked at Microsoft Hellas for 10 years, as a Platform Solutions Specialist for 3.5 years and as a Senior Consultant for 6.5 years at Microsoft Consulting Services. He has worked in many development projects in Greece, mostly in the Banking sector and Telcos, using Microsoft's development products, servers, and technologies like: .NET, C# (ASP.NET), SQL Server 200x, BizTalk Server 200x, and SharePoint Portal Server 200x. In all his engagements, he has extensively used Microsoft's Enterprise Library, since its very early releases.

Before joining Microsoft he worked as a developer in ISVs for two years working on client/server and Internet applications.

I would like to thank my family; my wife, Eirini, and my two lovely daughters, Kallia and Stavrianna, for their support over these years.

Anand Narayanaswamy is a Microsoft Most Valuable Professional (MVP) based in Trivandrum, India. He works as a freelance technical writer besides devoting time for blogging and tweeting. He also works as a technical editor for ASPAlliance.com. He had worked as a technical editor/reviewer for various publishers such as Sams, Addison-Wesley, McGraw Hill, and Packt. He is the author of *Community Server Quickly* (`www.packtpub.com/community-server/book`) published by Packt Publishing and runs `www.learnxpress.com` and `www.dotnetalbum.com`.

First, I would like to thank the Almighty for giving me the strength and energy to work every day. I specially thank my father, mother, and brother for providing valuable help, support, and encouragement. I also thank Leena Purkait, Project Coordinator, Packt Publishing, and Sachin Joshi for their assistance, cooperation, and understanding throughout the review process of this book.

This book is dedicated to:

My parents, Pravin and Geeta Joshi. They have dedicated their entire life towards making me what I am today. I Love You, You Rock!!!

Thanks to my beautiful wife, Nisha, for her love, support, and patience. I Love You, you make me complete.

Table of Contents

Preface

This book covers the fundamental elements of each application block so that you get a good understanding of its concepts. This is followed by referencing the required and optional assemblies and then initial configuration of that block using the configuration editor. Finally, it leverages the application block features to achieve goals of enterprise application development.

What this book covers

Chapter 1, *Getting Started with Enterprise Library*, introduces us to the Enterprise Library and explores various application blocks such as Unity, Policy Injection, Data Access block, Logging block, Exception Handling block, Caching block, Validation block, Security block, and Cryptography block.

Chapter 2, *Data Access Application Block*, explores the fundamental elements of the Data Access Application Block such as Database, SqlDatabase, OracleDatabase, SqlCeDatabase, GenericDatabase, Parameter Mapper, and Output Mappers.

Chapter 3, *Logging Application Block*, explores the fundamental elements of the Logging Application Block such as Log Category, Special Category, Logging Trace Listeners, Log Formatters, Logging Filters, Logger, LogWriter, LogEntry, and so on. We also learn about the various required and optional assemblies and learn to set up the initial configuration.

Chapter 4, *Exception Handling Application Block*, introduces us to the fundamental elements of the Exception Handling Application Block such as Exception Policy, Exception Types, and Exception Handler. We also learn about the required and optional assemblies, the initial infrastructure configuration, and the individual feature-level configuration.

Chapter 5, *Caching Application Block*, teaches us the fundamental elements of the Caching Application Block. We further learn to configure an encryption provider to encrypt cached data while using a persistent backing store.

Chapter 6, *Validation Application Block*, teaches us to validate objects using various approaches such as using an attribute, self-validation, programmatically, and through configuration. We also learn how the Validation Application Block can be integrated with Windows Forms-based applications and ASP.NET web applications.

Chapter 7, *Security Application Block*, introduces us to the key features of the Security Application Block and explores the elements of Authorization and Security Cache Providers. We also learn about the various required and optional assemblies.

Chapter 8, *Cryptography Application Block*, introduces us to the fundamental elements of the Cryptography Application Block such as IHashProvider, ISymmetricCryptoProvider, CryptographyManager, and so on. We also learn to generate hash, compare hash, and implement a custom hash provider. We also explore encryption and decryption of data and understand the basics of implementing a custom symmetric cryptography provider.

What you need for this book

To use this book you will need Microsoft Enterprise Library 5.0.

Who this book is for

If you are a programmer, consultant, or an associate architect, who is interested in developing Enterprise applications, this book is for you. We assume that you already have a good knowledge of Microsoft .NET framework and the C# programming language.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
Database database = container.Resolve<Database>();
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
Database database = container.Resolve<Database>();
```

Any command-line input or output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code for this book

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Getting Started with Enterprise Library

While developing enterprise-scale applications, developers often find themselves focusing on mundane and repeated tasks generally referred to as cross-cutting concerns: tasks such as writing repeated data access code, logging exceptions, handling and managing exceptions, caching data, validating user input, and so on. Although these tasks are important, developers often spend a lot of time implementing and debugging these cross-cutting concerns rather than channeling their efforts towards the core business requirements of customers. Developing this functionality in-house, as flexible and customizable reusable components, is one option but it involves time and money, not to mention the testing and bug-fixing effort.

This book will give you insight into Microsoft Enterprise Library, show you how to leverage the individual functional application blocks, and equip you with the knowledge to be productive in your work. Before embarking on the learning journey, read this chapter to get introduced to Enterprise Library; all other chapters are self contained so it doesn't matter whether you read the book sequentially or flip to a specific functional application block chapter with the intent to quickly get up to speed and leverage that specific functional application block.

In this chapter, you will:

- Receive an overview of Enterprise Library
- Receive a brief introduction to functional application blocks
- Learn about the dependencies between the functional application blocks
- Learn the system requirements
- Learn to install Enterprise Library

Introducing Enterprise Library

Enterprise Library (**EntLib**) is a collection of reusable software components or application blocks designed to assist software developers with common enterprise development challenges. Each application block addresses a specific cross-cutting concern and provides highly configurable features, which results in higher developer productivity. EntLib is implemented and provided by *Microsoft patterns & practices group*, a dedicated team of professionals who work on solving these cross-cutting concerns with active participation from the developer community. This is an open source project and thus freely available under the **Microsoft Public License** (**Ms-PL**) at the **CodePlex** open source community site (`http://entlib.codeplex.com`), basically granting us a royalty-free copyright license to reproduce its contribution, build derivative works, and distribute them.

More information can be found at the Enterprise Library community site `http://www.codeplex.com/entlib`.

Enterprise Library consists of nine application blocks; two are concerned with wiring up stuff together and the remaining seven are functional application blocks. This book focuses on the seven functional blocks and we have separate chapters in this book devoted to each functional application block.

The following is the complete list of application blocks; these are briefly discussed in the next sections.

- Wiring Blocks
    - Unity Dependency Injection
    - Policy Injection Application Block
- Functional Blocks
    - Data Access Application Block
    - Logging Application Block
    - Exception Handling Application Block
    - Caching Application Block
    - Validation Application Block
    - Security Application Block
    - Cryptography Application Block

Wiring Application Blocks

Wiring blocks provides mechanisms to build highly flexible, loosely coupled, and maintainable systems. These blocks are mainly about wiring or plugging together different functionalities. The following two blocks fall under this category:

- Unity Dependency Injection
- Policy Injection Application Block

Unity Application Block

The Unity Application Block is a lightweight, flexible, and extensible dependency injection container that supports interception and various injection mechanisms such as constructor, property, and method call injection. The Unity Block is a standalone open source project, which can be leveraged in our application. This block allows us to develop loosely coupled, maintainable, and testable applications. Enterprise Library leverages this block for wiring the configured objects. More information on the Unity block is available at `http://unity.codeplex.com`; the Unity block is not covered in this book.

Policy Injection Application Block

The Policy Injection Application Block is included in this release of Enterprise Library for backwards compatibility and policy injection is implemented using the Unity interception mechanism. This block provides a mechanism to change object behavior by inserting code between the client and the target object without modifying the code of the target object. The Policy Injection block is not covered in this book.

Functional Application Blocks

Enterprise Library consists of the following functional application blocks, which can be used individually or can be grouped together to address a specific cross-cutting concern. This book contains dedicated chapters for each of these functional application blocks; in each chapter we will explore the application block in detail.

- Data Access Application Block
- Logging Application Block
- Exception Handling Application Block
- Caching Application Block
- Validation Application Block

- Security Application Block
- Cryptography Application Block

Data Access Application Block

Developing an application that stores/retrieves data in/from some kind of a relational database is quite common; this involves performing **CRUD** (**Create, Read, Update, Delete**) operations on the database by executing T-SQL or stored procedure commands. But we often end up writing the plumbing code over and over again to perform these operations: plumbing code such as creating a connection object, opening and closing a connection, parameter caching, and so on.

The following are the key benefits of the Data Access block:

- The **Data Access Application Block** (**DAAB**) abstracts developers from the underlying database technology by providing a common interface to perform database operations.
- DAAB also takes care of the ordinary tasks like creating a connection object, opening and closing a connection, parameter caching, and so on.
- It helps in bringing consistency to the application and allows changing of database type by modifying the configuration.

We will further dive deep into the Data Access block in Chapter 2, *Data Access Application Block*.

Logging Application Block

Logging is an essential activity, which is required to understand what's happening behind the scene while the application is running. This is especially helpful in identifying issues and tracing the source of the problem without debugging. The Logging Application Block provides a very simple, flexible, standard, and consistent way to log messages. Administrators have the power to change the log destination (file, database, e-mail, and so on), modify message format, decide on which category is turned on/off, and so on. The Logging block is further discussed in *Chapter 3, Logging Application Block*.

Exception Handling Application Block

Handling exceptions appropriately and allowing the user to either continue or exit gracefully is essential for any application to avoid user frustration. The Exception Handling Application Block adapts the policy-driven approach to allow developers/ administrators to define how to handle exceptions.

The following are the key benefits of the Exception Handling Block:

- It provides the ability to log exception messages using the Logging Application Block.
- It provides a mechanism to replace the original exception with another exception, which prevents disclosure of sensitive information.
- It provides mechanism to wrap the original exception inside another exception to maintain the contextual information.

We will dive deep into Exception Handling Block in Chapter 4, *Exception Handling Application Block*.

Caching Application Block

Caching in general is a good practice for data that has a long life span; caching is recommended if the possibility of data being changed at the source is low and the change doesn't have significant impact on the application. The Caching Application Block allows us to cache data locally in our application; it also gives us the flexibility to cache the data in-memory, in a database or in an isolated storage. The Caching block is discussed in detail in Chapter 5, *Caching Application Block*.

Validation Application Block

The Validation Application Block (VAB) provides various mechanisms to validate user inputs. As a rule of thumb always assume user input is not valid unless proven to be valid. The Validation block allows us to perform validation in three different ways; we can use configuration, attributes, or code to provide validation rules. Additionally it also includes adapters specifically targeting ASP.NET, Windows Forms, and Windows Communication Foundation (WCF). We will explore the Validation block in detail in Chapter 6, *Validation Application Block*.

Security Application Block

The Security Application Block simplifies authorization based on rules and provides caching of the user's authorization and authentication data. Authorization can be done against Microsoft Active Directory Service, Authorization Manager (AzMan), Active Directory Application Mode (ADAM), and Custom Authorization Provider. Decoupling of the authorization code from the authorization provider allows administrators to change the provider in the configuration without changing the code. The Security block is explored in detail in Chapter 7, *Security Application Block*.

Cryptography Application Block

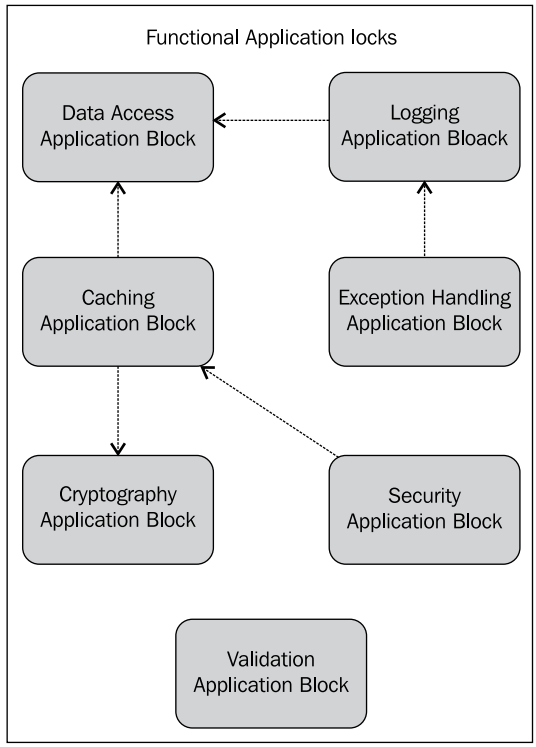
The Cryptography Application Block provides a common API to perform basic cryptography operations without inclining towards any specific cryptography provider and the provider is configurable. Using this application block we can perform encryption/decryption, hashing, & validate whether the hash matches some text. We will discuss the Cryptography block in detail in Chapter 8, *Cryptography Application Block*.

Functional Application Block Dependency

Several functional application blocks provide features that depends on other blocks; these dependencies are listed below.

Application Block	Dependencies	Condition
Logging Application Block	Data Access Application Block	If the messages have to be logged in database.
Exceptional Handling Application Block	Logging Application Block	If exception information has to be logged.
	Data Access Application Block	If exception information has to be logged to database.
Caching Application Block	Data Access Application Block	If data has to be cached in database.
	Cryptography Application Block	If cached data has to be encrypted.
Security Application Block	Caching Application Block	If credentials have to be cached.
	Cryptography Application Block	If cached credentials have to be encrypted.
	Data Access Application Block	If credentials have to be cached in database.

Following is the graphical representation of the dependencies between the functional application blocks:



Except the Validation block, all other application blocks are dependent on other blocks to provide additional features that are part of the respective application blocks. For example, the Exception Handling block is dependent on the Logging block to provide message logging functionality; additionally the Data Access block is also required if the message needs to be logged in database.

System requirements

Minimum requirements for Enterprise Library core features and the configuration tool are given below.

- **Supported Architectures**: x86 and x64
- **Operating System**:
    - Microsoft Windows® 7 Professional, Enterprise or Ultimate
    - Windows Server 2003 R2
    - Windows Server 2008 with Service Pack 2
    - Windows Server 2008 R2.
    - Windows Vista with Service Pack 2
- Microsoft .NET Framework 3.5 with Service Pack 1 or Microsoft .NET Framework 4.0
- Recommended Development Environment: Any of the following development systems:
    - Microsoft Visual Studio® 2008 Development System with Service Pack 1 (any edition)
    - Microsoft Visual Studio® 2010 Development System (any edition)
- **Required for Data Access Application Block**: A database server running a database that is supported by a .NET Framework 3.5 with Service Pack 1 or .NET Framework 4.0 data provider; data providers for OLE DB or ODBC are also supported. Below is the list for reference:
    - SQL Server 2000 or later
    - SQL Server 2005 Compact Edition
    - Oracle 9i or later
- **Required for Logging Application Block**: While using the Message Queuing (MSMQ) Trace Listener to store log messages, you need the Microsoft Message Queuing (MSMQ) components installed. Access to a database server is required to store log messages to database while using the Database Trace Listener. Access to an SMTP server is required to send e-mail, while using the E-mail Trace Listener to e-mail log messages.

- **Unit Testing Requirements**: To run the unit tests provided as part of the Enterprise Library source code installation we require the following:
    - Microsoft Visual Studio 2008 Professional or Visual Studio 2008 Team Edition or Visual Studio 2010 Premium or Visual Studio 2010 Professional, or Visual Studio 2010 Ultimate edition
    - Moq v3.1 assemblies, which can be downloaded from `http://code.google.com/p/moq/`

Installing Enterprise Library

Before we start exploring the individual application blocks, let us download and install Enterprise Library first. We can download the latest release of the Enterprise Library available at `http://msdn.microsoft.com/entlib/` from MSDN site; alternatively the download link is also available on the home page of the community site at `http://entlib.codeplex.com`. Click on the link **Enterprise Library 5.0 - April 2010** from the list of active releases and download the Microsoft Installer (MSI) package file from the download section. Now follow the steps given below to install the library. It is recommended to install the features given below.

Enterprise Library Binaries

This section provides options to selectively install specific application blocks; it is recommended that you install all the application blocks to avoid running the installer multiple times to add other blocks.

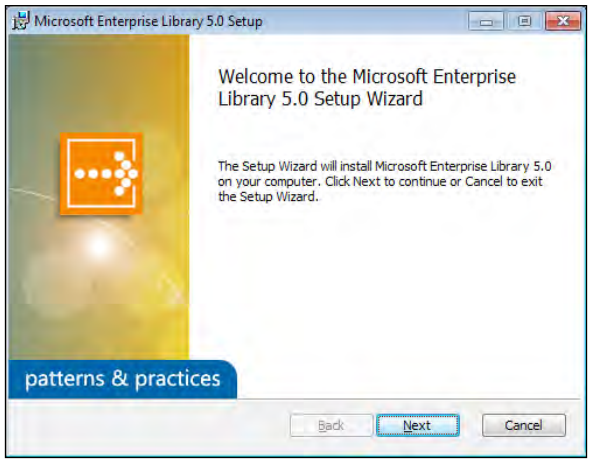
Configuration Editor for Visual Studio

Visual Studio integration of the configuration editor helps us in editing Enterprise Library configuration settings from within the development environment. This comes quite handy instead of switching between the standalone configuration editor and Visual studio IDE.

Source Code of Enterprise Library

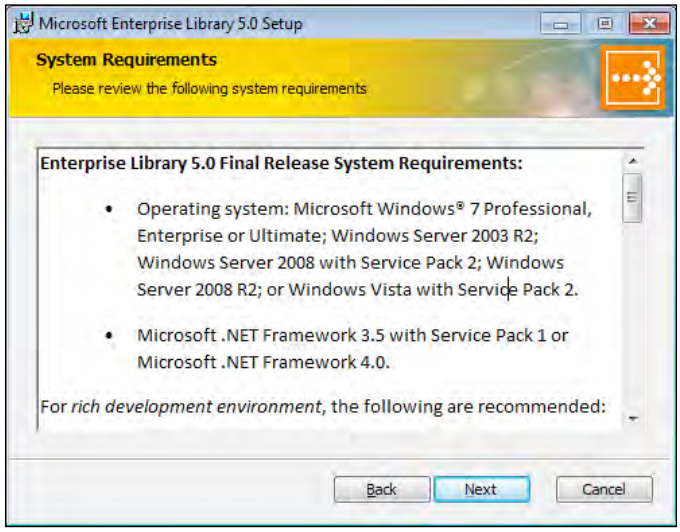
It is recommended to install the source code of Enterprise Library; the source code gives lot of insight in to the how each application is implemented and the best practices adopted by the Enterprise Library team.

1. Double-click the installation file to run Microsoft Enterprise Library 5.0 Setup. The following screenshot with the welcome message will be loaded; click **Next** to move to the next step of the wizard.

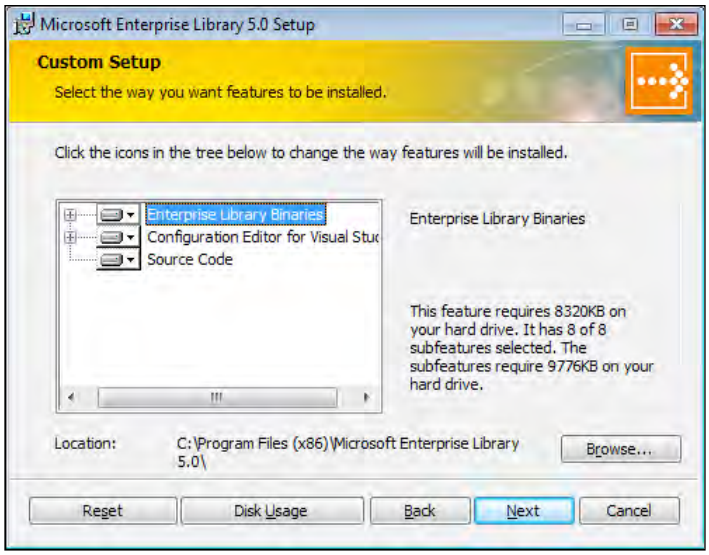


2. The **End-User License Agreement** screen is displayed as shown in the following screenshot. It is important and a good practice to read and fully understand the license agreement of any software we use to develop applications. Once we are satisfied with the license terms, we may click **Next** to move forward to the next installation step.

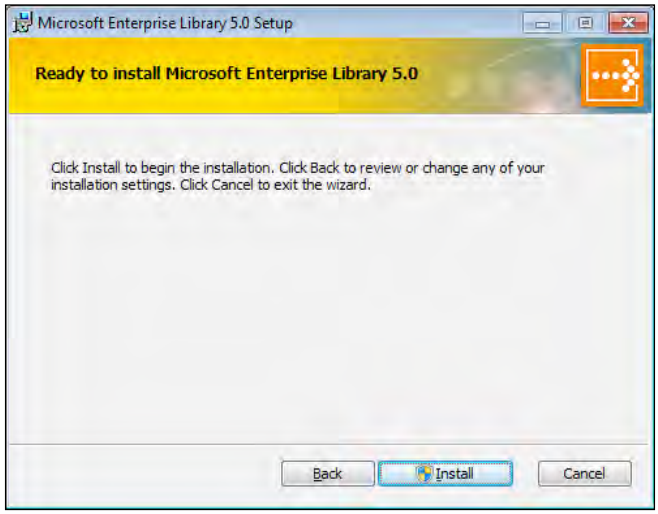
3. The following step in the installation wizard displays the system requirements and comes in quite handy to verify whether the system satisfies the minimum requirements. Click **Next** to move to the next installation step.



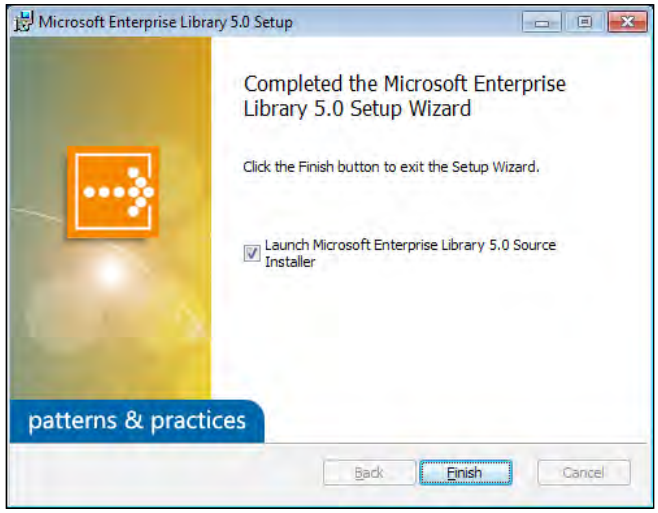
4. Once we click **Next** we are presented with a feature selection screen as shown in the following screenshot. The installer provides control over installation of the binaries of each individual application block; for the purposes of this demonstration we will install all the features.



5. Once we click **Next**, we will be presented with the **Ready to install Microsoft Enterprise Library 5.0** screen as shown in the following screenshot. Clicking on the **Install** button will initiate the installation process.



6. After the installation is completed the following screen will be shown; we may install the source code of Enterprise Library by selecting the checkbox **Launch Microsoft Enterprise Library 5.0 Source Installer** and clicking on the **Finish** button. Provide the appropriate install location for the source code and the installer will copy the source code and additionally build the assemblies if we choose.



Summary

In this chapter we got introduced to the Enterprise Library and explored various application blocks: the Unity, Policy Injection, Data Access, Logging, Exception Handling, Caching, Validation, Security, and Cryptography blocks. We discussed the dependencies between the functional application blocks. We understood the system requirements and explored the installation steps of Enterprise Library. In the next chapter we will explore the Data Access block in detail.

2
Data Access Application Block

A **Relational Database Management System** (**RDBMS**) is the most common and preferred storage mechanism for enterprise applications. **ADO.NET** is the cornerstone while working with databases on the **.NET** platform; it provides the framework and implementations for several databases. Developers leveraging ADO.NET often have to write boilerplate code over and over again. While performing database operations, this might lead to lower productivity, inefficient code, connection leakage, and so on.

The **Data Access Application Block** abstracts developers from the underlying database technology by providing a common interface to perform database operations. It simplifies common data access functionality by taking care of the mundane tasks like creating a connection object, opening and closing a connection, parameter caching, and so on. The Data Access block supports all the features supported by ADO.NET; it goes a step further by bringing consistency and simplifying the common database tasks.

The benefits of the Data Access block are as follows:

- It reduces the plumbing code to perform common tasks.
- It builds on top of the functionality provided by ADO.NET, so both ADO.NET's and application block's functionality are available at our disposal.
- It allows changing the database type without changing or re-compiling the application code.
- It provides parameter caching for all databases and implements simple connection pooling for **SQL CE** as well.
- It increases developer productivity through its rich set of methods, which reduces the data access code considerably.

In this chapter, you will:

- Get to know the key elements of the Data Access block
- Reference the required and optional assemblies
- Configure Data Access settings using the configuration editor
- Add a namespace for convenience
- Create Data Access block objects
- Retrieve records using `ExecuteReader` and `ExecuteDataSet`
- Retrieve a record as an Object
- Retrieve multiple records as an Object Collection
- Retrieve records as XML using `ExecuteXmlReader`
- Execute a command using `ExecuteNonQuery`
- Leverage output parameters
- Access a scalar result using `ExecuteScalar`
- Update records using `DataSet`
- Work with Transactions

Working of Data Access Application Block

It takes two to tango, and in this case it takes configuration and application code. We configure the database connection string and set the provider name attribute; this attribute is mandatory for the Data Access block to work. Optionally, we may also set the default database instance attribute part of the **Database Settings**. In the application code, there are several elements involved in making the Data Access block work, but it all begins with a class called `Database`. The `Database` class abstracts us from the underlying database technology and provides us with a simple model to perform various actions against the configured database. It internally leverages the ADO.NET provider factory model (`DbProviderFactory`); an instance of `Database` contains a reference to a concrete `DbProviderFactory` object, which exposes common ADO.NET objects such as `DbConnection`, `DbCommand`, `DbDataAdapter`, `DbDataReader`, and so on. The `DbProviderFactory` class is an abstract class, part of the `System.Data.Common` namespace.

The following is a partial list of concrete implementations of `DbProviderFactory`.

Provider name	Provider Factory Implementation
`System.Data.SqlClient`	`SqlClientFactory`
`System.Data.SqlServerCe.3.5`	`SqlCeProviderFactory`
`System.Data.OracleClient`	`OracleClientFactory`
`System.Data.EntityClient`	`EntityProviderFactory`
`System.Data.OleDb`	`OleDbFactory`
`System.Data.Odbc`	`OdbcFactory`

Ever noticed an attribute called `providerName` in the connection string entry? This information is used to construct the appropriate provider factory object and that's the reason it is a required attribute as far the Data Access block is concerned. Data Access block configuration code contains default mappings for the data providers. `System.Data.SqlClient` data provider maps with `SqlDatabase`, `System.Data.OracleClient` data provider maps with `OracleDatabase`, and `GenericDatabase` is used for all other data providers.

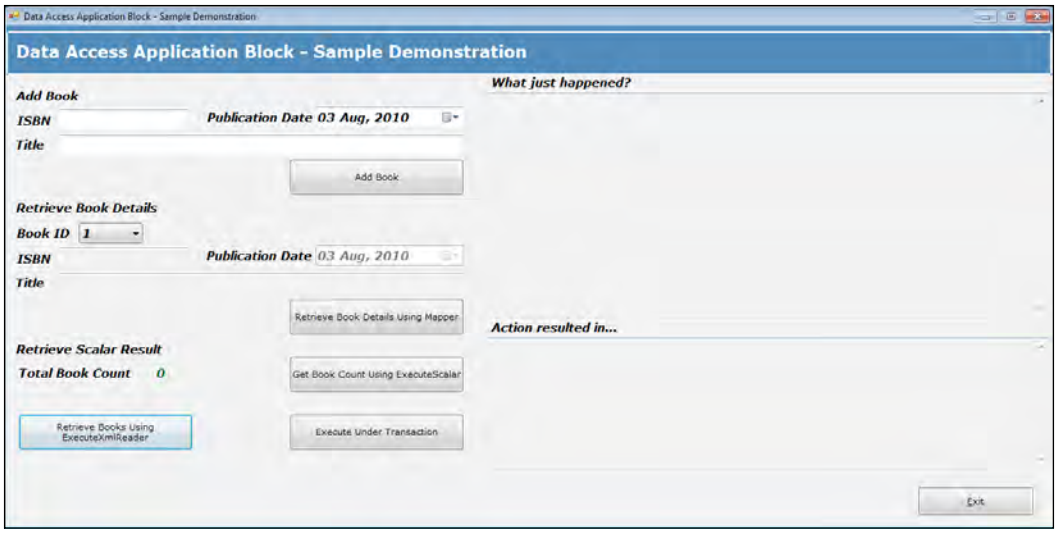
There is an active community called "EntLib Contrib" developing a library of extensions for Enterprise Library. Many third-party providers are available such as SQLite, Oracle (ODP.NET), MySql, IBM DB2, and PostgreSQL databases. Visit `http://entlibcontrib.codeplex.com` for more information.

Developing an application

We will explore each individual Data Access block feature and along the way we will understand the concepts behind the individual elements. This will help us to get up to speed with the basics. To get started, we will do the following:

- Reference the Data Access block assemblies
- Configure Data Access settings
- Add the required Namespace
- Create an instance of `Database`
- Perform actions using the `Database` instance

To complement the concepts and sample code of this book and allow you to gain quick hands-on experience of different features of the Data Access block, we have created a sample demonstration application. The following is a screenshot of the sample application:



Referencing the required assemblies

For the purposes of this demonstration we will be referencing non-strong-named assemblies but based on individual requirements, Microsoft strong-named assemblies or a modified set of custom assemblies can be referenced as well.

The following table lists the required assemblies:

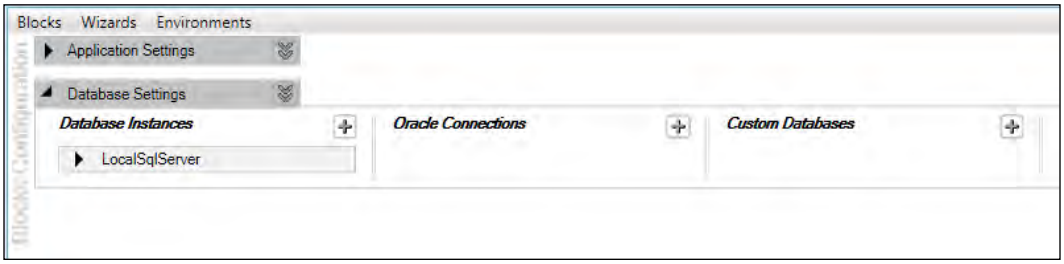
Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Data.dll`	Required

Adding Data Access Settings

Before we can leverage the features of the Data Access block we have to add the initial **Data Access Settings** to the configuration. The following steps will add the settings to the configuration file.

1. Open the Enterprise Library configuration editor either using the shortcut available in **Start | All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration** or by just right-click the configuration file in the **Solution Explorer** window of **Visual Studio IDE**.
2. Next click on **Edit Enterprise Library V5 Configuration**. Initially, we will have a blank configuration file with default **Application Settings** and **Database Settings**.

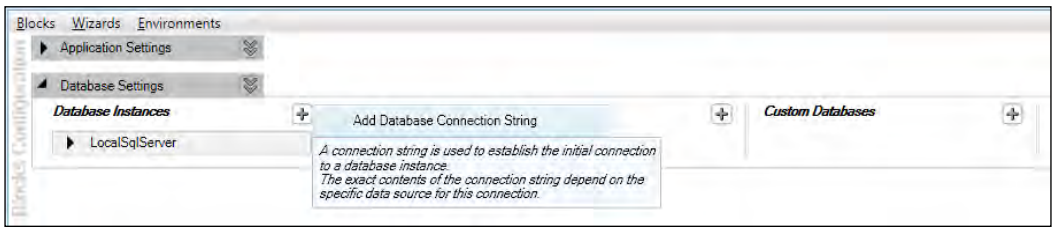
The following screenshot shows the default configuration settings:



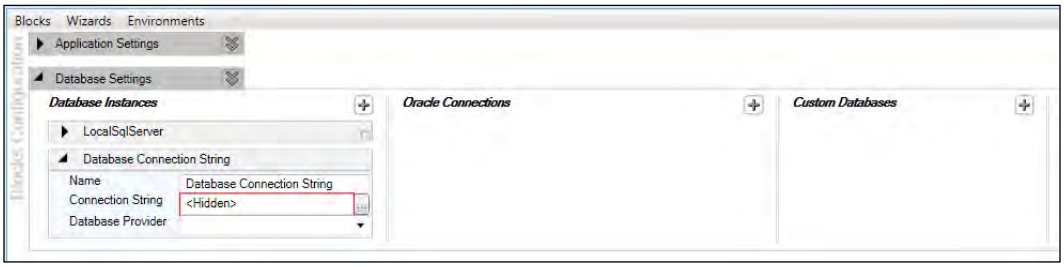
Database Settings configuration is already loaded and consists of three sections: **Database Instances**, **Oracle Connections**, and **Custom Databases**. Let us go ahead and configure the connection string in the **Database Instances** section.

1 Click on the plus symbol provided on the top right corner of the Database Instances section and click on the **Add Database Connection String** menu item.

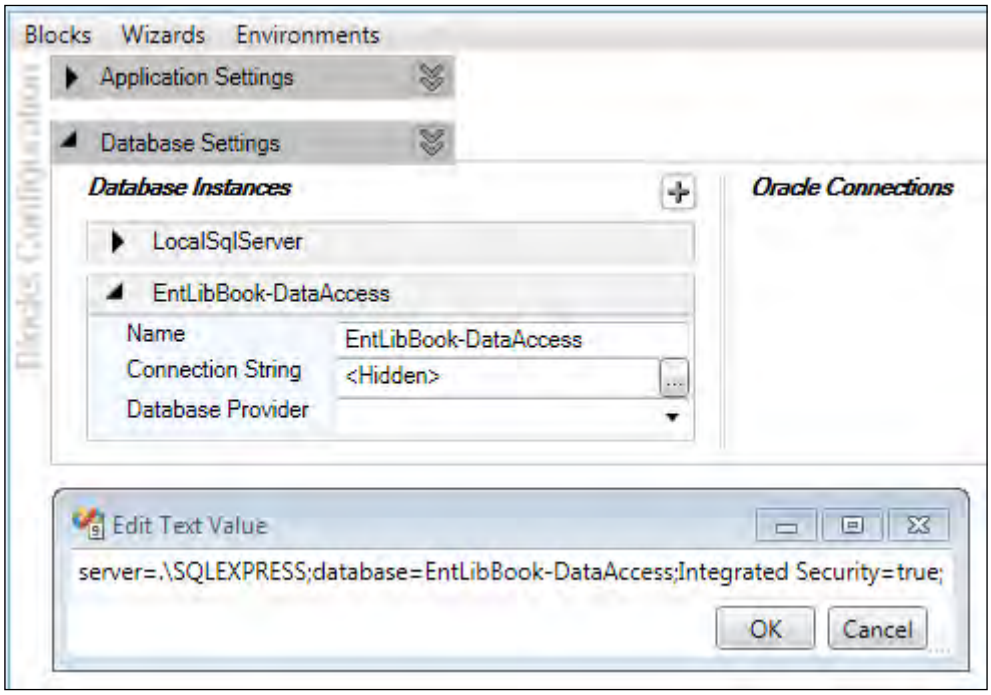
The following screenshot shows the menu option **Add Database Connection String**:



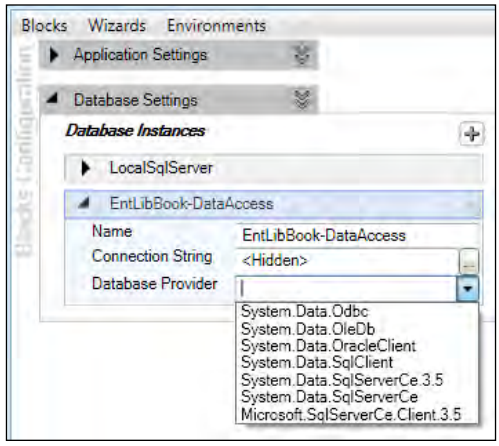
2. Once we click on the **Add Database Connection String**, the configuration editor will add a new connection string as shown in the following screenshot:



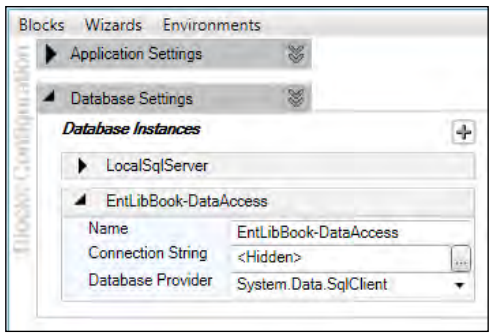
3. The configuration editor has added a connection string section with **Name** as **Database Connection String** and an empty value and database provider.
4. Change the **Name** property to the name of your choice. The **Name** property can be used to create an instance of `Database`.
5. Next click on the ellipsis button provided against the **Connection String** property. This action will pop up a small **Edit Text Value** dialog as shown in the following screenshot. Enter the database connection string you wish to connect to while leveraging the **Data Access Application Block**.



6. Next select the appropriate **Database Provider** from the drop-down list of providers. For the purposes of this demonstration, we will be using SQL Server database and so we will select `System.Data.SqlClient` database provider.



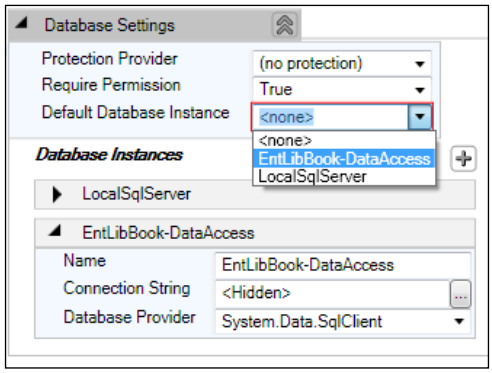
7. The following screenshot shows the selected **Database Provider**:



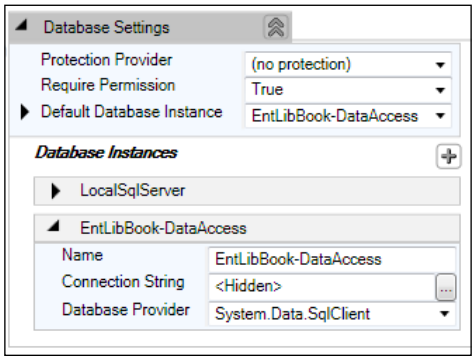
Although this step is optional, it helps in creating an instance of `Database` without providing the database instance name for the object construction. This basically means creating a default `Database` instance.

1. Click on the arrow (representing expand/collapse) provided on the right side of the **Database Settings**; this will allow us to configure the **Default Database Instance** property.
2. Next, select the **Default Database Instance** you wish to configure from the drop-down list.

The following screenshot shows the selection of the **Default Database Instance** configuration option:



The following screenshot shows the selected **Default Database Instance**:



The configuration editor generates the corresponding XML in the configuration file. The following is the output of the configuration; certain values are truncated for readability.

```
[XML Configuration]
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <configSections>
        <section name="dataConfiguration" type="Microsoft.Practices.
EnterpriseLibrary.Data.Configuration.DatabaseSettings, Microsoft.
Practices.EnterpriseLibrary.Data, Version=5.0.414.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35" requirePermission="true" />
    </configSections>
    <dataConfiguration defaultDatabase="EntLibBook-DataAccess" />
    <connectionStrings>
```

```
        <add name="EntLibBook-DataAccess" connectionString="server=.\
SQLEXPRESS;database=EntLibBook-DataAccess;Integrated Security=true;"
            providerName="System.Data.SqlClient" />
    </connectionStrings>
</configuration>
```

Adding a namespace

We definitely don't want to get bored by fully qualifying the type on every instance of its usage, so to make our life easy we can add the namespace given below to the source code file to use the Data Access block elements without fully qualifying the reference. Although we will be using `EnterpriseLibraryContainer` to instantiate objects (so we will also add `Microsoft.Practices.EnterpriseLibrary.Common.Configuration` namespace to the source file), the Unity Namespace section is listed to make you aware of the availability of the alternate approach of instantiating objects.

Core Namespace:

- `Microsoft.Practices.EnterpriseLibrary.Data`

Common Data Related Namespace:

- `System.Data`
- `System.Data.Common`

Configuration Namespace (Optional): Required while using the `EnterpriseLibraryContainer` to instantiate objects.

- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`

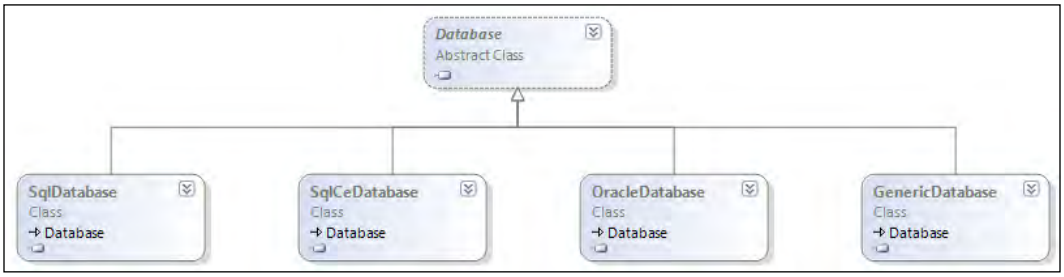
Unity Namespace (Optional): Required while instantiating objects using the Unity container.

- `System.Configuration`
- `Microsoft.Practices.Unity`
- `Microsoft.Practices.Unity.Configuration`

Understanding the Database class

`Database` is an abstract class part of `Microsoft.Practices.EnterpriseLibrary.Data` namespace, this class is in the heart of the action. When we generate an instance of this class based on the configuration, we get the respective concrete implementation. This class provides several virtual (Overridable in Visual Basic) properties and methods, default behavior/logic is implemented, and this provides flexibility to derived classes to override the behavior/logic based on the individual requirements. It exposes several utility methods such as `CreateConnection`, `CreateParameter`, `GetDataAdapter`, `GetParameterValue`, `GetSqlStringCommand`, `GetStoredProcCommand`, and so on; these methods have several helpful overloads as well. Apart from these, it also provides methods such as `ExecuteReader`, `ExecuteNonQuery`, `ExecuteDataSet`, `ExecuteScalar`, `LoadDataSet`, `UpdateDataSet`, and so on to perform **CRUD** (**Create, Read, Update, Delete**) operations.

The following diagram shows the `Database` class and the derived classes:



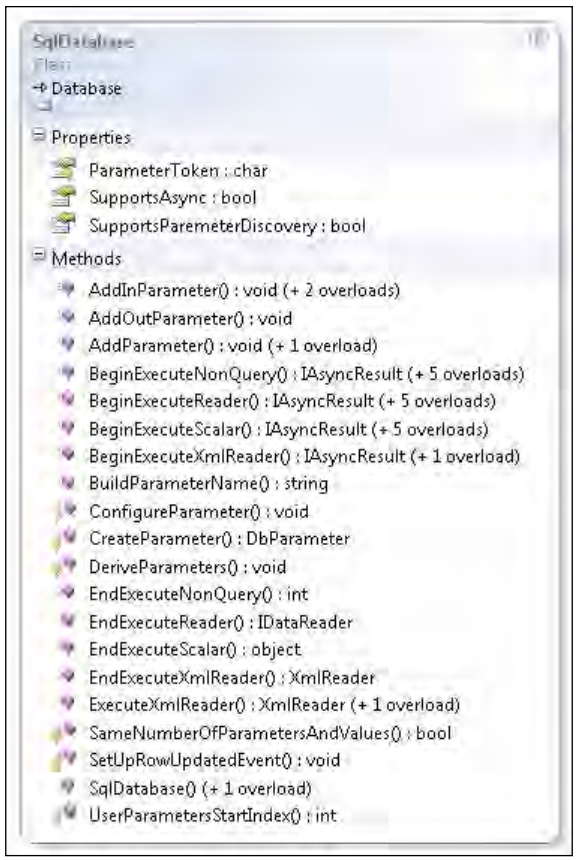
The Data Access block provides parameter caching services for stored procedures; while executing the command more than once, parameter caching eliminates the round trip to the database to get the parameters and types. The `ParameterCache` class is internally used by the abstract `Database` class to cache parameters. `CachingMechanism` is an `internal` class, which provides the actual caching functionality to the `ParameterCache` class.

The following class diagram shows the methods exposed by the `ParameterCache` class:

SqlDatabase class

The `SqlDatabase` class inherits from the `Database` class and is part of the `Microsoft.Practices.EnterpriseLibrary.Data.Sql` namespace. This class represents an SQL Server database and uses SQL Server .NET managed provider `System.Data.SqlClient` to connect and perform operations on an SQL Server database. This class overrides several properties and methods and provides implementation specific to SQL Server database. Properties such as `SupportsAsync` (returns true), `ParameterToken` (returns @), `SupportsParemeterDiscovery` (returns true), and so on are overridden. Also it overrides methods such as `BeginExecuteNonQuery`, `BeginExecuteReader`, `BeginExecuteScalar`, and corresponding "End" methods to leverage asynchronous execution. Additionally, it adds methods such as `ExecuteXmlReader`, `BeginExecuteXmlReader`, and `EndExecuteXmlReader` to expose functionality specific to SQL Server database.

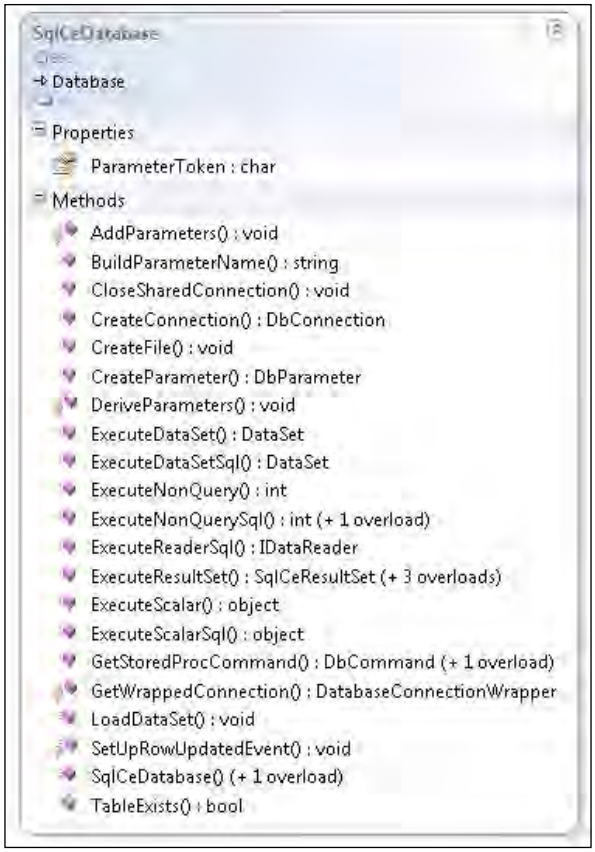
The following class diagram shows the properties and methods exposed by the `SqlDatabase` class:



SqlCeDatabase class

The `SqlCeDatabase` class inherits from the `Database` class and is part of the `Microsoft.Practices.EnterpriseLibrary.Data.SqlCe` namespace. This class provides implementation to work with SQL Server Compact Edition database. SQL Server CE doesn't provide any connection pooling and so the cost of opening the initial connection is high; this class provides a simple connection pooling implementation to improve the performance. The implementation overrides several methods and provides its own logic specific to SQL Server CE database. It is to be noted that since SQL Server CE doesn't support stored procedures, all the methods related to stored procedures will throw an exception of type `NotImplementedException`. Instead, use the methods ending with **Sql** such as `ExecuteDataSetSql`, `ExecuteNonQuerySql`, `ExecuteReaderSql`, `ExecuteScalarSql`, and so on. This class also adds additional utility methods such as `CreateFile`, `CloseSharedConnection`, and so on.

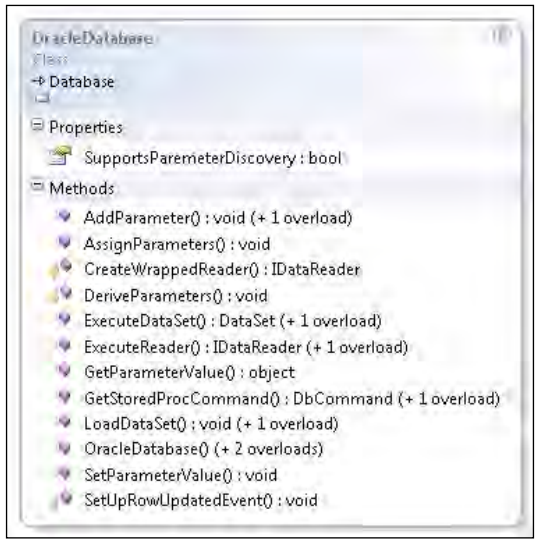
The following class diagram shows the properties and methods exposed by the `SqlCeDatabase` class:



OracleDatabase class

The `OracleDatabase` class inherits from the `Database` class and is part of the `Microsoft.Practices.EnterpriseLibrary.Data.Oracle` namespace. This class provides implementation to access and perform **CRUD** operations on an Oracle database. It internally leverages Oracle .NET Managed Provider `System.Data.OracleClient` to connect and perform operations on an Oracle 9*i* database.

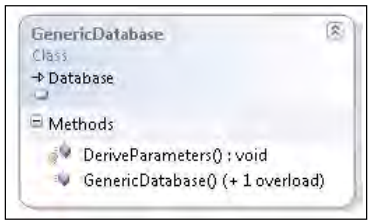
The following class diagram shows the properties and methods exposed by the `OracleDatabase` class:



GenericDatabase class

The `GenericDatabase` class also inherits from the `Database` class and is part of the `Microsoft.Practices.EnterpriseLibrary.Data` namespace. This class doesn't provide any specific behavior and is used when none of the other concrete implementations of `Database` maps to the specific data provider. It overrides only one method; the `protected DeriveParameters` method is overridden and it throws an exception of type `NotSupportedException`. Being a generic implementation, there is obviously no generic way and support for parameter discovery.

The following class diagram shows the methods exposed by `GenericDatabase` class:



Creating a Database instance

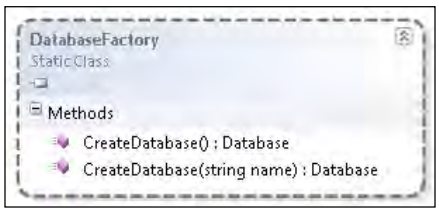
We have several options at hand while creating a `Database` object such as using the static `DatabaseFactory` class, using Unity Service Locator, and using Unity container directly. A few approaches such as configuring the container through configuration file or code are not listed here but the recommended approach is either to use the Unity Service Locator for applications with few dependencies or create objects using Unity container directly to leverage the benefits of this approach. Use of the static factory class is not recommended.

More information on Unity Container and Service Locator is available at `http://msdn.microsoft.com/en-us/library/ff664535(PandP.50).aspx`.

Using the DatabaseFactory class

`DatabaseFactory` is a static class and is part of the `Microsoft.Practices.EnterpriseLibrary.Data` namespace. This class contains factory methods for creating `Database` objects; it exposes a method called `CreateDatabase` with an overload accepting the connection string name. Internally, it leverages `EnterpriseLibraryContainer`, which is part of the `Microsoft.Practices.EnterpriseLibrary.Common.Configuration` namespace; this class is an entry point for the container infrastructure for Enterprise Library.

The following class diagram shows the methods exposed by the `DatabaseFactory` class:



Static factory classes were the default approach to creating objects with versions prior to 5.0. This approach is no longer recommended and is still available for backward compatibility.

The following is the syntax to create a deafult instance of `Database` using the `DatabaseFactory` class:

```
Database db = DatabaseFactory.CreateDatabase();
```

The following is the syntax to create a named instance of `Database` using the `DatabaseFactory` class:

```
Database db = DatabaseFactory.CreateDatabase
                    ("EntLibBook-DataAccess");
```

Using Unity service locator

This approach is recommended for applications with few dependencies. The `EnterpriseLibraryContainer` class exposes a static property called `Current` of type `IServiceLocator`, which resolves and gets an instance of the specified type.

The following is the syntax to create a deafult instance of `Database` using Unity service locator:

```
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();
```

The following is the syntax to create a named instance of `Database` using Unity service locator:

```
Database db = EnterpriseLibraryContainer.Current.GetInstance<Database>
("EntLibBook-DataAccess");
```

Using Unity container directly

Larger complex applications demand looser coupling. This approach leverages the dependency injection mechanism to create objects instead of explicitly creating instances of concrete implementations. Unity container resolves objects using type registrations and mappings; these can be configured programmatically or through a configuration file and based on the configuration it resolves the appropriate type whenever requested. The following example instantiates a new Unity container object and adds the Enterprise Library Core Extension. This loads the configuration and makes registrations and mappings of Enterprise Library available.

The following is the syntax to create a default `Database` instance directly using Unity container:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
Database database = container.Resolve<Database>();
```

The following is the syntax to create a named `Database` instance directly using Unity container:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
Database database = container.Resolve<Database>("EntLibBook-
DataAccess");
```

Retrieving records using ExecuteReader

Retrieving records is one of the most common database operations and the Data Access block provides several different ways to retrieve data. The `ExecuteReader` method allows us to execute a database command and returns an object implementing the `IDataReader` interface. This provides us a way to read records as a read-only and forward-only stream of rows.

The following code block shows records retrieval using `ExecuteReader`:

```
//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Step 2: Create Database Command - SQL String
DbCommand dbCommand = db.GetSqlStringCommand("SELECT CustomerID,
FirstName, LastName FROM Customers WHERE CustomerID = @CustomerID");

//Step 3: Add Input Parameters
db.AddInParameter(dbCommand, "CustomerID", DbType.Int32, 1);

//Step 4: Execute Query
using (IDataReader reader = db.ExecuteReader(dbCommand))
{
    // Read Data and map to business entity
}
```

We created an instance of `Database` using `EnterpriseLibraryContainer`. Since we are executing a query with parameters we created `DbCommand` object using the `GetSqlStringCommand` method of `Database`. Next, we added the input parameter using the `AddInParameter` method of `Database`, then we took the final step of executing the command using the `ExecuteReader` method. This method returns `IDataReader`. The data reader will be closed as it is wrapped with a `using` statement and the connection will be closed automatically.

Retrieving records using DataSet

Records can be retrieved as a `DataSet` by invoking the `ExecuteDataSet` method of the `Database`; also `LoadDataSet` can be used to load the data to an existing `DataSet`.

The following code block shows record retrieval using `ExecuteDataSet`:

```
//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Step 2: Create Database Command - SQL String
DbCommand dbCommand = db.GetSqlStringCommand("SELECT CustomerID,
FirstName, LastName FROM Customers");

//Step 3: Execute Query
DataSet categoryDataSet = db.ExecuteDataSet(dbCommand);
```

The given code block demonstrates retrieving a `DataSet` from the `Customers` table using a simple SQL query and executing against the configured database.

Retrieving a record as an object

While working with data it is quite common to retrieve data and store it in a business/data entity. Generally, this is achieved by looping through the records and mapping each column with the corresponding property of the business/data entity. The `Database` class provides two methods, `ExecuteSprocAccessor` and `ExecuteSqlAccessor`, to return the result as an object. The accessor can also be created separately using the `CreateSprocAccessor` and `CreateSqlStringAccessor` methods and executed by calling its `Execute` method. The accessor uses a parameter mapper to map parameters and an output mapper to map the columns to the properties of the specified object.

Parameter mappers

The `Database` class exposes several methods that accept a stored procedure name and an `object[]` for parameter values. These methods construct a `DbParameter` object using the parameter value and the information obtained by executing the **ADO.NET** `DeriveParameters` method to discover the parameters required by the procedure. Default mapping uses the position to map stored procedure parameters to the values in the `object[]`.

The following are the methods that accept the parameter values as `object[]`:

```
public virtual DataSet ExecuteDataSet(string storedProcedureName,
params object[] parameterValues);

public virtual DataSet ExecuteDataSet(DbTransaction transaction,
string storedProcedureName,params object[] parameterValues);

public virtual int ExecuteNonQuery(string storedProcedureName, params
object[] parameterValues);

public virtual int ExecuteNonQuery(DbTransaction transaction, string
storedProcedureName, params object[] parameterValues);

public IDataReader ExecuteReader(string storedProcedureName, params
object[] parameterValues);

public IDataReader ExecuteReader(DbTransaction transaction, string
storedProcedureName, params object[] parameterValues);

public virtual object ExecuteScalar(string storedProcedureName, params
object[] parameterValues);

public virtual object ExecuteScalar(DbTransaction transaction, string
storedProcedureName, params object[] parameterValues);

public virtual DbCommand GetStoredProcCommand(string
storedProcedureName, params object[] parameterValues);

public virtual void LoadDataSet(string storedProcedureName, DataSet
dataSet, string[] tableNames, params object[] parameterValues);

public virtual void LoadDataSet(DbTransaction transaction, string
storedProcedureName, DataSet dataSet, string[] tableNames, params
object[] parameterValues);
```

```
public virtual IAsyncResult BeginExecuteNonQuery(string
storedProcedureName, AsyncCallback callback, object state, params
object[] parameterValues);

public virtual IAsyncResult BeginExecuteNonQuery(DbTransaction
transaction, string storedProcedureName, AsyncCallback callback,
object state, params object[] parameterValues);

public virtual IAsyncResult BeginExecuteReader(string
storedProcedureName, AsyncCallback callback, object state, params
object[] parameterValues);

public virtual IAsyncResult BeginExecuteReader(DbTransaction
transaction, string storedProcedureName, AsyncCallback callback,
object state, params object[] parameterValues);

public virtual IAsyncResult BeginExecuteScalar(string
storedProcedureName, AsyncCallback callback, object state, params
object[] parameterValues);

public virtual IAsyncResult BeginExecuteScalar(DbTransaction
transaction, string storedProcedureName, AsyncCallback callback,
object state, params object[] parameterValues);
```

While default mapping is useful, there might be circumstances where we might want to create a custom mapping. We can create a custom parameter mapper by inheriting from the `IParameterMapper` interface; we have to provide implementation for the `AssignParameters` method, which holds the logic for custom parameter mapping.

The following code block shows a simple implementation of a custom parameter mapper:

```
public class CustomerParameterMapper : IParameterMapper
{
    public void AssignParameters(DbCommand command, object[]
parameterValues)
    {
        DbParameter parameter = command.CreateParameter();
        parameter.ParameterName = "@CustomerID";
        parameter.Value = parameterValues[0];
        command.Parameters.Add(parameter);
    }
}
```

The following class diagram shows the method exposed by the `IParameterMapper` interface:

Output mappers

The output mapper is a very useful feature that allows us to map the columns of a record from database to the property of an object. We have several options to map the record(s) to object(s) such as the default row mapper, custom row mapping using the `MapBuilder` class, custom row mapping using `IRowMapper<TResult>`, and using `IResultSetMapper<TResult>` for mapping a hierarchy of objects.

Default row mappers

The default row mapper simply matches each property of the provided object type with the columns on the retrieved result. This is done based on the names of the column and property; hence, this approach requires the column and property names to be the same.

Row mapping using MapBuilder

The Database Access block provides a class called `MapBuilder` that makes it very easy to create custom output mapping. It has several methods that help in mapping column names with property names; this information is used to create entity objects.

The following is a sample mapping to demonstrate the power of this approach:

```
IRowMapper<Customer> rowMapper = MapBuilder<Customer>.
MapNoProperties().Map(c => c.ID).ToColumn("CustomerID").Build();
```

Row mapping using IRowMapper<TResult>

We can write a mapping class by inheriting from the `IRowMapper<TResult>` interface; this interface provides a `MapRow` method, which will be called by the Data Access block during the mapping process. We have to provide our mapping logic in the `MapRow` method and return the object.

The following is a simple `Customer` class mapping implementation:

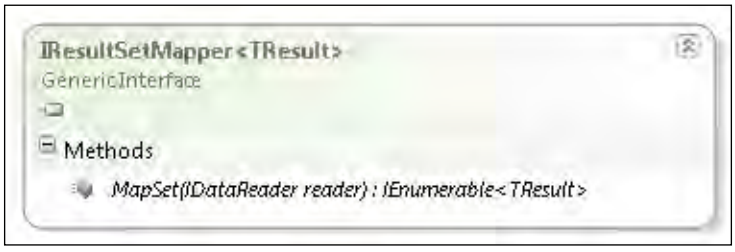
```
public class CustomerRowMapper : IRowMapper<Customer>
{
    public Customer MapRow(IDataRecord row)
    {
        Customer customer = new Customer();
        customer.ID = (int)row["CustomerID"];
        customer.FirstName = row["FirstName"] as string;
        customer.LastName = row["LastName"] as string;

        return customer;
    }
}
```

The following class diagram shows the method exposed by the `IRowMapper` interface:

Result Set mappers

Row mapping generates a single instance of the object type; there might be scenarios in which we want to create an entire object hierarchy of a simple or complex graph. For this very purpose, the Data Access block provides the `IResultSetMapper<TResult>` interface, which has a `MapSet` method. We have to provide custom mapping logic in the `MapSet` method and return the object.

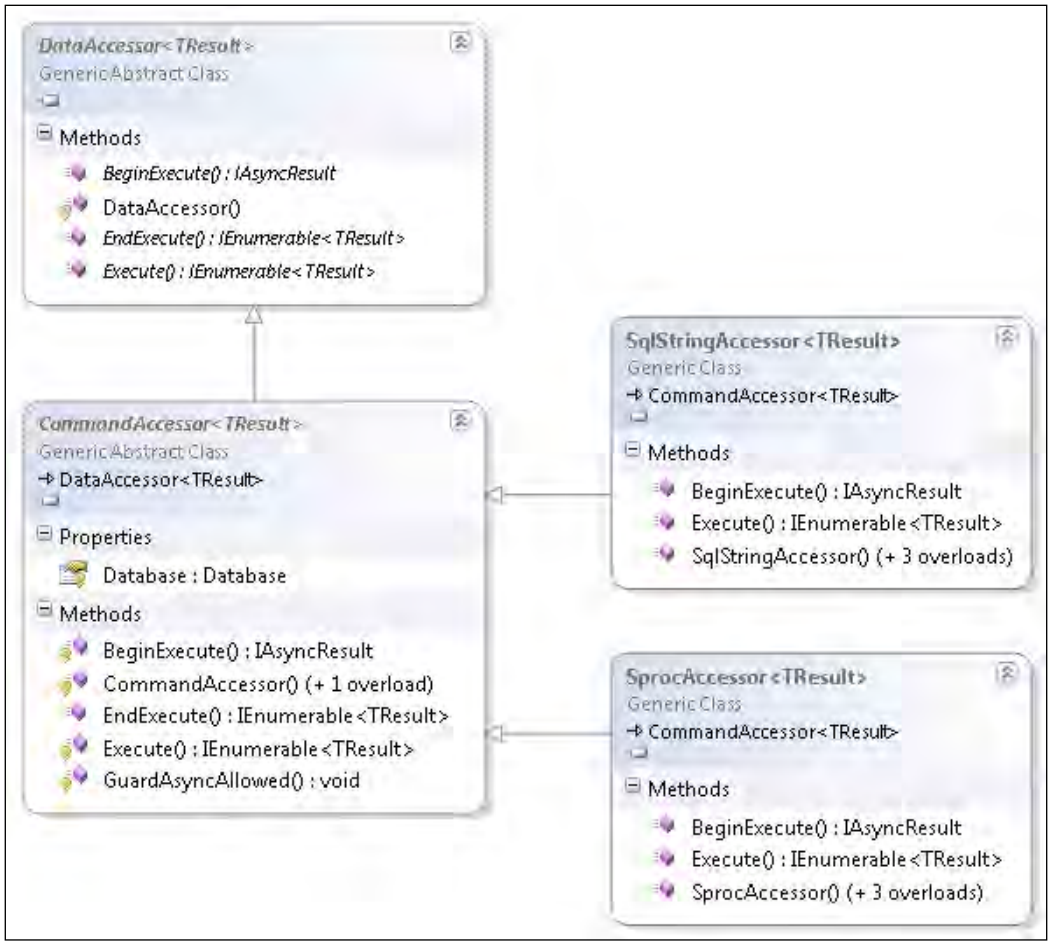
The following class diagram shows the method exposed by the `IResultSetMapper` interface:



Data Accessors

Accessors is the means through which we leverage the parameter and output mapper functionality. It executes the specified query using the parameter values and an optional parameter mapper and returns the result as an object of the specified type. The Database Access block provides two types of accessors: **SQL String Accessor** and **Stored Procedure Accessor**.

The following class diagram shows Data Accessor-related classes, inheritance hierarchy, and the methods exposed by each class:



Creating and executing Accessors

The following code snippet creates a row mapper that maps the properties of the `Book` class to the columns retrieved by the stored procedure. This definition gives enough information to the Data Access block to generate an object of type `Book`. Once the `IRowMapper` object is ready, we create an accessor object using the `CreateSprocAccessor` (`CreateSqlStringAccessor` for an SQL string) method of `Database`. While creating this object, we are specifying the stored procedure name and passing the `IRowMapper` object. Parameter mapping information is not explicitly passed but other overloaded methods of `CreateSprocAccessor` (`CreateSqlStringAccessor` for an SQL string) accept an `IParameterMapper` object. Finally, we use the accessor object and execute it while passing the parameter value of `BookID`.

The following code block shows the usage of `IRowMapper` with `MapBuilder` to map column name with properties of the class and finally retrieve object using `CreateSprocAccessor` method:

```
Book book = null;

//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Step 2: Create Row Mapper
IRowMapper<Book> rowMapper = MapBuilder<Book>.MapNoProperties()
                            .Map(b => b.BookID).ToColumn("BookID")
                            .Map(b => b.ISBN).ToColumn("ISBN")
                            .Map(b => b.Title).ToColumn("Title")
                            .Map(b => b.PublicationDate).
ToColumn("PublicationDate")
                            .Build();

//Step 3: Create Accessor
var accessor = db.CreateSprocAccessor<Book>("usp_get_Book",
rowMapper);

//Step 4: Execute
book = accessor.Execute(id).SingleOrDefault();
```

Retrieving multiple records as an object collection

While a row mapper is used to generate a single instance of the object type specified for each row, there are situations where we want to map a complex hierarchy of objects. A custom result set mapper class can be written for this purpose by implementing the `IResultSetMapper` interface. The following code snippet is for a result set mapper of type `Book`. We just need to implement the mapping logic in the `MapSet` method; we are mapping the properties of the `Book` object with the columns of the `DataReader`.

The following code block shows the custom implementation of result set mapper by implementing `IResultSetMapper` interface:

```
class BookResultSetMapper : IResultSetMapper<Book>
{
    public IEnumerable<Book> MapSet(IDataReader reader)
    {
        List<Book> bookList = new List<Book>();

        while (reader.Read())
        {
            Book book = new Book();
            book.BookID = reader.GetInt32
                                    (reader.GetOrdinal("BookID"));
            book.ISBN = reader.GetString(reader.GetOrdinal("ISBN"));
            book.Title = reader.GetString(reader.GetOrdinal("Title"));
            book.PublicationDate = reader.GetDateTime
                              (reader.GetOrdinal("PublicationDate"));

            bookList.Add(book);
        }

        return bookList;
    }
}
```

The following code snippet demonstrates the usage of the `BookResultSetMapper` class. We create the accessor by passing the SQL string and an instance of `BookResultSetMapper`. Next, we invoke the `Execute` method of the accessor to execute the query and generate the output of type `IEnumerable<Book>`.

```
string sqlString = "SELECT BookID, ISBN, Title, PublicationDate FROM
Books";

//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Step 2: Create Accessor
var accessor = db.CreateSqlStringAccessor<Book>(sqlString, new
BookResultSetMapper());

//Step 3: Execute
var books = accessor.Execute();
```

Retrieving records as XML

Application requirements challenge developers in the least expected ways and there might be a scenario in which we need to retrieve data from the database as XML. SQL Server supports retrieval of data in XML format through a mechanism called `SQLXML`. As this feature is limited to SQL Server, the functionality is only exposed as part of `SqlDatabase` and hence we have to cast the `Database` as `SqlDatabase` to execute the `ExecuteXmlReader` method.

The following code snippet shows the retrieval of records as XML using the `ExecuteXmlReader` method:

```
//Step 1: Create Default Database instance
SqlDatabase db = EnterpriseLibraryContainer.Current.
GetInstance<Database>() as SqlDatabase;

//Step 2: Create Database Command - SQL String
DbCommand dbCommand = db.GetSqlStringCommand("SELECT BookID, ISBN,
Title, PublicationDate FROM Books FOR XML AUTO");

try
{
    //Step 3: Execute Query
    using (XmlReader reader = db.ExecuteXmlReader(dbCommand))
    {
        // Read Data and map to business entity
```

```
            while (!reader.EOF)
            {
                // Read/Process Data
            }
        }
    }
    finally
    {
        //Step 4: Close Connection
        if (dbCommand.Connection != null)
        {
            dbCommand.Connection.Close();
        }
    }
```

This code block executes an SQL statement containing the `FOR XML` statement; `ExecuteXmlReader` returns an `XmlReader` object. Unlike other execute methods that set the command behavior to close the connection when the reader is closed, this method doesn't automatically close the database connection. We have added a `try/finally` block to make sure the connection is closed once we are done reading/processing the data.

Executing a command using ExecuteNonQuery

`ExecuteNonQuery` executes a command and returns the number of records affected. There are six overloaded methods available to meet different needs such as executing an SQL query, a stored procedure, a stored procedure with parameter values, with transaction, and so on.

The following code snippet shows the usage of the `ExecuteNonQuery` method and the retrieval of the output parameter value:

```
//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Step 2: Create Database Command - Stored Procedure
DbCommand dbCommand = db.GetStoredProcCommand("usp_insert_Customer");

//Step 3: Add Input Parameters
db.AddInParameter(dbCommand, "FirstName", DbType.String, "John");
db.AddInParameter(dbCommand, "LastName", DbType.String, "Lennon");
```

```
//Step 4: Add Output Parameter
db.AddOutParameter(dbCommand, "CustomerID", DbType.Int32, int.
MaxValue);

//Step 5: Execute Query
int numberOfRecordsAffected = db.ExecuteNonQuery(dbCommand);

if (numberOfRecordsAffected > 0)
{
    //Step 6: Retrieve Output Parameter Value
    int customerID = (int)db.GetParameterValue(dbCommand,
"CustomerID");
}
```

This code snippet demonstrates a typical usage of `ExecuteNonQuery` where a stored procedure is used to insert a record and we retrieve the primary key value as part of the value of output parameter.

Retrieving scalar values

One of the common requirements while working with databases is to retrieve a single value. The `ExecuteScalar` method provides the ability to execute a command or a query and returns the value of the first column of the first record. Similar to other execute methods, the `ExecuteScalar` method contains six overloads satisfying several different scenarios.

The following code snippet shows the retrieval of scalar value using the `ExecuteScalar` method:

```
//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Step 2: Create Database Command - SQL String
DbCommand dbCommand = db.GetSqlStringCommand("SELECT COUNT(*) FROM
Customers");

//Step 3: Execute Query
int totalCustomers = (int)db.ExecuteScalar(dbCommand);
```

The previous code block demonstrates the typical usage of the `ExecuteScalar` method that retrieves the total record count for a specific SQL query.

Updating records using DataSet

`DataSet` is an in-memory cache of data retrieved from a data source; it is especially very useful during disconnected mode. Records can be added, updated, and deleted in memory; the `DataSet` keeps track of these changes and can be used to make batch updates to the database. Typically a `DataSet` object is created or loaded using the `ExecuteDataSet` or `LoadDataSet` methods respectively. The only difference is that `LoadDataSet` loads data on existing `DataSet` objects; this approach is useful while retrieving data through multiple execution. Once the data is retrieved and records have been added/modified/deleted, the `DataSet` can be passed on to the `UpdateDataSet` method of `Database` to update the database with the changes.

The following code snippet shows a typical record update using `DataSet`:

```
DataSet customerDataSet = new DataSet();

//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//-----------------------------------------------------
//Step 2: Create Database Command to retrieve Customers
DbCommand selectCommand = db.GetSqlStringCommand("Select CustomerID,
FirstName, LastName From Customers");

//Step 3:  Retrieve Customers using LoadDataSet
db.LoadDataSet(selectCommand, customerDataSet, "Customers");
//-----------------------------------------------------

//Step 4: Get the Customer DataTable Object for convenience
DataTable customerTable = customerDataSet.Tables["Customers"];

//-----------------------------------------------------
//Step 5: Create Database Command to insert Customers
DbCommand insertCommand = db.GetSqlStringCommand("INSERT INTO
Customers(FirstName, LastName) VALUES(@FirstName, @LastName)");

//Step 6: Add input parameters to insert Customers
db.AddInParameter(insertCommand, "FirstName", DbType.String,
"FirstName", DataRowVersion.Current);
db.AddInParameter(insertCommand, "LastName", DbType.String,
"LastName", DataRowVersion.Current);

//Add new Customer to the table
```

```
customerTable.Rows.Add(new object[] { DBNull.Value, "Mark", "Twain"
});

//-----------------------------------------------------

//-----------------------------------------------------
//Step 7: Create Database Command to update Customers
DbCommand updateCommand = db.GetSqlStringCommand("UPDATE Customers
SET FirstName = @FirstName, LastName = @LastName WHERE CustomerID = @
CustomerID");

//Step 8: Add input parameters to update Customers
db.AddInParameter(updateCommand, "CustomerID", DbType.Int32,
"CustomerID", DataRowVersion.Current);
db.AddInParameter(updateCommand, "FirstName", DbType.String,
"FirstName", DataRowVersion.Current);
db.AddInParameter(updateCommand, "LastName", DbType.String,
"LastName", DataRowVersion.Current);

//Modifying First & Last Name of Customer
customerTable.Rows[0]["FirstName"] = "Rob";
customerTable.Rows[0]["LastName"] = "Connery";

//-----------------------------------------------------

//-----------------------------------------------------
//Step 9: Add input parameters to delete Customers
DbCommand deleteCommand = db.GetSqlStringCommand("DELETE FROM Customer
WHERE CustomerID = @CustomerID");

//Step 10: Add input parameters to delete Customers
db.AddInParameter(deleteCommand, "CustomerID", DbType.Int32,
"CustomerID", DataRowVersion.Current);

//Deleting Customer
customerTable.Rows[4].Delete();

//-----------------------------------------------------

//-----------------------------------------------------
//Step 11: Update DataSet
int rowsAffected = db.UpdateDataSet(customerDataSet, "Customers",
                                    insertCommand, updateCommand,
                                      deleteCommand,
                                    UpdateBehavior.Standard);
//-----------------------------------------------------
```

Although this code snippet demonstrates this functionality in a single method, the retrieval and modification will be two separate tasks. We have to provide a `DbCommand` object for `Insert`, `Update`, and `Delete` to the `UpdateDataSet` method as these commands are required to perform the appropriate operations. The `Database` class is abstracting us from writing the boilerplate code of creating and executing the `DataAdapter` method to update the data.

Working with transactions

Transaction is an important piece of functionality. While executing multiple operations against the database it is a common requirement to successfully execute all the operations or the database must roll back to the state before the operation began. A typical example is to debit one account with an amount and then credit the same amount into another account. It becomes important for the database to execute both the debit and credit operations successfully or neither of them should be committed to the database.

There are several ways to achieve this functionality, for example, controlling the transaction in a stored procedure using T-SQL statements such as `BEGIN TRANSACTION`, `END TRANSACTION`, and `ROLLBACK TRANSACTION`. `TransactionScope` can also be used to execute several database operations within or across the database. However we will be exploring the transaction support provided by **ADO.NET**; this transaction is initiated explicitly by calling the `BeginTransaction` method on the `DbConnection` object and we explicitly commit or roll back the transaction by calling the `Commit` or `Rollback` method on the instance of `DbTransaction`.

The following code block shows multiple operations performed under a transaction:

```
//Step 1: Create Default Database instance
Database db = EnterpriseLibraryContainer.Current.
GetInstance<Database>();

//Create Database Command Object to perform credit operation
DbCommand creditCommand = db.GetStoredProcCommand("usp_Account_
CreditAmount");
db.AddInParameter(creditCommand, "BankAccountID", DbType.Int32, 1234);
db.AddInParameter(creditCommand, "Amount", DbType.Int32, 5000);

//Create Database Command Object to perform debit operation
DbCommand debitCommand = db.GetStoredProcCommand("usp_Account_
DebitAmount");
db.AddInParameter(debitCommand, "BankAccountID", DbType.Int32, 4321);
db.AddInParameter(debitCommand, "Amount", DbType.Int32, 5000);
```

```
//Step 2: Create Database Connection
using (DbConnection dbConnection = db.CreateConnection())
{
    //Step 3: Open Database Connection
    dbConnection.Open();

    //Step 4: Begin Transaction
    DbTransaction dbTransaction = dbConnection.BeginTransaction();

    try
    {
        //Step 5: Perform Database Operations
        int creditAccountRowsAffected = db.ExecuteNonQuery
          (creditCommand, dbTransaction);
        int debitAccountRowsAffected = db.ExecuteNonQuery
          (debitCommand, dbTransaction);

        if (creditAccountRowsAffected > 0
                                    && debitAccountRowsAffected > 0)
        {
            //Step 6: Commit the transaction
            dbTransaction.Commit();
        }
    }
    catch
    {
        //Exception Occured: Roll back the transaction
        dbTransaction.Rollback();
    }

    dbConnection.Close();
}
```

The previous code snippet demonstrates the transaction mechanism using the built-in support of **ADO.NET** transaction. The `Database` class exposes a method called `CreateConnection` to create a connection, which returns a generic `DbConnection` object, and using this object we explicitly open and close the database connection. This is required as only we are aware of the boundary and the commands to be executed. Once the database connection is opened, we use the connection object to explicitly initiate the transaction by invoking the `BeginTransaction` method. This method returns the `DbTransaction` object. This object will be used to commit or roll back based on the outcome of the query execution. It is a good practice to wrap the query execution block in a `try/catch/finally` block to roll back during exceptions.

Summary

In this chapter, we have learned about the fundamental elements of the Data Access Application Block such as `Database`, `SqlDatabase`, `OracleDatabase`, `SqlCeDatabase`, `GenericDatabase`, Parameter Mapper, Output Mappers, and so on. We have learned about the required assemblies and configuration. We have also explored various ways of data retrieval, adding of input and output parameters, mapping of parameters and rows, and learned to leverage `DbTransaction`. In the next chapter, we will explore, understand, and leverage the Logging block to write messages.

3
Logging Application Block

We spend a lot of time and effort to develop world-class applications; it's as if we are painstakingly creating a virtual life. Unfortunately, this virtual living being (application) cannot send a distress signal to seek help during unforeseen circumstances. Developers have a responsibility to empower the application so that it can leave us a message with enough information to track and resolve the issue. Let's face it, there are millions of things that can go wrong during development, or while the application is in production. While in development, we have the luxury to debug the application and fix the bugs, in production we need a mechanism to flip a switch and make the application tell us "what happened", "when it happened", and so on. Such logging of information is crucial to understand the root cause of an issue and helps in quickly resolving it. Also, logging is not just limited to persisting exception/error messages; it can also be useful for auditing purposes too.

The **Logging Application Block** provides developers with a flexible library that satisfies simple to complex logging requirements. The simple task of logging to a file using the Logging block requires just two lines of code and a simple three or four-click configuration. Developers can categorize log entries (and log them to one or more logging targets) and format them using the available formatters. Logging filters allow developers to enable or disable logging based on category, priority, or if required, disable logging completely. It also provides a mechanism for tracing application activities. Although the Logging block provides lot of options, if required we can leverage extension points to write custom logging targets, log entry formatters, and logging filters to meet specific requirements. The way it works is that our application code sends the logging information (`LogEntry`) using `LogWriter` or a static façade `Logger` class. The log entry consists of a log message and may also contain category, priority, event ID, severity, title, and other additional context information. On the configuration side, we can add categories and associate these categories to one or more destinations called **Logging Target Listeners** (file, database, and so on). Additionally, these listeners can be configured to use a formatter to format the log entry. Before writing the log entry, the Logging block checks whether there are any filter conditions, generally called **Logging Filters**. This helps in controlling logging through configuration file.

In this chapter, you will:

- Be introduced to the Logging Application Block
- Understand the concepts behind the Logging Application Block
- Learn about referencing the required and optional assemblies
- Learn to set up the initial infrastructure configuration using the configuration editor
- Learn about the required and optional namespaces to avoid fully qualifying types
- Explore the design elements of the Logging block
- Learn to leverage the `LogEntry` class
- Learn to use `Logger` and `LogWriter` to write log messages
- Learn to configure Special Categories
- Learn to add and configure Log Categories
- Learn to log messages to the Event Log
- Learn to configure to log messages to a flat file
- Learn to configure to log messages to a series of flat files
- Learn to configure to log messages to a text file in XML format
- Learn to configure to log messages to a database
- Learn to configure to send log messages to an e-mail address
- Learn to configure System Diagnostics Trace Listener
- Learn to configure to send log messages to a Message Queue
- Learn to configure to send log messages to WMI
- Learn to configure Log Message Formatters
- Understand and learn to configure Logging Filters such as Category Filter, Logging Enabled Filter, and Priority Filter
- Learn to implement custom trace listeners, log formatters, and log filters

Developing an application

Before we dig deeper in to individual features of the Logging block, we will touch upon the basic elements by creating a sample application. This will help us to get up-to-speed with the basics. In this section, we will do the following:

- Reference the Logging block assemblies
- Set up the initial configuration
- Write code to log a message

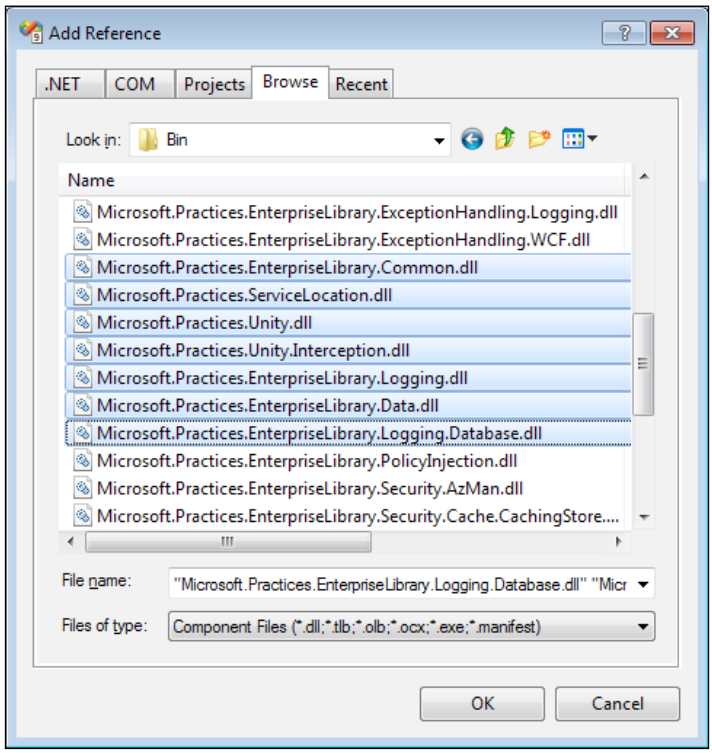
Referencing assemblies

For the purposes of this demonstration, we will be referencing non-strong-named assemblies but based on individual requirements Microsoft strong-named assemblies, or a modified set of custom assemblies can be referenced as well. Since we will also be exploring the configuration of database logging features in this chapter, we will include references to the database logging-related assemblies to the project.

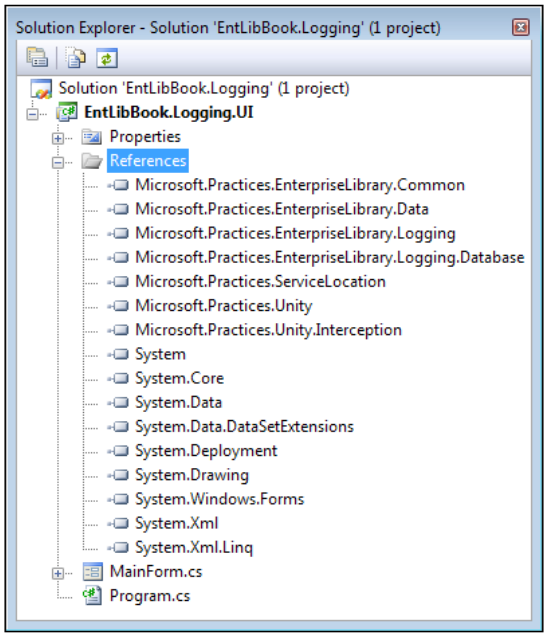
The following table lists the required/optional assemblies.

Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Logging.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Data.dll`	Optional
`Microsoft.Practices.EnterpriseLibrary.Logging.Database.dll`	Only if database logging is required

Open Visual Studio 2008/2010 and create a new sample **Windows Forms Application** by selecting **File | New | Project | Windows Forms Application**, and provide the appropriate name for the solution and the desired project location. Currently, the application will have a default form and assembly references. In the **Solution Explorer** right-click on the **References** section and click on **Add Reference** and go to the **Browse** tab. Next, navigate to the Enterprise Library 5.0 installation location; the default install location is `%Program Files%\Microsoft Enterprise Library 5.0\Bin`. Now, select all the assemblies listed in the previous table. The final assembly selection will look similar to the following screenshot; note that the assemblies have been moved together for your reference.



After clicking on **OK**, the following screenshot displays the **Solution Explorer** listing all the added assemblies.

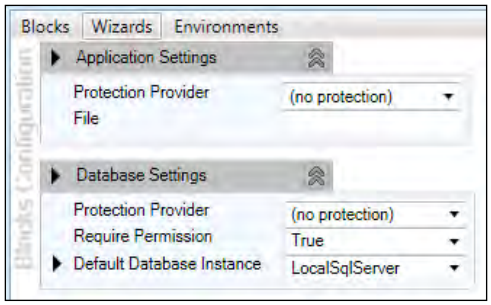


The next step is to add a configuration file to the project. Right-click on the project and navigate and click on the menu **Add** | **New Item**; this will display the **Add New Item** dialog. Select **Application Configuration File** and click on **Add**. This action will add a configuration file named `App.config` to the project. We can now add the Logging settings to the configuration file.

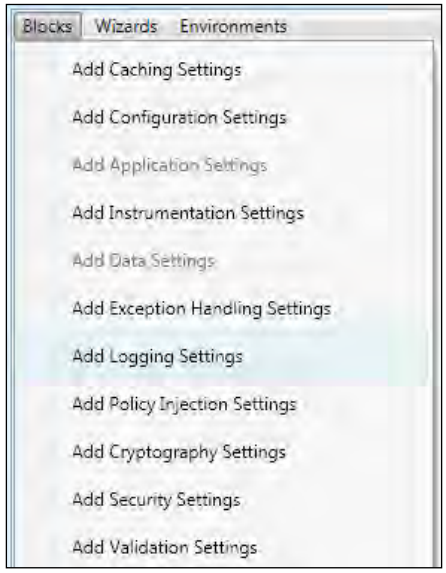
Adding Logging Settings

Before we can leverage the features of the Logging Application Block, we have to add the initial **Logging Settings** to the configuration file. Open the **Enterprise Library configuration editor** either using the shortcut available in **Start** | **All Programs** | **Microsoft patterns & practices** | **Enterprise Library 5.0** | **Enterprise Library Configuration** or by just right-clicking the configuration file in the **Solution Explorer** window of **Visual Studio IDE** and clicking on **Edit Enterprise Library V5 Configuration**. Initially, Enterprise Library configuration editor will display two default sections: **Application Settings** and **Database Settings**.

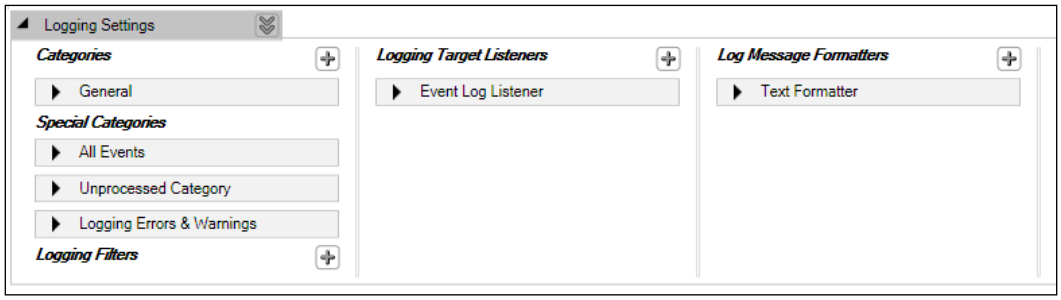
The following screenshot displays the default settings displayed in the configuration editor:



Let us go ahead and add the **Logging Settings** in the configuration file. Select the menu option **Blocks**, which lists several application block settings to be added to the configuration. Click on the **Add Logging Settings** menu item to add the Logging configuration settings.



The following screenshot displays the **Logging Settings** section added to the configuration editor:



Notice the Logging Settings are grouped in to five headings, namely **Categories**, **Special Categories**, **Logging Filters**, **Logging Target Listeners** and **Log Message Formatters**. By default, the settings are configured with a category called **General**, **Logging Target Listener** as **Event Log Listener** and **Log Message Formatter** as **Text Formatter**. We will change the default configuration further; but for now, we are in good shape to leverage the Logging block and write our first log message.

Adding namespaces

Instead of fully qualifying the type on every instance of its usage, we can add the namespaces given below to the source code file to use the Logging block elements without fully qualifying the references. Although we will be using `EnterpriseLibraryContainer` to instantiate objects (so we will also add the `Microsoft.Practices.EnterpriseLibrary.Common.Configuration` namespace to the source file), the Unity Namespace section is listed to make you aware of the availability of the alternative approach to instantiating objects.

Core Namespaces:

- `System.Diagnostics`
- `Microsoft.Practices.EnterpriseLibrary.Logging`

Configuration Namespace (Optional): Required while using the `EnterpriseLibraryContainer` to instantiate objects

- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`

Unity Namespaces (Optional): Required while instantiating objects using **UnityContainer**

- `System.Configuration`
- `Microsoft.Practices.Unity`
- `Microsoft.Practices.Unity.Configuration`

Writing a log message

We are now ready to write our first log message. Since we are using the default configuration, the log message will use the default category **General**, the log entry will be written to the Windows Event Log, and the message will be formatted using the Text Formatter. The first step in writing the log message is to create an instance of `LogWriter`. `LogWriter` is an abstract class and is the primary interface in this release for creating log entries; this abstract class belongs to the `Microsoft.Practices.EnterpriseLibrary.Logging` namespace.

The following code snippet creates the `LogWriter` instance using the `EnterpriseLibraryContainer` class.

```
//Create a LogWriter instance using the EnterpriseLibraryContainer
LogWriter logWriter = EnterpriseLibraryContainer.Current.
GetInstance<LogWriter>();
```

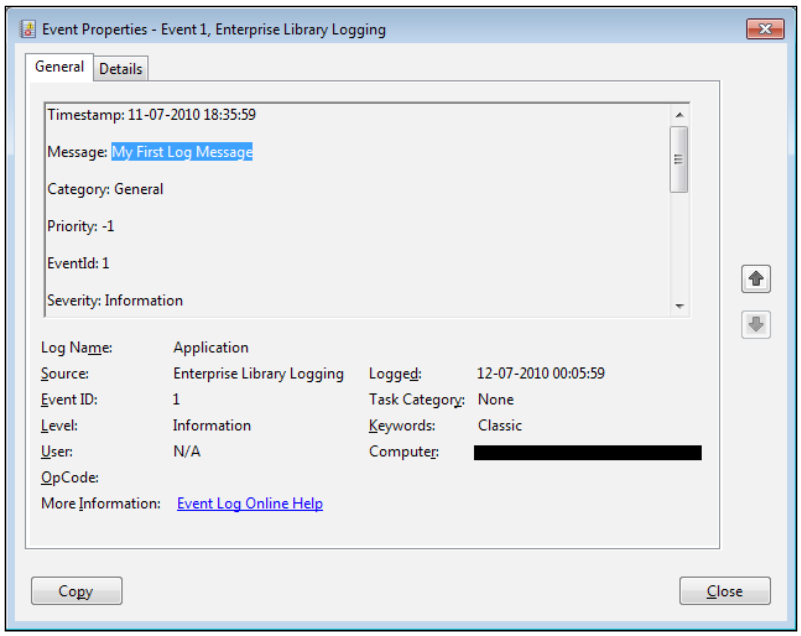
So now we have an instance of `LogWriter` (from this point on we will be using the variable `logWriter` to log the messages), the following code snippet calls the `Write` method of the `LogWriter` instance. The simplest overloaded `Write` method accepts a single parameter of type `System.String` representing the log message.

```
//Writes a new log entry to the default category
logWriter.Write("My First Log Message");
```

In the given code snippet, we are using the simplest overload of the `Write` method of the `LogWriter` class, this method uses the default category and the configured log destination and formatter. Execution of this code will result in creation of a log entry in the Windows Event Log. To view the result, open the Windows Event Viewer and check the log message in the Application section.

By default, the **Enterprise Library Logging** source name is used while writing to the Windows Event Log. Since creation of event sources requires administrator privilege, we will have to run the sample application with administrator privilege for the first time to successfully write the log entry. During deployment, the application installer should take care of creating the appropriate event source.

The following screenshot displays the log message written to the Windows Event Log.



We have successfully written our first log message, in hindsight we have also completed one cycle of the logging process by adding the assembly references, configuring the Logging block settings, adding the namespace, and writing the code to log the message. We will now pick each individual configuration and code elements and learn to understand them in detail.

Exploring design elements

The design of the Logging Application Block involves several elements such as log message, category, logging destination/target and the format in which the information has to be logged. Additionally, we may have filters to enable/disable logging based on certain criteria.

The design elements of the Logging Application Block are as follows:

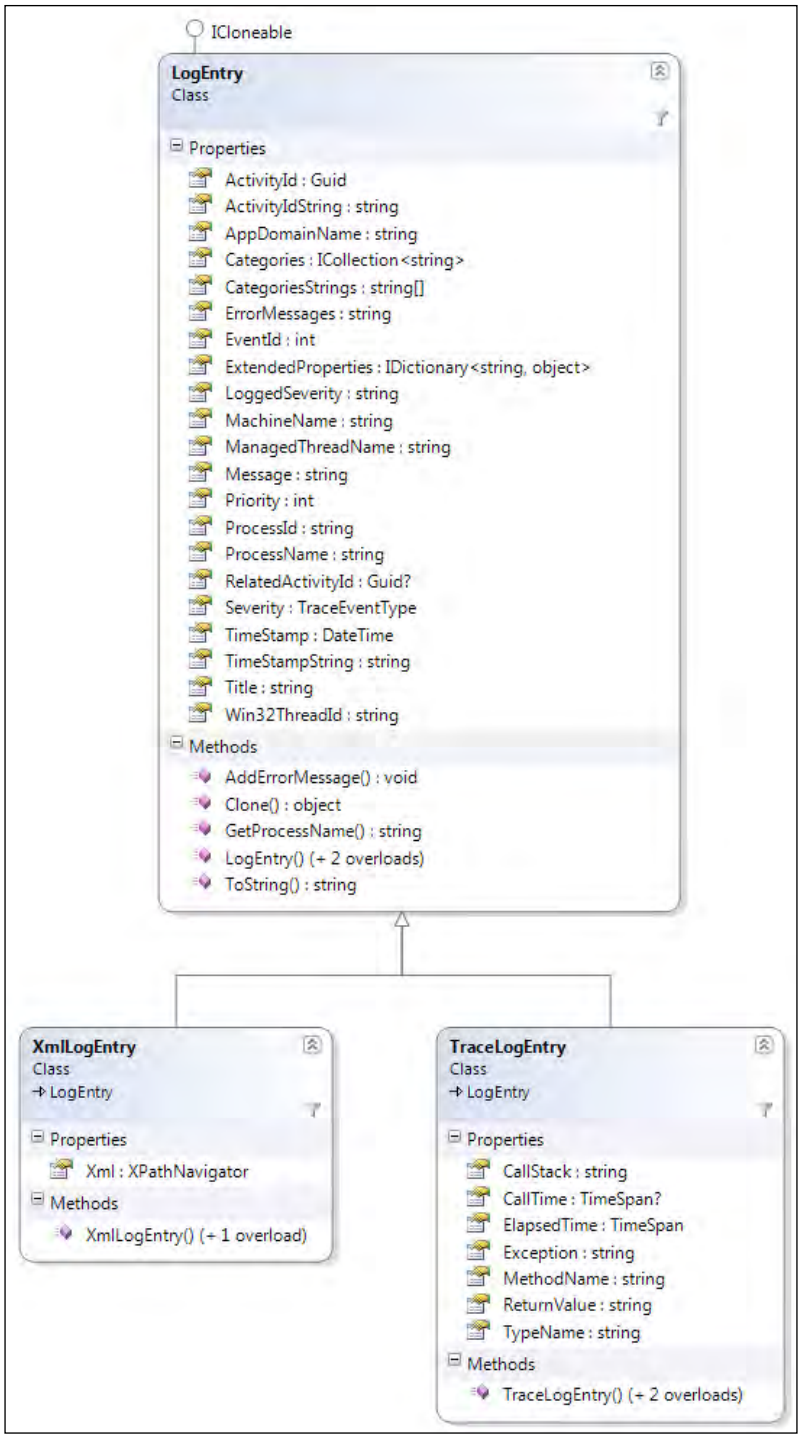
- `LogEntry`
- `Logger`
- `LogWriter`
- Trace Source Categories
- Trace Listeners
- Log Message Formatters
- Logging Filters
- TraceManager and Tracer

We will now explore the technical details of each one of these design elements.

LogEntry

The very basic information for a log entry is the log message; additionally it may have other information such as `Title`, `Priority`, `Categories`, `EventId`, `Severity`, `ActivityId`, `TimeStamp`, and so on. The `LogEntry` class part of the `Microsoft.Practices.EnterpriseLibrary.Logging` namespace holds all this information, which can be passed on to the `Write` method of a `Logger` or `LogWriter` instance. We have not used the `LogEntry` while writing our first log message; the `Write` method provides several overloads, which can be used to pass as little information as we want and simplify the task of logging. However `LogWriter` internally creates a `LogEntry` object with the details provided in the respective overloaded `Write` method.

The following screenshot shows the class diagram of `LogEntry`, `XmlLogEntry`, and `TraceLogEntry`.


ICloneable
LogEntry
Class
Properties
ActivityId : Guid
ActivityIdString : string
AppDomainName : string
Categories : ICollection<string>
CategoriesStrings : string[]
ErrorMessages : string
EventId : int
ExtendedProperties : IDictionary<string, object>
LoggedSeverity : string
MachineName : string
ManagedThreadName : string
Message : string
Priority : int
ProcessId : string
ProcessName : string
RelatedActivityId : Guid?
Severity : TraceEventType
TimeStamp : DateTime
TimeStampString : string
Title : string
Win32ThreadId : string
Methods
AddErrorMessage() : void
Clone() : object
GetProcessName() : string
LogEntry() (+ 2 overloads)
ToString() : string
XmlLogEntry
Class
LogEntry
Properties
Xml : XPathNavigator
Methods
XmlLogEntry() (+ 1 overload)
TraceLogEntry
Class
LogEntry
Properties
CallStack : string
CallTime : TimeSpan?
ElapsedTime : TimeSpan
Exception : string
MethodName : string
ReturnValue : string
TypeName : string
Methods
TraceLogEntry() (+ 2 overloads)

The `LogEntry` class inherits from the `ICloneable` interface and so it supports cloning; the `Clone` method can be called to create a new `LogEntry` object that is a copy of the current instance. The `XmlLogEntry` class inherits from the `LogEntry` class; it provides support to log messages in XML format, and the `XmlTraceListener` class leverages `XmlLogEntry` to deliver the trace data as XML. A `LogEntry` object can be constructed using several different constructors that accept several parameters, apart from which it exposes several public properties that can be assigned. Many of these properties such as `MachineName`, `ProcessId`, `ProcessName`, and so on are initialized internally if the value is not assigned explicitly. A special mention to `ExtendedProperties`, this property allows us to add additional information to the log entry and is quite handy to log custom information.

The following is a list of the properties of the `LogEntry` class for your quick reference. Values of these properties will be part of the generated log entry in the configured destination (file, database, e-mail address, and so on).

Property	Type	Description
`Title`	`String`	Gets or sets the title of the log message; by default this property is set to `String.Empty`.
`Message`	`String`	Gets or sets the message body to log; by default this property returns `String.Empty`.
`Categories`	`ICollection<String>`	Gets or sets the category name as a collection of strings; this information will be used to route the log entry to one or more trace listeners.
`CategoriesStrings`	`String[]`	This read-only property returns categories as a string array; this property is available to support **WMI** queries.
`Priority`	`int`	Gets or sets the priority or importance of the log message; by default it is -1. It is to be noted that only messages satisfying the priority filter configuration of minimum and maximum priorities (inclusive) will be processed.

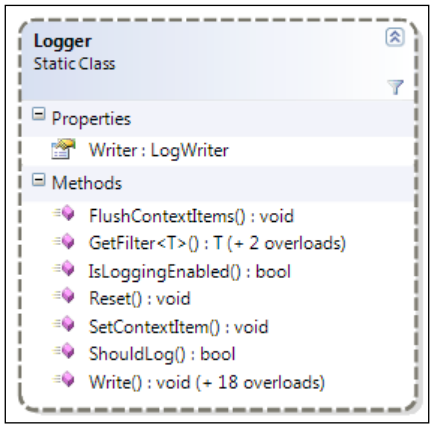
Property	Type	Description
`Severity`	`TraceEventType`	Gets or sets the severity of type `System.Diagnostics.TraceEventType`; the default value is `TraceEventType.Information`.
`LoggedSeverity`	`String`	This read-only property returns the string representation of Severity, which is of type `System.Diagnostics.TraceEventType`.
`EventId`	`int`	Gets or sets the event number or identifier.
`ActivityId`	`Guid`	Holds tracing activity ID; a `Guid` is generated and assigned automatically if tracing is enabled. Returns empty `Guid` if tracing is not enabled.
`ActivityIdString`	`String`	This read-only property returns a string representation of the tracing activity ID to support **WMI** queries.
`RelatedActivityId`	`Guid?`	Gets or sets the related activity ID; by default this property is `null`.
`AppDomainName`	`String`	Gets or sets the `AppDomain` name; if this property is not set then the name of the `AppDomain` in which the program is running will be used.
`MachineName`	`String`	Gets or sets the machine name; if this property is not set then the current name of the machine (`Environment.MachineName`) in which the program is running will be used.
`ManagedThreadName`	`String`	Gets or sets the name of the **.NET** thread; if this property is not set then the current thread name (`Thread.CurrentThread.Name`) will be used.

Property	Type	Description
`ProcessId`	`String`	Gets or sets the Win32 process ID; if this property is not set then the Win32 process ID for the current running process will be used.
`ProcessName`	`String`	Gets or sets the process name; if this property is not set then the process name of the current running process will be used.
`Win32ThreadId`	`String`	Gets or sets the Win32 thread ID; if this property is not initialized then it will automatically return the Win32 thread ID of the current thread provided unmanaged code permission is available.
`TimeStamp`	`DateTime`	Gets or sets the date and time of the log entry message; if this property is not initialized then it will automatically return `DateTime.UtcNow`.
`TimeStampString`	`String`	This read-only property returns the string representation of the `TimeStamp` property formatted using the current culture.
`ErrorMessages`	`String`	This read-only property returns as `String` the error message that was added using the `AddErrorMessage` method.
`ExtendedProperties`	`IDictionary<String, object>`	Gets or sets additional properties through a dictionary of key-value pairs.

Logger

While writing our first log message we leveraged the `LogWriter`. Alternatively we can also use the `Logger` class. The `Logger` class is a static façade to write log enties to one or more logging destinations (trace listeners). This class is part of the `Microsoft.Practices.EnterpriseLibrary.Logging` namespace and was used in the previous versions to perform logging using the exposed methods, primarily the `Write` method.

The following class diagram screenshot depicts the exposed properties and methods of the `Logger` class.



The `Logger` class exposes several methods not only to write log messages but also to perform supporting actions. The following is a list of the methods and brief summary of each method:

Method Name	Description
`Write`	The `Logger` class provides a total of nineteen `Write` methods with variable parameter signatures. These overloaded methods go a long way in logging a meaningful log entry. At the bare minimum it requires only the message to be logged, allowing it to write the log entry to the default category.
`ShouldLog`	This method is useful to query whether a `LogEntry` object should be logged; it accepts a `LogEntry` instance and returns true if the entry should be logged.
`IsLoggingEnabled`	This method queries whether logging is enabled and returns `true/false` based on the outcome of the query.
`GetFilter`	There are three `GetFilter` methods, these methods return the matching `ILogFilter` instance from the filters collection. If no match is found then they return `null`.
`SetContextItem`	This method accepts two parameters: key and value of type `Object`. The added context items will be written with every log entry.
`FlushContextItems`	Calling this method will empty the context items dictionary.
`Reset`	The `Reset` method as per the documentation is marked `public` for testing purposes; it basically resets the writer used by the façade. Please note threads still holding references to the old LogWriter will fail when the LogWriter gets disposed.

Using Logger

Since `Logger` is a static façade class, we can start calling the methods mentioned in the previous table without creating an instance. Internally, it creates a local instance of `LogWriter` using `EnterpriseLibraryContainer.Current.GetInstance<LogWriter>()` and forwards all the actions to the `LogWriter` instance.

The following code snippets demonstrate the usage of different overloads of the `Write` method of the `Logger` class.

Logging using the default category:

```
Logger.Write("Log Message");
```

Logging using a specific category:

```
Logger.Write("Log Message", "LOG CATEGORY");
```

Passing a little more information:

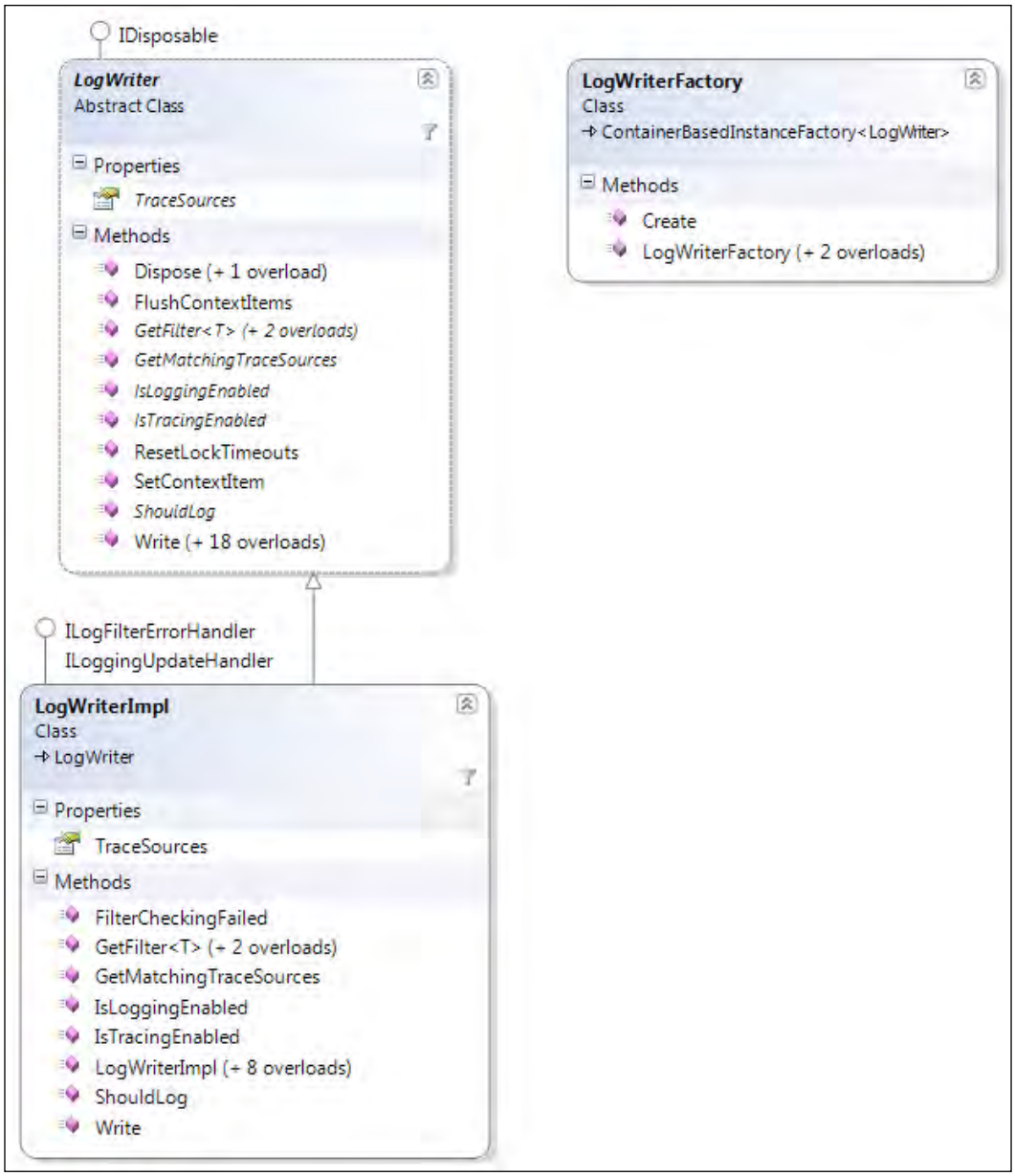
```
//Message | Category | Priority | EventId | Severity | Title
Logger.Write("Log Message", "LOG CATEGORY", 1, 1234, TraceEventType.
Critical, "Log Title");
```

There are in total nineteen overloaded `Write` methods, each accepting a different set of parameters and helping us construct the `LogEntry` object internally using the information provided by us.

LogWriter

We have already explored the usage of `LogWriter` while writing our first log message. Basically, `LogWriter` is an abstract class and is the primary interface in this release for creating log entries; this abstract class belongs to the `Microsoft.Practices.EnterpriseLibrary.Logging` namespace. `EnterpriseLibraryContainer` has the mapping information that resolves the type (`LogWriter`) and creates an instance of `LogWriterImpl`. The `LogWriter` instance can be created using the dependency injection approach or if required, the concrete implementation `LogWriterImpl` can be used directly as well. The `LogWriter` instance writes log messages based on the configuration and the messages are routed to the respective logging destinations (trace listeners) based on category.

The following screenshot shows the methods exposed by `LogWriter`, and inheritance details of `LogWriterImpl` and the `LogWriterFactory` class.


IDisposable
LogWriter
Abstract Class
Properties
TraceSources
Methods
Dispose (+ 1 overload)
FlushContextItems
GetFilter< T> (+ 2 overloads)
GetMatchingTraceSources
IsLoggingEnabled
IsTracingEnabled
ResetLockTimeouts
SetContextItem
ShouldLog
Write (+ 18 overloads)
LogWriterFactory
Class
ContainerBasedInstanceFactory< LogWriter>
Methods
Create
LogWriterFactory (+ 2 overloads)
ILogFilterErrorHandler
ILoggingUpdateHandler
LogWriterImpl
Class
LogWriter
Properties
TraceSources
Methods
FilterCheckingFailed
GetFilter<T> (+ 2 overloads)
GetMatchingTraceSources
IsLoggingEnabled
IsTracingEnabled
LogWriterImpl (+ 8 overloads)
ShouldLog
Write

The `LogWriter` class also exposes several methods and some of them perform the same actions as with the `Logger` class. There are few additional methods that are not available in the `Logger` class. The following is a list of the methods and a brief summary of each method.

Method Name	Description
`Write`	The `LogWriter` class provides a total of nineteen `Write` methods with variable parameter signatures. These overloaded methods go a long way in logging a meaningful log entry. At the bare minimum it requires only the message to be logged, allowing it to write the log entry to the default category.
`ShouldLog`	This method is useful to query whether a `LogEntry` object should be logged; it accepts a `LogEntry` instance and returns true if the entry should be logged.
`IsLoggingEnabled`	This method queries whether logging is enabled and returns `true/false` based on the outcome of the query.
`IsTracingEnabled`	This method queries whether tracing is enabled and returns `true/false` based on the outcome of the query.
`GetFilter`	There are a total of three `GetFilter` methods; these methods return the matching `ILogFilter` instance from the filters collection. If no match is found then they return `null`.
`SetContextItem`	This method accepts two parameters, key and value of type `Object`. The added context items will be written with every log entry.
`FlushContextItems`	Calling this method will empty the context items dictionary.
`GetMatchingTraceSources`	This method returns `IEnumerable<LogSource>` for the matching trace category sources specified in the given `LogEntry` instance.

We used the simplest overload of the `Write` method while logging our first log message. To demonstrate the power of the other overloads, we will explore two more overloaded options. One of the overloads allows us to pass message, title, category, priority, event ID, and severity. This overload allows writing a log entry with the value specified for several key elements.

The following code snippet calls the `Write` method that accepts `Message`, `Category`, `Priority`, `EventId`, `Severity`, and `Title`.

```
//Writes a new log entry with the specified category, priority, event
id, severity and title
logWriter.Write("Log Message", "Log Category", 1, 1234,
TraceEventType.Information, "Log Title");
```

So far we haven't explored the usage of the `LogEntry` class; we can construct a `LogEntry` instance with the values for one or more key elements and pass it to the overloaded `Write` method of the `LogWriter` instance.

The following is a code snippet that constructs a `LogEntry` instance and calls the `Write` method.

```
//Create new LogEntry object
LogEntry logEntry = new LogEntry();

//Assign the category
logEntry.Categories = new string[] { "UI Events" };

//Assign title
logEntry.Title = "Log Title";

//Assign message
logEntry.Message = "Log Message";

//Assign priority
logEntry.Priority = 1;

//Assign severity
logEntry.Severity = TraceEventType.Information;

//Writes a new log entry using the LogEntry instance
logWriter.Write(logEntry);
```

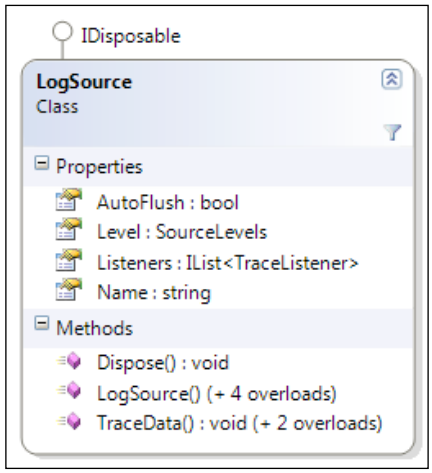
Adding trace source categories

So far we have been using the default category (General) to log messages, which was added automatically while setting up the Logging block settings. Now, let us understand the concept behind **Trace Source Categories** and learn to add and configure new categories. While logging in our application code, we provide one or more category under which the log entry will be logged. Categories allow us to group together a set of log messages. This helps us in controlling the logging behavior such as log destination, log format, and enabling/disabling logging through log filters. These categories can be associated with one or more logging target listeners (log destinations).

We can configure two types of categories:

- Special categories
- Log categories

The following class diagram depicts the exposed properties and methods of the `LogSource` class:



Configuring special categories

Special categories are nothing but out-of-the-box category sources provided by the Logging Application Block. We cannot add additional categories or remove these sources but we can provide one or more log destination (trace listener) to the special category source.

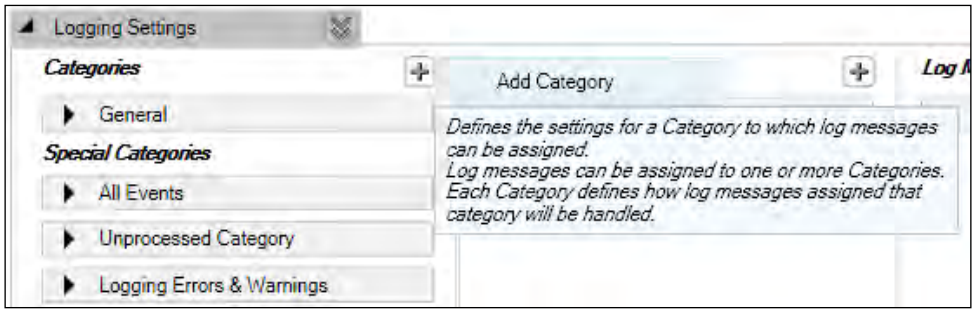
The following table lists the three special categories and their descriptions.

Special Category	Description
All Events	If this special category is configured then regardless of other matching categories, the log entry will be traced through the log source.
Unprocessed Category	If **All Events** special category is not configured and this category is configured and the category specified in the `LogEntry` instance is not defined in the configuration then the log entry will be logged to this special category.
Logging Errors & Warnings	If both **All Event** and **Unprocessed Category** are not configured and the property **Warn If No Category Match** in Logging Settings is set to true then the log entry will be logged to this special category.

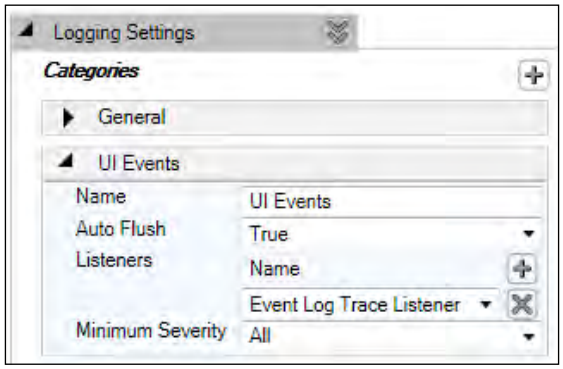
Configuring log categories

Logging a message with defined categories not only gives more context to the message but also allows finer control over it while deciding whether to turn on/off logging for a particular category. It is a good practice to decide the logging categories (for example, Debug, Trace, UI Events, Data Access Events, and so on) beforehand instead of logging into the default category (General). Category sources are defined in the configuration settings as part of the Logging Application Block configuration; a default category is also set while adding the configuration using the Enterprise Library configuration tool.

Let us add a new category in the categories section of the **Logging Settings**. Click on the plus symbol provided in the **Categories** section. Next, click on the **Add Category** menu item, a new category with default values will be loaded in the configuration tool as seen in the following screenshot.



We will add a category named **UI Events**; for the purposes of the demonstration the category name has been updated to **UI Events**. The following screenshot displays the newly added category:



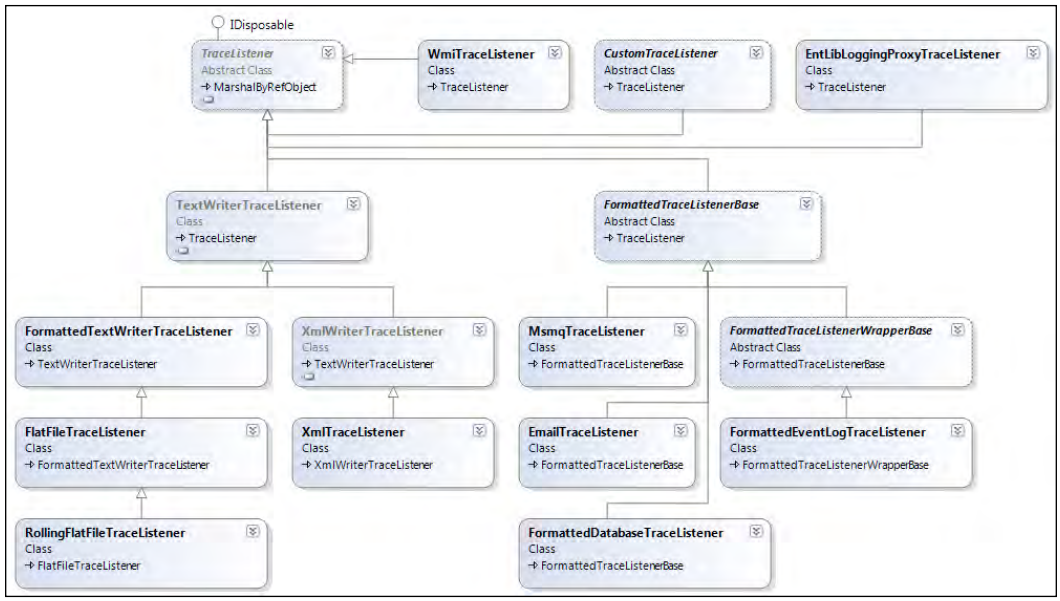
Note that the default **Event Log Trace Listener** is mapped by clicking on the plus symbol against the **Listeners** and selecting the **Event Log Trace Listener** from the drop-down list.

The following is the list of configurable properties and their description:

Property	Description
Name	Category name, used to identify this category. By default this property is set to "**Category**" or if the name already exists then the configuration tool appends the number 2, 3, 4 and so on.
Auto Flush	Indicates whether Logging Target Listeners will automatically flush messages and write the log entries as soon as they are received. Setting it to **False** will buffer the log entries and they will be written in batches or when a significant event occurs such as machine shutting down. By default this property is set to **True**.
Listeners	Allows adding one or more Logging Target Listeners for this category; log entries will be sent to the configured listeners provided they meet the minimum severity. We have to explicitly configure the listeners.
Minimum Severity	Indicates the minimum severity level required to log the message. By default it is set to **All**.

Configuring trace listeners

Trace listeners determine where exactly the log entry will be sent for storage, each trace source category may be associated with one or more trace listeners. Several trace listeners are available out of the box to meet varied requirements; these inherit from the abstract class called `TraceListener`, part of the `System.Diagnostics` namespace. Apart from the trace listeners provided by the Logging Application Block, **.NET Framework** also provides several trace listeners. The Logging block provides additional formatting functionality, which is not available with the **.NET Framework** trace listeners as they only send strings not a `LogEntry` object. The following class diagram shows the base classes through which several trace listeners are derived and additional functionality is implemented.



Trace listeners such as `FormattedEventLogTraceListener`, the `FlatFileTraceListener`, and the `WMITraceListener` use the same configuration information as `System.Diagnostics` trace listeners. This means we can leverage these three trace listeners provided as part of the `<system.diagnostics>` configuration section. Several trace listeners have common properties that can be configured; these properties are explained next:

Property Name	Description
Name	Used to identify an item.
Severity Filter	This setting determines the minimum severity of message that will be sent to the logging target. Below are the options for this setting; the default setting is **All**. • **All** • **Off** • **Critical** • **Error** • **Warning** • **Information** • **Verbose** • **ActivityTracing**

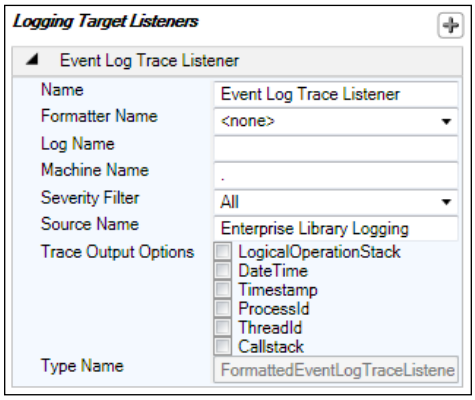
Property Name	Description
Trace Output Options	Trace listeners that do not output to a Text Formatter use this setting to determine the elements/options to be included in the trace output. Below are the possible values; by default none of the values is included. • **LogicalOperationStack** • **DateTime** • **Timestamp** • **CallStack** • **ProcessId** • **ThreadId**

Configuring Event Log Trace Listener

Although we have already seen this trace listener in action as it is part of the default configuration, which was used while writing our first log message, we haven't yet explored the design elements and the available configuration options. Logging formatted messages to the Windows Event Log is provided by the `FormattedEventLogTraceListener` class and is part of the `Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners` namespace. This class inherits from the abstract class `FormattedTraceListenerWrapperBase`. The `FormattedEventLogTraceListener` class internally creates an instance of `System.Diagnostics.EventLogTraceListener` and passes on to its base class.

The Logging block creates the event log source if it does not exist; since creation of event log source requires appropriate privileges (access rights to the registry key `HKLM\System\CurrentControlSet\Services\EventLog`) the application/component must run with those privileges. Alternatively, the event log source can be created while installing the application/component under an account with the required privileges.

The following screenshot depicts the default settings without any association to the log formatter.



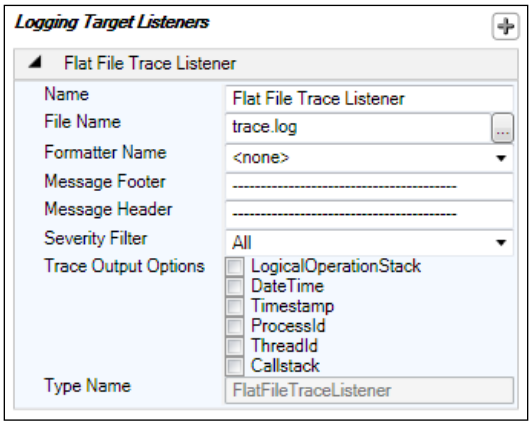
The following table provides a listing of all the configurable properties and their description. It will help in modifying the default behavior of the Formatted Event Log Trace Listener.

Property	Description
Name	Logging target listener name used to identify this item.
Formatter Name	Name of the log message formatter; the drop-down list allows selecting the currently added log message formatters.
Log Name	Indicates the name of the Windows Event Log such as **Application** or **System** to which the log messages will be written.
Machine Name	Name of the machine to which the log messages should be written; the default value is "**.**" denoting the local machine.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Source Name	Source name to be used while writing to the Windows Event Log; the default value is **Enterprise Library Logging**.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.

Configuring Flat File Trace Listener

This trace listener writes log entries to a flat file using the configured log formatter. The `FlatFileTraceListener` class is part of the `Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners` namespace and inherits from the `FormattedTextWriterTraceListener` class.

Flat File Trace Listener allows to output log messages to a disk file. The following is a screenshot of the default configuration setting:



The following table lists the configuration properties and their description:

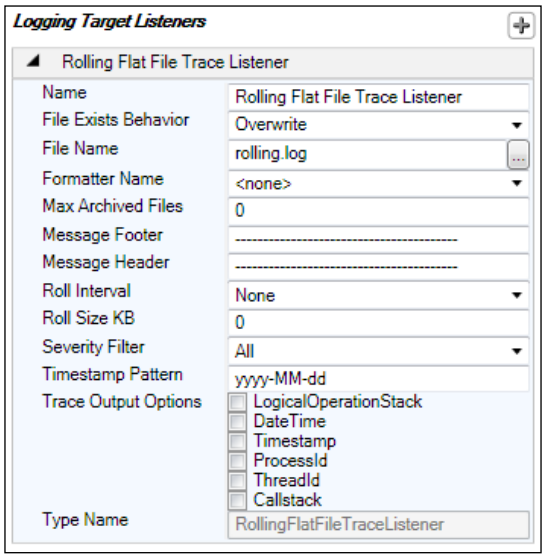
Property	Description
Name	Logging target listener name used to identify this item.
File Name	Path and file name for the log file, using environment variables such as `%TEMP%`, `%WINDIR%`, etc.
Formatter Name	Name of the log message formatter, the drop-down list allows selecting the currently added log message formatters.
Message Footer	Footer text to be added to the log message.
Message Header	Header text to be added to the log message.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.

While running the application in debug mode, the log file will be generated in `%Program Files%\Microsoft Visual Studio 9.0\Common7\IDE` folder, the "File Name" can be changed to "`.\trace.log`" to generate the log file in the executing assembly folder.

Configuring Rolling Flat File Trace Listener

While logging to a flat file is a good option, sometimes we might want to log to new file based on the size or age of the file. The Rolling Flat File Trace Listener provides this functionality by allowing us to configure the size and time thresholds. The `RollingFlatFileTraceListener` class is part of the `Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners` namespace and it inherits from the `FlatFileTraceListener` class. The Rolling Flat File Trace Listener provides several properties to tweak the behavior of the listener through configuration.

The following screenshot displays the default configuration settings.



The list of properties and their description is given next. These properties can be configured to tweak the behavior of the Rolling Flat File Trace Listener.

Property	Description
Name	Logging target listener name used to identify this item.
File Exists Behavior	Determines whether to overwrite the file or create a new file using a name created using the timestamp when it rolls over.
File Name	Path and file name for the log file, using environment variables such as `%TEMP%`, `%WINDIR%`, etc.
Formatter Name	Name of the log message formatter; the drop-down list allows selecting the currently added log message formatters.

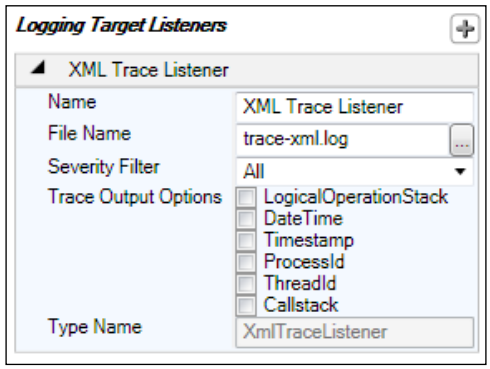
Property	Description
Max Archived Files	This property specifies the maximum number of log files to be retained; when the number of log files exceeds the specified number the listener will purge the old files based on the file creation date.
Message Footer	Footer text to be added to the log message.
Message Header	Header text to be added to the log message.
Roll Interval	Determines the log file roll-over interval; the default value is **None**. Options include: • **Minute** • **Hour** • **Day** • **Week** • **Month** • **Year** • **Midnight**
Roll Size KB	Determines the maximum log file size (in kilobytes) before rolling over.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Timestamp Pattern	Specifies the date/time format to be used to suffix the file name.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.

Configuring XML Trace Listener

This trace listener as the name suggests writes the log message to a file in XML form. The `XmlTraceListener` class is part of the `Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners` namespace and it inherits from the `XmlWriterTraceListener` class available in the `System.Diagnostics` namespace. It does not require a log formatter as it internally formats `LogEntry` or any class derived from `LogEntry` into an XML string using the `XmlLogFormatter` class.

XML Trace Listener configuration consists of three key properties: `File Name`, `Severity Filter`, and `Trace Output Options`, which might be modified to change the respective behavior.

The following screenshot shows the default settings of the XML Trace Listener:



The following table listing shows the configurable properties and their descriptions.

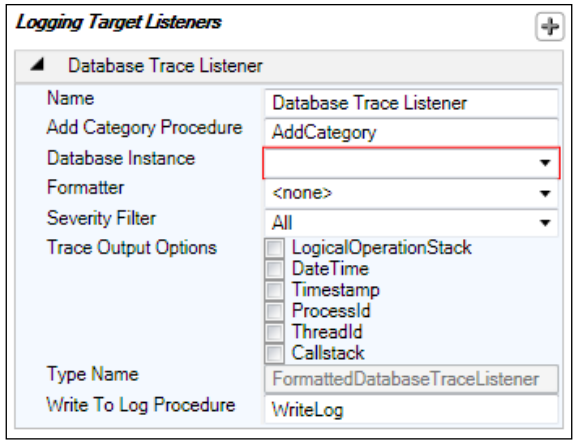
Property	Description
Name	Logging target listener name used to identify this item.
File Name	Path and file name for the log file, using environment variables such as `%TEMP%`, `%WINDIR%`, etc.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.

Configuring Database Trace Listener

As the name suggests, this trace listener writes log messages to a database formatting the output using the configured log formatter. The `FormattedDatabaseTraceListener` class is part of the `Microsoft.Practices.EnterpriseLibrary.Logging.Database` namespace and it inherits from an abstract class named `FormattedTraceListenerBase`. The Logging block provides the database table schema and stored procedures to log messages in the database; the database script `LoggingDatabase.sql` and Windows command script `CreateLoggingDb.cmd` are available in the default source folder (`EntLib50Src\Blocks\Logging\Src\DatabaseTraceListener\Scripts`). Although the SQL script generates a database named **Logging**, the script can be modified to create tables and stored procedures in our custom database as well. By default the command file generates the database, tables, and stored procedures in local instance of SQL Server Express; this can be customized in the command script file.

Database Trace Listener configuration involves setting of **Database Instance** and **Formatter** at the bare minimum; other properties might be modified to change their respective behavior.

The next screenshot shows the default settings of the Database Trace Listener:



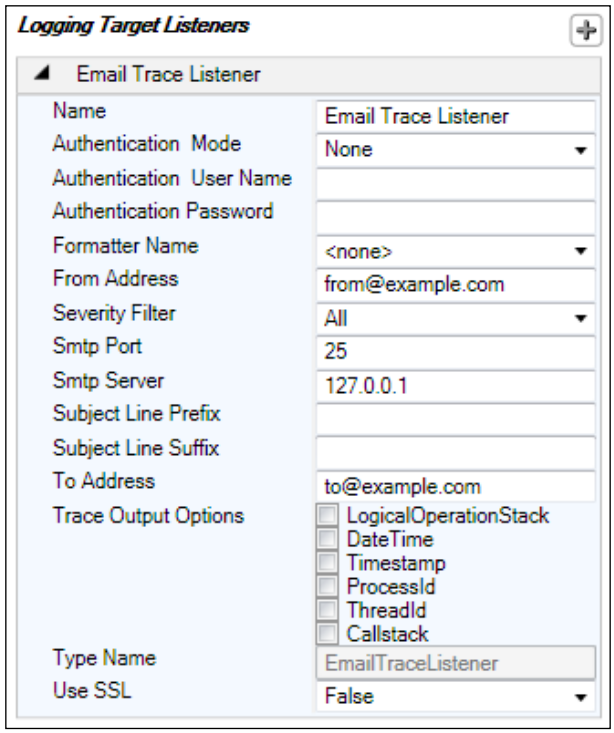
The following table listing shows the configurable properties and their description:

Property	Description
Name	Logging target listener name used to identify this item.
Add Category Procedure	Name of the stored procedure that creates a new category in the tables; the default value is **AddCategory**, which is generated by the script provided in the source folder of database logging.
Database Instance	Name of the database instance to be used for logging messages.
Formatter Name	Name of the log message formatter; the drop-down list allows selecting the currently added log message formatters.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.
Write To Log Procedure	Name of the stored procedure that inserts log messages into the tables; the default value is **WriteLog**, which is generated by the script provided in the source folder of database logging.

Configuring to send log messages to an e-mail address

The e-mail trace listener provides the ability to send log entries as e-mail messages to the specified e-mail address. This trace listener is feature-packed; it allows setting the authentication mode, from address, SMTP port and server, SSL mode, and so on. The `EmailTraceListener` class provides the implementation for this functionality; it inherits from the `FormattedTraceListenerBase` abstract class and both are part of the `Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners` namespace. Email Trace Listener configuration involves setting of several properties that are mandatory for the functioning of this trace listener.

The following screenshot shows the default setting of the Email Trace Listener:

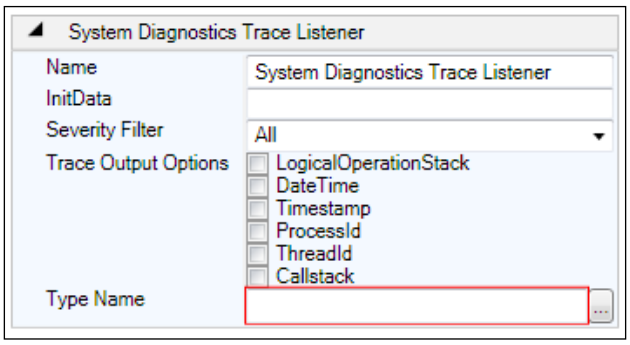


The following table listing shows the configurable properties and their description:

Property	Description
Name	Logging target listener name used to identify this item.
Authentication Mode	Determines how the listener will authenticate the user. The default value is **None**. Options include: • **None** • **WindowsCredentials** • **UserNameAndPassword**
Authentication User Name	User name to use for authentication while sending e-mail messages.
Authentication Password	Password to use for authentication for the specified user name.
Formatter Name	Name of the log message formatter; the drop-down list allows selecting the currently added log message formatters.
From Address	E-mail address to be used to send the e-mail messages from.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Smtp Port	Specifies the **SMTP port** to be used to send the e-mail message; the default value is **25**.
Smtp Server	Specifies the **SMTP server** name or IP address to be used to send the e-mail message; the default IP address is **127.0.0.1** (local host).
Subject Line Prefix	Prefix to add at the start of the e-mail subject.
Subject Line Suffix	Suffix to add to the end of the e-mail subject.
To Address	The address to send the log entry e-mail to.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.
Use SSL	Determines whether to use Secure Socket Layer (SSL).

Configuring System Diagnostics Trace Listener

The System Diagnostics Trace Listener is an interesting trace listener; it provides the **Type Name** property to configure the trace listener to be used from the list of available trace listeners. The following screenshot displays the default settings of this trace listener.

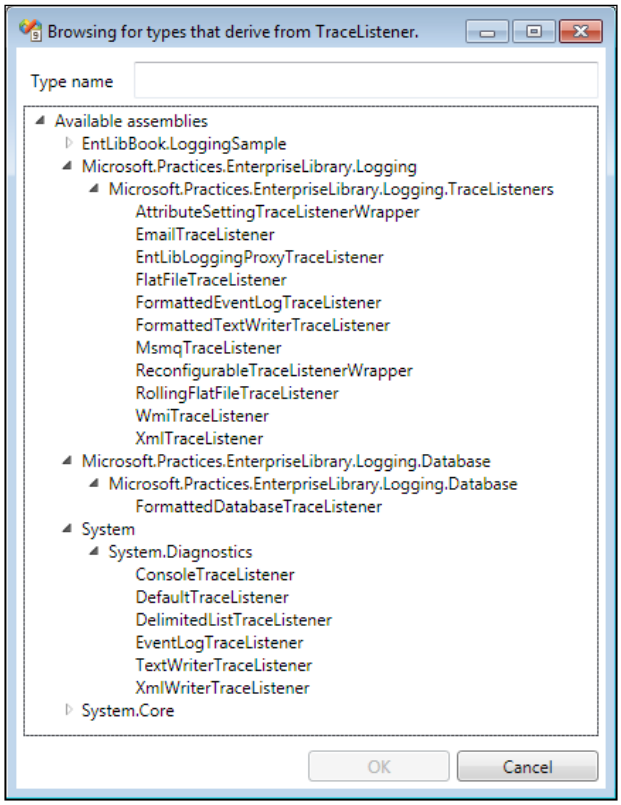


Apart from the common properties, the previous screenshot has two interesting properties. The **Type Name** property allows us to assign the fully qualified type name of the trace listener to be used while writing log messages. The **InitData** property allows us to pass initialization data to the configured trace listener.

The following table listing shows the configurable properties and their description:

Property	Description
Name	Logging target listener name used to identify this item.
InitData	The value provided in this property will be passed on to the configured trace listener as initialization data.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.
Type Name	Fully qualified type name of the trace listener to be used to write log messages.

The following screenshot shows the type selection dialog box with the list of available trace listeners derived from `TraceListener`:

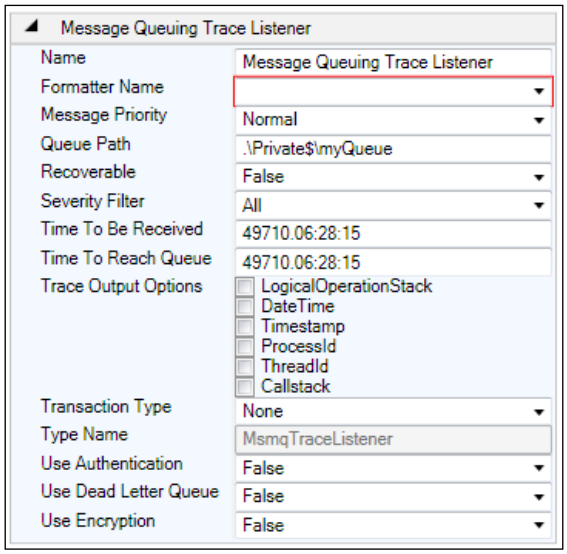


We can select any trace listener from the list as shown in the above screenshot, for example selecting `ConsoleTraceListener` will write all messages to `Console.Out` or `Console.Error`.

Configuring Message Queuing Trace Listener

The Message Queuing Trace Listener sends the log entries to the configured MSMQ instance; the `MsmqTraceListener` class inheriting the `FormattedTraceListenerBase` abstract class, both part of the `Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners` namespace, provides the implementation for this functionality.

The following screenshot displays the default settings of Message Queuing Trace Listener.



The following table provides the list of configurable properties and their description.

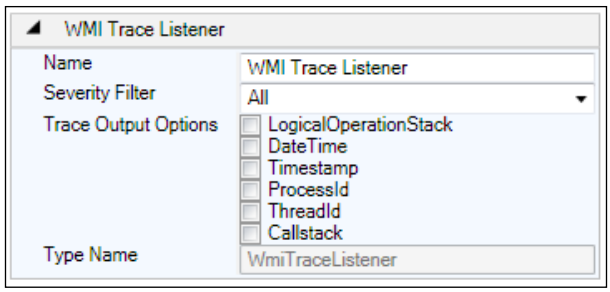
Property	Description
Name	Logging target listener name used to identify this item.
Formatter Name	Name of the log message formatter; the drop-down list allows selecting the currently added log message formatters.
Message Priority	This property sets the priority of a log entry; while in transit the message priority determines where the log entry is inserted into its destination queue. The default value is **Normal**. Available options are: • **Lowest** • **VeryLow** • **Low** • **Normal** • **AboveNormal** • **High** • **VeryHigh** • **Highest**

Property	Description
Queue Path	Message queuing path to be used by the MSMQ Trace Listener instance. The default value is **.\Private$\myQueue**.
Recoverable	This property determines whether the log entry is delivered even following computer failure or network problem. The default value is **False**. Available options: • **True** • **False**
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Time To Be Received	This property allows setting the total time to receive the log entry by the destination queue. The default value is **49710.06:28:15**.
Time To Reach Queue	This property allows setting the maximum time to reach the queue for a log entry. The default value is **49710.06:28:15**.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.
Transaction Type	This property determines the Message Queuing transaction type. The default value is **None**. Available options: • **None** • **Automatic** • **Single**
Use Authentication	This property determines whether to use authentication before the message is sent. The default value is **False**.
Use Dead Letter Queue	This property determines whether a copy of any undelivered message should be sent to dead letter queue. The default value is **False**.
Use Encryption	This property determines whether to use encryption. The default value is **False**.

Configuring WMI Trace Listener

The WMI Trace Listener raises a WMI event passing the `LogEntry` instance; this functionality is implemented by the `WmiTraceListener` class inheriting directly from the `System.Diagnostics.TraceListener` abstract class.

The following screenshot displays the default settings of **WMI Trace Listener**:



The following table provides a list of the configurable properties and their description:

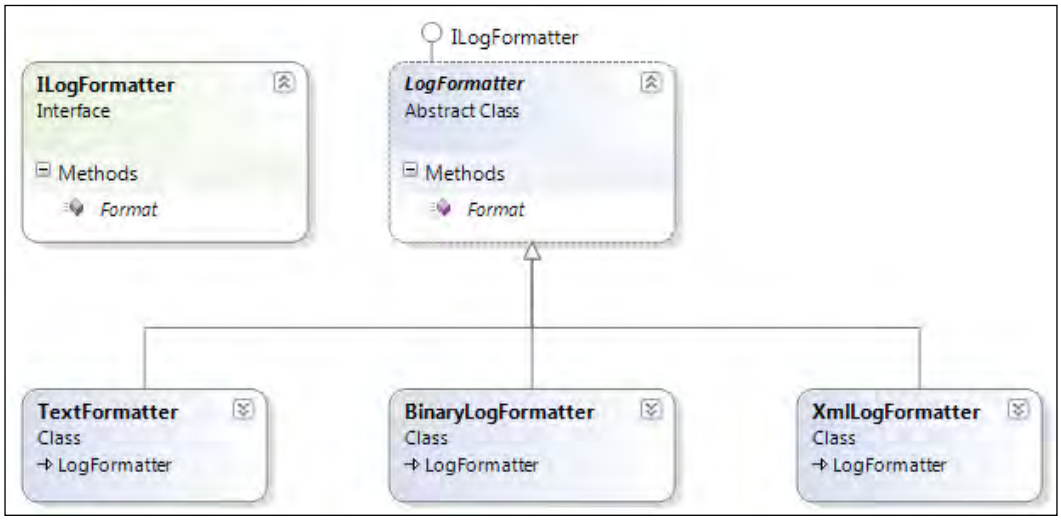
Property	Description
Name	Logging target listener name used to identify this item.
Severity Filter	Indicates the minimum severity of messages that should be processed and sent to the logging target.
Trace Output Options	Determines the elements included in the trace output for listeners that do not output to a Text Formatter. The default value is **None** and this property is optional.

Configuring custom trace listeners

The Logging block provides an abstract class called `CustomTraceListener` as an extension point for implementing custom trace listeners. Also, we may extend one of the existing trace listener implementations to satisfy our unique requirements.

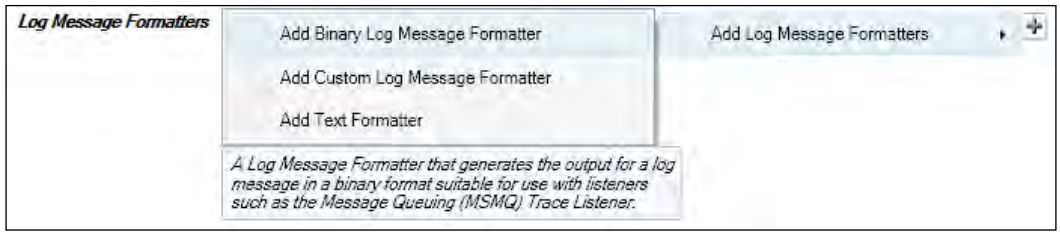
Configuring log message formatters

While logging information to a log destination, the log entry must often be formatted. The Logging block provides two log message formatters (`TextFormatter` and `BinaryLogFormatter`) to format the information in the `LogEntry` instance. Both these formatters inherit from an abstract class named `LogFormatter`, which in turn implements the `ILogFormatter` interface. All the mentioned formatter elements are part of the `Microsoft.Practices.EnterpriseLibrary.Logging.Formatters` namespace. `ILogFormatter` exposes a method called `Format` that accepts a `LogEntry` instance and returns the formatted string; derived classes are expected to provide implementation for the `Format` method.

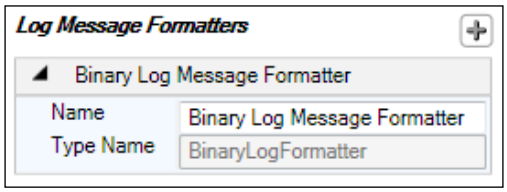


- `TextFormatter`: This is a template-based formatter that formats `LogEntry` information using the default template tokens.
- `BinaryLogFormatter`: This serializes a `LogEntry` object using `BinaryFormatter` and returns it as a base-64 encoded string. This formatter should be used with Message Queuing.
- `XmlLogFormatter`: As the name suggests, this formatter formats a `LogEntry` object to an XML string representation. This formatter is not available as part of configuration but is internally used by `XmlTraceListener` to convert the `LogEntry` object to an XML string.

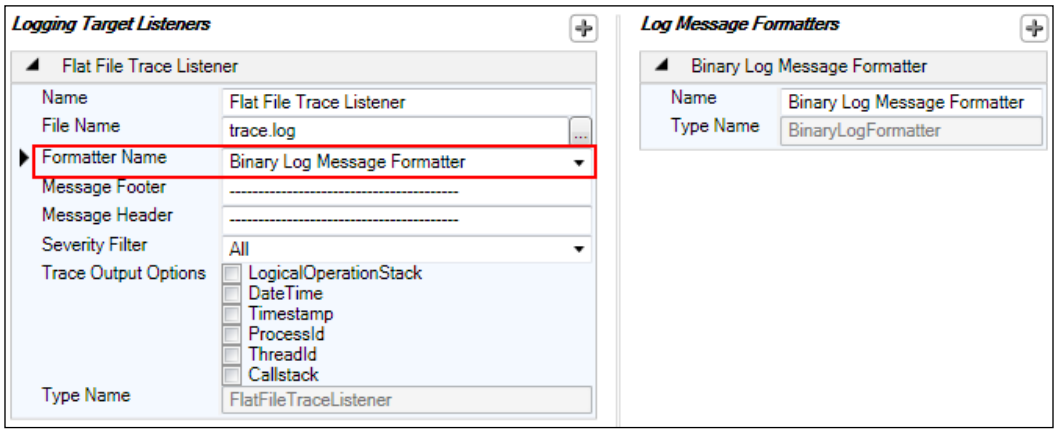
Let's see how to configure a trace listener to use Binary Log Message Formatter; the process is same for Text Formatter as well. Click on the plus symbol in the **Log Message Formatters** section and navigate to **Add Log Message Formatters** | **Add Binary Log Message Formatter** as shown in the following screenshot.



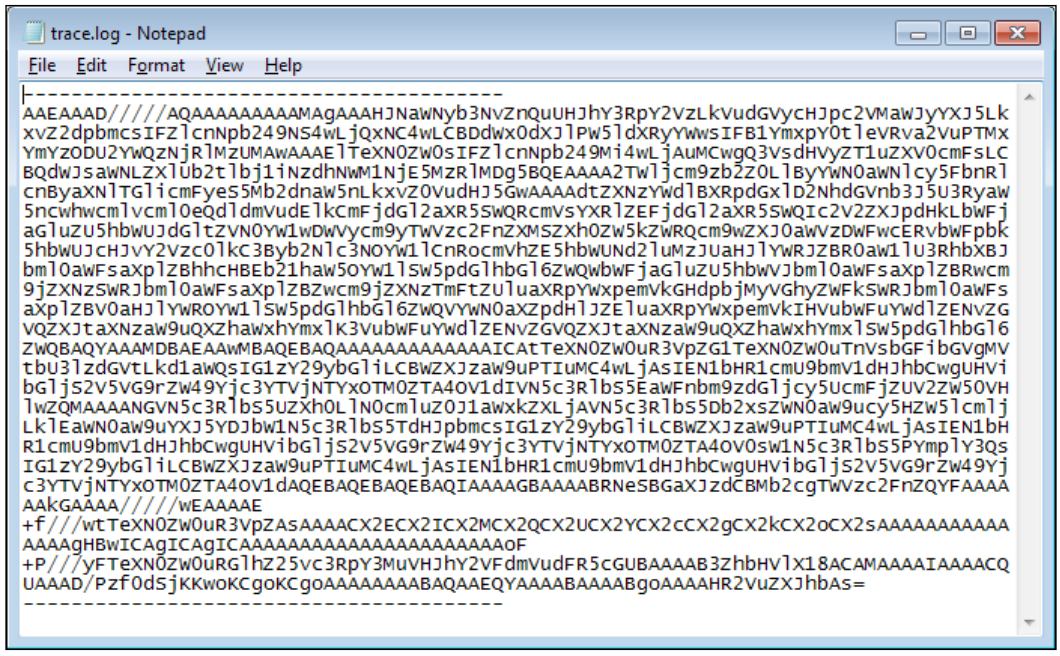
The action performed in the above screenshot will add a **Binary Log Message Formatter** to the configuration editor as shown in the following screenshot.



Once the log message formatter is added to the configuration, the next step is to configure the formatter in the trace listener. Most trace listeners have a property named **Formatter Name**, which lists the available log message formatters in a drop-down list. The following screenshot shows the **Formatter Name** configured to use the **Binary Log Message Formatter**.

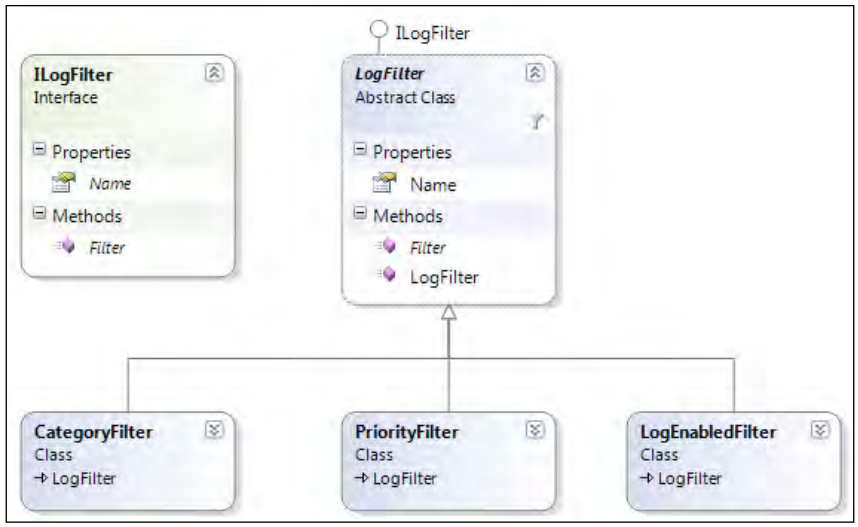


While writing log messages using the preceding configuration, the log messages will be formatted using the binary format. The following screenshot shows output from the `trace.log` file with the log message formatted using the **Binary Log Message Formatter**.



Configuring logging filters

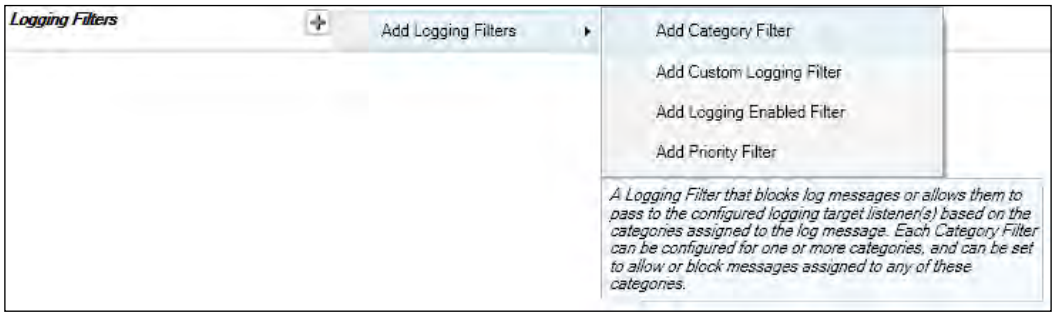
Logging is very helpful but it also comes with a cost; too much logging might impact performance. Also there are scenarios where we might want to disable logging based on certain conditions. Logging filters provide the mechanism to switch on/off logging. We can provide filter conditions and prevent the Logging block from sending the `LogEntry` object to the trace listeners. The Logging block provides three types of logging filters: `CategoryFilter`, `PriorityFilter`, and `LogEnabledFilter`. These filters inherit from an abstract class named `LogFilter` which in turn implements the `ILogFilter` interface. All these logging filter elements are part of the `Microsoft.Practices.EnterpriseLibrary.Logging.Filters` namespace. The `ILogFilter` interface exposes two members; derived classes are expected to provide implementation for both the members. The `Name` property returns the name of the log filter and the `Filter` method accepts a `LogEntry` object and returns a `Boolean` value indicating whether or not to send the message to the trace listeners.



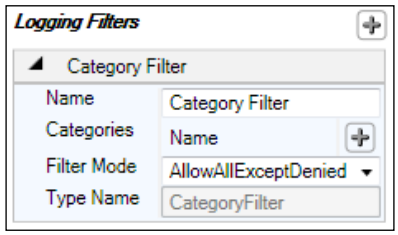
- `CategoryFilter`: Filters `LogEntry` objects based on categories, this allows us to turn on/off logging for specific categories.
- `PriorityFilter`: Filters `LogEntry` objects based on the priority, we can specify the minimum and maximum priority condition for logging.
- `LogEnabledFilter`: This filter gives us control to completely turn on/off logging.

Adding a category filter

Category filter configuration allows us to add one or more categories and set the filter mode to either "allow all except denied" or "deny all except allowed". The following screenshot shows how to add a category filter:



The following screenshot shows the default settings of the newly added category filter:



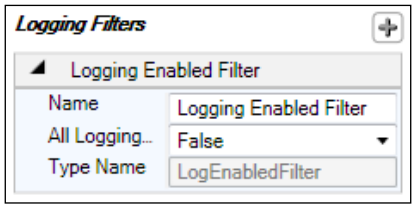
The following table shows the list of configurable properties and their description:

Property	Description
Name	Logging filter name used to identify this item.
Categories	List of all the categories defined for this filter.
Filter Mode	Filter mode determines whether the configured categories will be allowed or denied logging. The default value is **AllowAllExceptDenied**. Options are: • **AllowAllExceptDenied** • **DenyAllExceptAllowed**

Adding a logging enabled filter

Logging enabled filter configuration is pretty straight forward, it just allows us to specify whether all logging activities are enabled or disabled by setting the **All Logging Enabled** property. The following screenshot shows how to add the logging enabled filter:

The following screenshot shows the default settings of the newly added **Logging Enabled Filter**:

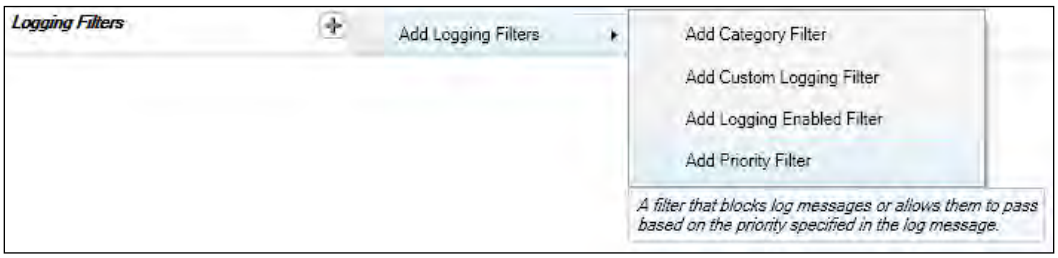


The following table listing shows the available configurable properties and their description:

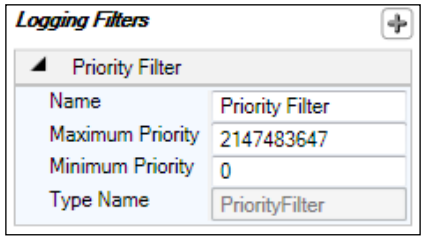
Property	Description
Name	Logging filter name used to identify this item.
All Logging Enabled	Determines whether all logging is enabled or disabled. The default value is **False**.

Adding a priority filter

Priority filter configuration allows us to configure the maximum and minimum priority values based on which the log messages will be filtered. The following screenshot shows how to add a priority logging filter:



The following screenshot shows the default settings of the newly added **Priority Filter**:



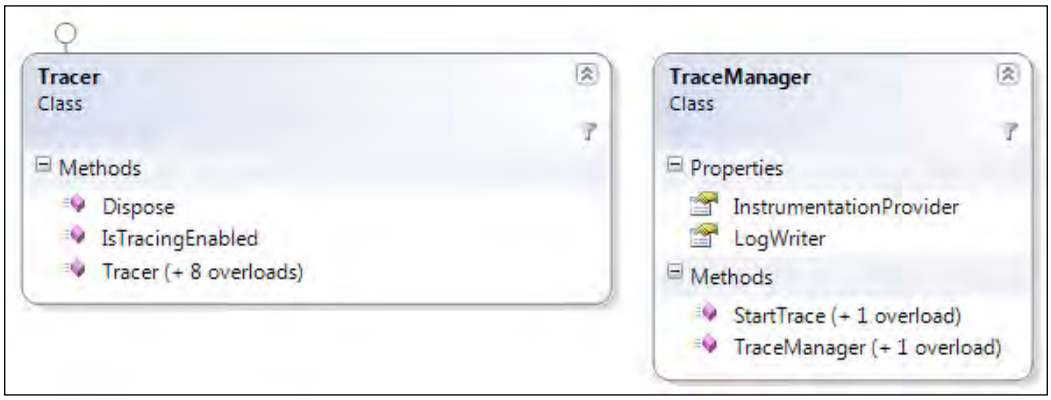
The table below shows the list of configurable properties and their description:

Property	Description
Name	Logging filter name used to identify this item.
Maximum Priority	Maximum priority filter value; any log message priority greater than this value will not be logged. The default value is **2147483647**.
Minimum Priority	Minimum priority filter value; any log message priority less than this value will not be logged. The default value is **0**.

TraceManager and Tracer

The `TraceManager` class provides application activity tracing functionality to log method entry/exit and duration; it is part of the `Microsoft.Practices.EnterpriseLibrary.Logging` namespace. This class exposes a method named `StartTrace`; this method internally creates and returns a new `Tracer` object. The `Tracer` class provides the actual implementation of the tracing functionality. Tracing starts with the creation of the `Tracer` object and ends when the object is disposed.

The following screenshot displays the class diagram of the `Tracer` and `TraceManager` classes:



Tracing activities

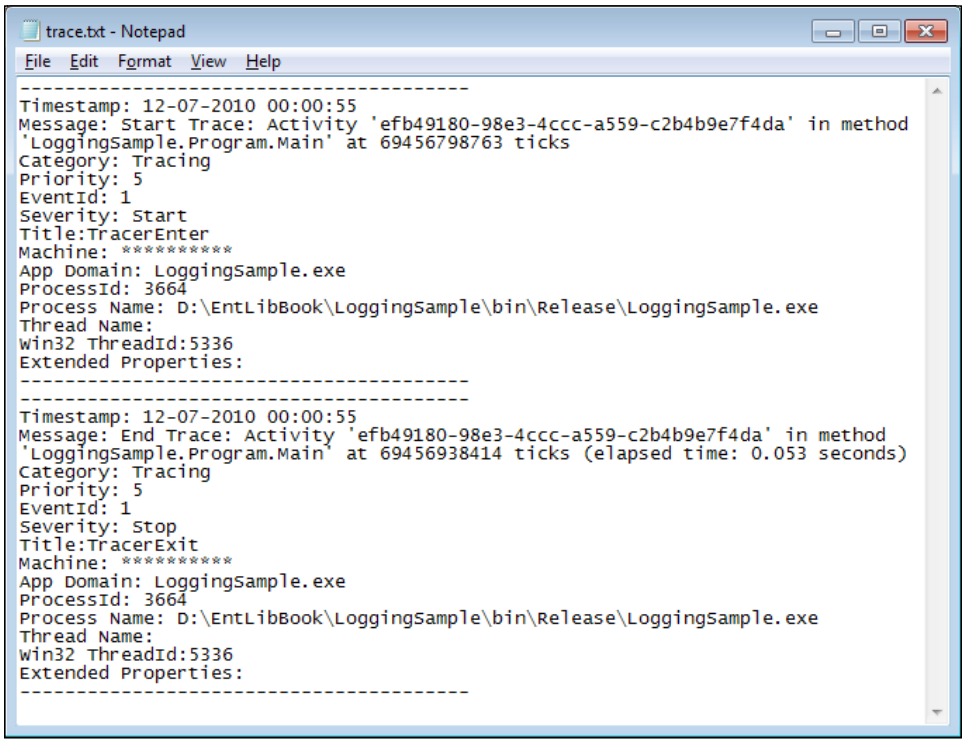
We can trace application activities using the `TraceManager` class. This class exposes a method called `StartTrace`, which starts with the invocation of the `StartTrace` method and stops the tracing activity when the `Tracer` instance gets disposed.

The following code snippet shows how to initiate tracing and end the tracing activity:

```
//Create a TraceManager instance using the EnterpriseLibraryContainer
TraceManager traceManager = EnterpriseLibraryContainer.Current.
GetInstance<TraceManager>();

using (traceManager.StartTrace("Tracing"))
{
    //Perform application actions here
}
```

The previous code snippet first creates an instance of `TraceManager` using the `EnterpriseLibraryContainer` class. Next, we use the `TraceManager` instance and call the `StartTrace` method inside a `using` statement. This makes sure that the `Tracer` instance created internally gets disposed and the tracing activity ends with the disposal of the `Tracer` instance.

The following screenshot shows the typical log entry of a tracing activity:



The above given screenshot shows two log entries representing the start and end of the tracing activity with the timestamp, activity ID, ticks, and other details.

Customizing Logging block elements

The Logging block provides extension points to implement custom Trace Listeners, Log Formatters, and Log Filters. Although the Logging block provides commonly used logging functionality every project has its own set of requirements and on some occasions we would like to extend an existing element or write a custom implementation using the extension points.

Implementing a custom trace listener

Implementing a custom trace listener is simple; we just need to inherit from the abstract class called `CustomTraceListener`. The `CustomTraceListener` class inherits from `System.Diagnostics.TraceListener`, also an abstract class exposing several virtual methods that can be overridden to provide the custom implementation. We have to add reference to the `System.Configuration.dll` assembly in the project while implementing custom trace listener.

The following code snippet shows the list of required namespaces for the custom implementation:

```
using System.Diagnostics;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners;
```

The folowing code block shows the implementation details such as `ConfigurationElementType` attribute, inheritance, methods to be overridden, and so on.

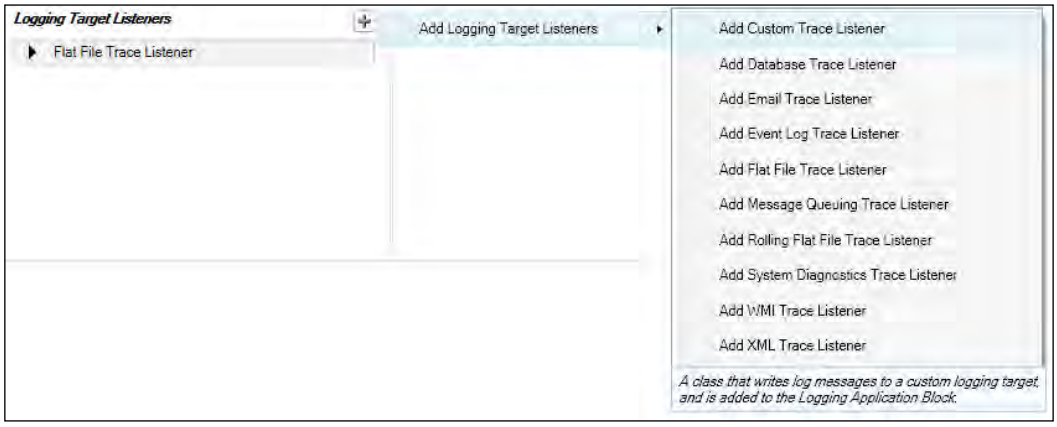
```
[ConfigurationElementType(typeof(CustomTraceListenerData))]
public class MyCustomTraceListener : CustomTraceListener
{
    public override void TraceData(TraceEventCache eventCache, string
source, TraceEventType eventType, int id, object data)
    {
        if (data is LogEntry && this.Formatter != null)
        {
            this.WriteLine(this.Formatter.Format(data as LogEntry));
        }
        else
        {
            this.WriteLine(data.ToString());
        }
    }
```

```
        public override void Write(string message)
        {
            this.WriteLine(message);
        }

        public override void WriteLine(string message)
        {
            //Write to custom destination
        }
    }
```

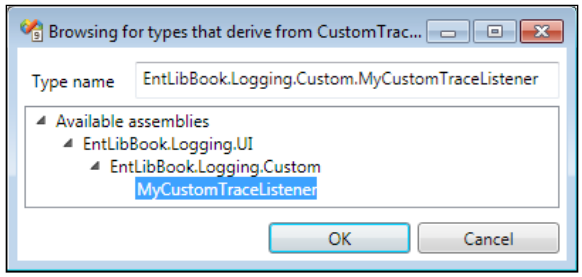
The `MyCustomTraceListener` class inherits from `CustomTraceListener` and has overridden three methods: `TraceData`, `WriteLine`, and `Write`. We have to provide our custom logic to write the messages to the destination in the `Write` and `WriteLine` methods. It is to be noted that in the `TraceData` method we verify whether the parameter data is of type `LogEntry`; this check is carried out to ensure that the custom trace listener executes correctly outside of the Logging block. We also verify whether we have the formatter to format the log message; based on the outcome of the condition we write the message by passing the message to the `WriteLine` method. Also to be noticed is that the `MyCustomTraceListener` class is decorated with the `ConfigurationElementType` attribute with the input as `CustomTraceListenerData`; this attribute indicates the configuration object type to be used.

Configuring the custom trace listener

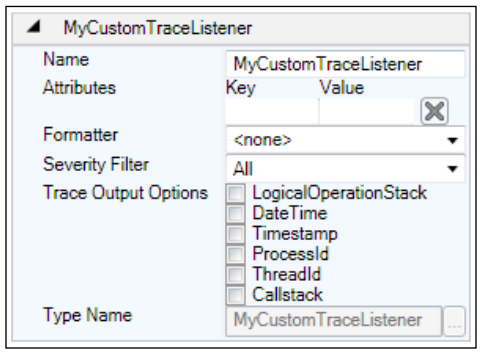
We have to configure the custom trace listener to leverage the trace listener; configuration is similar to what we have seen with other trace listeners. The following screenshot shows how to add a custom trace listener:



After clicking on **Add Custom Trace Listener**, a dialog box is displayed with the available types that derive from `CustomTraceListener`.



After selecting the required type and clicking on the **OK** button, the configuration editor will add the details to the configuration file. The following screenshot displays the added **MyCustomTraceListener**:



We are already aware of the common properties such as **Name**, **Formatter**, **Severity Filter**, and **Trace Output Options**. The property named **Attributes** comes in quite handy to pass additional configuration information to our custom trace listener.

Implementing a custom log formatter

A custom log formatter can be implemented by implementing the `ILogFormatter` interface. We need to implement the `Format` method that accepts a `LogEntry` instance and provides custom formatting logic to return the formatted string.

The following code snippet shows the required namespaces:

```
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging.Formatters;
```

The following code snippet shows the implementation structure of a custom log formatter.

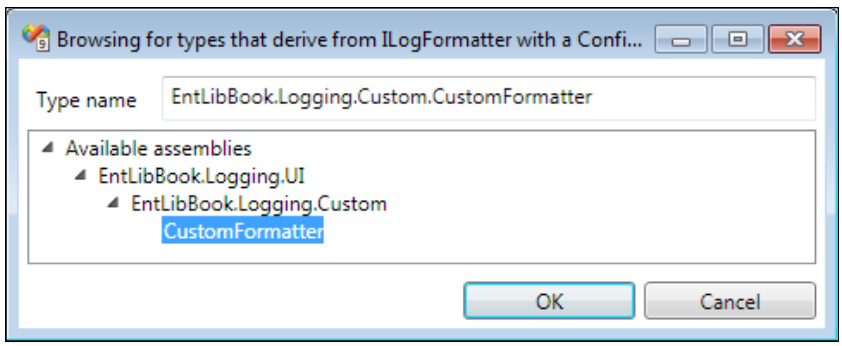
```
[ConfigurationElementType(typeof(CustomFormatterData))]
public class CustomFormatter: ILogFormatter
{
    public string Format(LogEntry log)
    {
        //Provide custom formatting logic here
    }
}
```

We have to provide the custom formatting logic in the `Format` method of the `CustomFormatter` class.

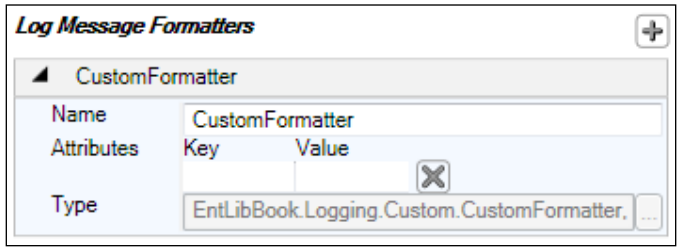
Configuring the custom log formatter

Configuration of the custom log message formatter is similar to that for Text Formatter and Binary Formatter; the configuration editor provides an option **Add Custom Log Message Formatter**. The following screenshot highlights the option:

Once we click on the **Add Custom Log Message Formatter** option, a dialog box to browse types that derive from `ILogFormatter` interface opens up. The following screenshot displays the dialog box with the selected custom log formatter:



After clicking the **OK** button, the configuration editor adds a new log formatter in the **Log Formatters** section. The following screenshot displays the newly added log formatter:



Attributes can be passed on to the custom formatter by providing the key and value details. Trace listeners will be able to select this custom formatter from the formatter drop-down list.

Implementing a custom log filter

We will implement a simple custom log filter to understand the creation of custom log filters; this log filter implements the `ILogFilter` interface and uses the configuration element type `CustomLogFilterData`. The `CustomLogFilterData` class provides the infrastructure (configuration data) for custom log filters. This custom filter allows us to pass the required information as custom attributes. We will use this to pass the name of a machine that should not be allowed to write log entries.

The following code snippet shows the list of required namespaces for the custom implementation:

```
using System.Collections.Specialized;
using Microsoft.Practices.EnterpriseLibrary.Common.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Logging.Filters;
```

The following code snippet shows the custom log filter implementation that filters log entries based on machine name:

```
[ConfigurationElementType(typeof(CustomLogFilterData))]
public class MachineNameLogFilter : ILogFilter
{
    string filterMachineName = string.Empty;

    public MachineNameLogFilter(NameValueCollection attributes)
    {
```

```
            filterMachineName = attributes["MachineName"];
        }

        public bool Filter(LogEntry log)
        {
            return string.Compare(log.MachineName, filterMachineName,
    true) != 0;
        }

        public string Name
        {
            get { return "Machine name Log Filter"; }
        }
    }
```

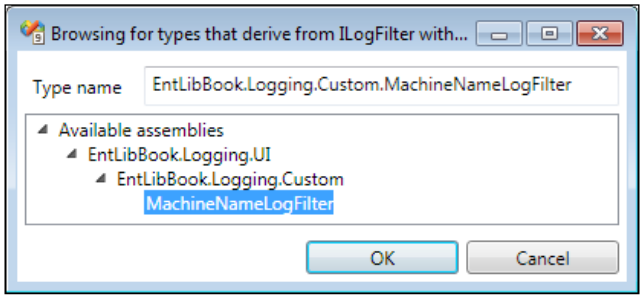
The `MachineNameLogFilter` class is annotated with a `ConfigurationElementType` attribute of `CustomLogFilterData`; we have implemented a constructor that accepts `NameValueCollection`, which contains the attributes added in the configuration. The `Filter` method is at the heart of the action that determines whether the machine name matches with the log entry's machine name; if there is a match then the method returns `false` to stop the log entry from being written.

Configuring the custom log filter

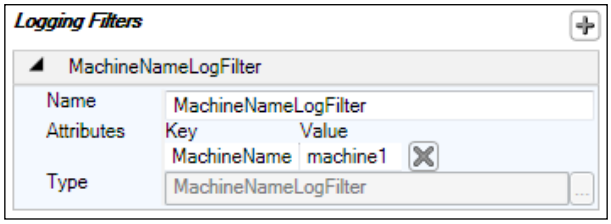
Configuration of a custom log filter is pretty straightforward: click on the plus symbol provided in the **Logging Filters** section, navigate, and click on the menu item **Add Logging Filters** | **Add Custom Logging Filter**.

The following screenshot displays the configuration option to add custom log filters:

After clicking on the menu item **Add Custom Logging Filter** a type-browser dialog box will be displayed with the list of available custom log filters. The following screenshot displays the type **MachineNameLogFilter**:



Select the custom log filter and click **OK** button; this will add the custom log filter to the configuration editor. The following screenshot displays the newly added custom log filter; an attribute **MachineName** with the value has been manually added for your reference.



The configuration specified in the above screenshot will block all log entries with machine name as "**machine1**".

Summary

In this chapter, we have explored the fundamental elements of the Logging Application Block such as Log Category, Special Category, Logging Trace Listeners, Log Formatters, Logging Filters, `Logger`, `LogWriter`, `LogEntry`, and so on. We have learned about the various required and optional assemblies and learned to set up the initial configuration. We have also learned to create `LogWriter` instances and write log entries using several overloaded `Write` methods. We have further explored how to configure trace listeners such as Event Log, Flat File, Rolling Flat File, XML Trace Listener, Database, Email, MSMQ, WMI, and so on; we have also explored the configuration options of Trace Source Categories and Logging Filters such as Category Filter, Priority Filter and Logging Enabled Filter. Finally, we implemented a custom trace listener, log formatter, and log filter.

4
Exception Handling Application Block

To err is human, and we developers are but humans. It's a myth that we can develop bug-free software at one go. We can definitely take measures to reduce bugs through test-driven development, unit test cases, stringent code check-in policy, and so on. However, the fact is that there are bugs in every application and they will show their ugly faces in the production environment. Additionally, applications have to face unforeseen scenarios such as the database server not being available, network failure, and so on. Hence, handling exceptions and providing meaningful and user-friendly messages to the user helps in avoiding/reducing user frustration. We need to handle exceptions not only to gracefully recover but also to log useful information, which can be used to fix bugs in the application.

Many good developers or project teams develop reusable components to handle and manage exceptions within and across software projects. Unfortunately, developing a good reusable component that caters to various requirements involves huge cost and effort, and also, maintenance of such in-house components is a nightmare. The Exception Handling Application Block is a reusable library that addresses many common requirements that developers have to deal with and there is enough room for extensibility through custom implementation to satisfy unique requirements. The beauty of the application block lies in its design. We have to sprinkle very much less code in our application to manage exceptions. The configuration determines how an exception is processed and the application code dictates which policy processes the exception. This flexibility allows the application to modify the exception handling process without recompiling the code.

In this chapter, you will:

- Receive an overview of the Exception Handling Application Block
- Be introduced to concepts such as **Exception Policy**, **Exception Types**, and **Exception Handlers**
- Learn about referencing the required and optional assemblies
- Learn about the initial infrastructure configuration using the configuration editor
- Learn about adding a namespace to avoid fully qualifying types
- Learn how to wrap exceptions using **Wrap Handler**
- Learn how to replace exceptions using **Replace Handler**
- Learn how to log exception information using **Logging Handler**
- Learn how to shield exceptions in WCF Service using **ExceptionShielding**
- Learn how to implement a custom **Exception Handler**

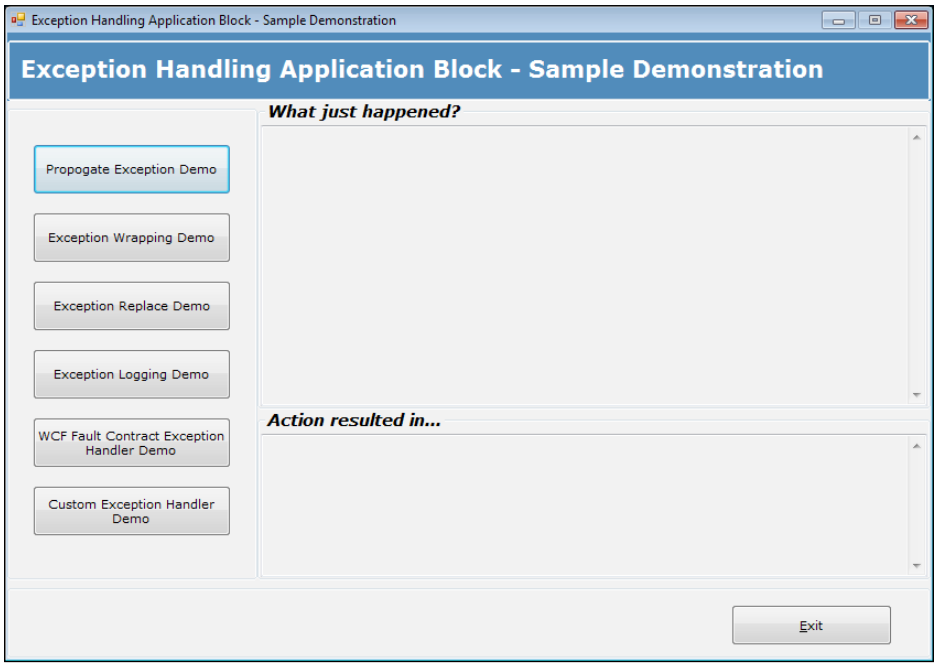
Developing an application

Before we leverage and dig deeper into individual features of the Exception Handling block, we will create a simple application that will help us to get up-to-speed with the basics. In this section, we will do the following:

- Reference the Exception Handling block assemblies
- Set up the initial configuration
- Add namespaces

To complement the concepts and sample code of this book and allow you to gain quick hands-on experience of different features of the Exception Handling Application Block, we have created a sample demonstration application, which simulates different layers of an application.

A screenshot of the sample application follows:



Referencing required assemblies

For the purposes of this demonstration, we will be referencing non-strong-named assemblies but, based on individual requirements, Microsoft strong-named assemblies or a modified set of custom assemblies can be referenced as well.

The following table lists the required/optional assemblies:

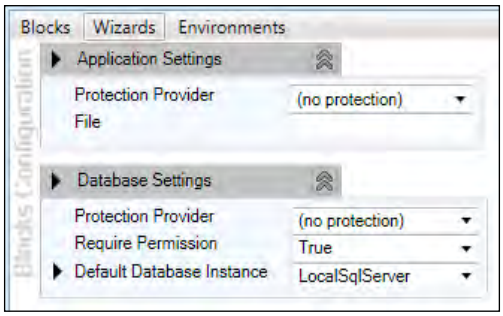
Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.dll`	Required

Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging.dll`	Optional Used while leveraging Logging functionality
`Microsoft.Practices.EnterpriseLibrary.Data.dll`	Optional; used only if exception logging is configured to be stored in database

Adding initial Exception Handling settings

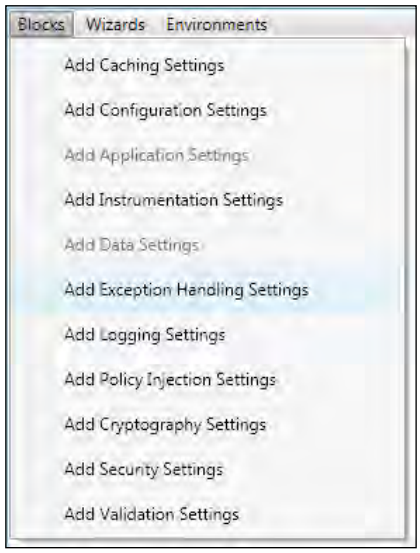
Before we can leverage the features of the Exception Handling block, we have to add the initial **Exception Handling Settings** to the configuration. Open the **Enterprise Library configuration** editor either using the shortcut available in **Start** | **All Programs** | **Microsoft patterns & practices** | **Enterprise Library 5.0** | **Enterprise Library Configuration** or just by right-clicking the configuration file in the **Solution Explorer** window of **Visual Studio IDE** and clicking on **Edit Enterprise Library V5 Configuration**. Initially, we will have a blank configuration file with default **Application Settings** and **Database Settings**.

The following screenshot displays the default settings displayed in the configuration editor:

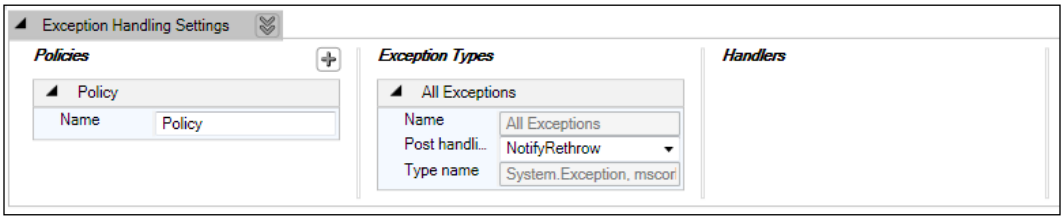


Let us go ahead and add the **Exception Handling Settings** in the configuration file. Select the menu option **Blocks**, which lists many different settings to be added to the configuration, and click on the **Add Exception Handling Settings** menu item to add the configuration settings.

The following screenshot shows the menu option **Add Exception Handling Settings**:



Once we click on **Add Exception Handling Settings** the configuration editor will display the default **Exception Handling Settings** as shown in the following screenshot:



Notice that the settings consist of three sections: **Policies**, **Exception Types**, and **Handlers**. By default, a policy named **Policy** with exception type **All Exceptions** is added to the configuration. We will change the default configuration later, but for now, we are in good shape with regards to the initial infrastructure configuration.

Adding namespaces

Instead of fully qualifying the type on every instance of its usage, we can add the namespace given below to the source code file to use the Exception Handling block elements without fully qualifying the reference.

Core Namespace:

- `Microsoft.Practices.EnterpriseLibrary.ExceptionHandling`

Configuration Namespace (Optional): Required while using the `EnterpriseLibraryContainer` to instantiate objects.

- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`

Unity Namespace (Optional): Required while instantiating objects using `UnityContainer`.

- `System.Configuration`
- `Microsoft.Practices.Unity`
- `Microsoft.Practices.Unity.Configuration`

WCF Namespace (Optional): Required while leveraging the Exception Handling block in a WCF Service.

- `Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.WCF`

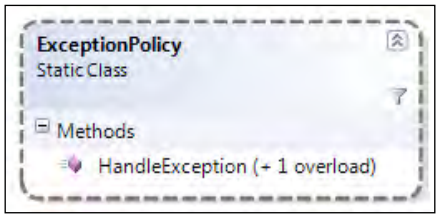
Understanding the Exception Handling block

The Exception Handling Application Block is driven by elements such as **Exception Policy**, **Exception Types**, **Exception Handler** and the `ExceptionManager` class.

Exception policy

Exception policy is like creating a group under which one or more exception types are configured and under each exception type, one or more handlers can be configured. For example, we might have **Data Access Exception Policy** to handle data access-related exceptions with multiple exception types such as `DBConcurrencyException`, `DbException`, and so on. While configuring the exception policy we have to provide a unique name, which can be used in the application code to process the exception.

The following class diagram shows the method exposed by the `ExceptionPolicy` class:



Exception types

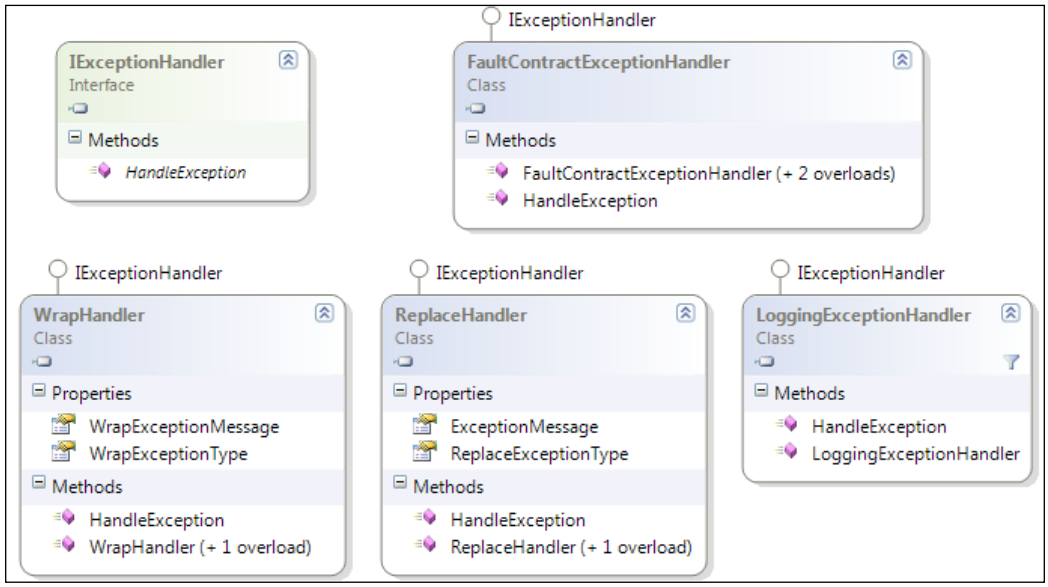
Exception type is nothing but any type that inherits from `System.Exception`; based on the configuration the application block chooses the matching exception type in the class hierarchy. If an exception of type `System.NotFiniteNumberException` is thrown and the policy is configured to handle exceptions of types `System.Exception` and `System.ArithmeticException` then the application block will process the exception using `System.ArithmeticException` based on the class hierarchy. While configuring the exception type, we have to decide on the **post handling action**. There are three options to choose from.

- **None**: Indicates to the calling code to continue execution.
- **NotifyRethrow**: Indicates to the calling code to throw the same exception.
- **ThrowNewException**: Indicates that the application block will throw an exception after executing all the configured handlers and the exception will be a result of the executed handlers.

Exception handler

Exception handlers are `.NET` classes that implement the Exception Handling block's interface called `IExceptionHandler`. The application block includes the four commonly required handlers **Wrap**, **Replace**, **Logging**, and **Fault Contract Exception Handler** (used to guard the WCF service boundary and generate new fault contract from the exception). We can also implement custom handlers to meet our custom requirements and configure them in the configuration file using the editor.

The following class diagram shows various concrete implementations of exception handlers and the `IExceptionHandler` interface:



The description of each of the concrete exception handlers is given as follows:

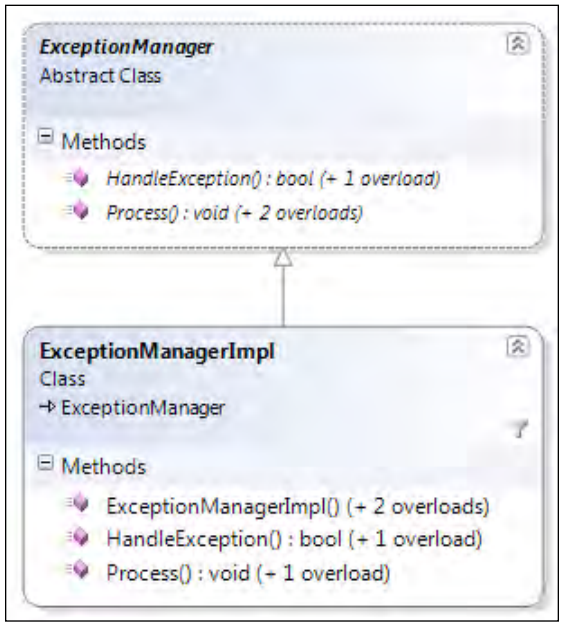
- **Wrap Handler**: The Wrap handler is very useful in scenarios where we want to provide a more meaningful message to the calling code rather than throwing the original exception. The original exception is wrapped with another exception that gives more detailed information to the caller. This exception handling pattern is referred to as the *Exception Translation* pattern.
- **Replace Handler**: The Replace handler as the name suggests replaces the original exception with the configured exception type; this avoids revealing sensitive information to the calling code. This exception handling pattern is referred to as the *Exception Shielding* pattern.
- **Logging Handler**: Handling exceptions is not enough; as developers we have to identify issues and resolve them. The Logging handler leverages the Logging Application Block to log exception details, which helps in issue identification and resolution. This exception handling pattern is referred to as the *Exception Logging* pattern.
- **WCF Fault Contract Exception Handler**: This guards the WCF service boundary and generates a new `FaultContract` from the exception; developers working on the WCF service would appreciate the ability to shield the exception and return the configured `FaultContract` based on the exception type as part of the response.

- **Custom Exception Handler**: Out of luck? None of the out-of-the-box handlers fits your requirement? Extensibility is the key aspect of the Enterprise Library; we can always write a custom exception handler by implementing the `IExceptionHandler` interface.

Exception Manager class

`ExceptionManager` is one of the key classes of the Exception Handling Application Block; this abstract class is part of the `Microsoft.Practices.EnterpriseLibrary.ExceptionHandling` namespace. It acts as an entry point to the exception handling functionality and provides two different ways to manage exceptions. The signatures of both the `HandleException` and `Process` methods are given next. The actual implementation is provided by the `ExceptionManagerImpl` class, which inherits from the `ExceptionManager` class.

The following class diagram shows the inheritance hierarchy and methods exposed by `ExceptionManager` and the `ExceptionManagerImpl` class:



HandleException method

The `HandleException` method provides granular control while processing exceptions; it returns a `Boolean` value indicating whether or not an exception re-throw is recommended. Typical usage of the `HandleException` method will be similar to the code snippet given next:

```
try
{
    BusinessLayer.BlogManager blogManager = new BusinessLayer.
BlogManager();

    //Get Blog Post
    BusinessLayer.BlogPost post = blogManager.GetBlogPost(0);
}
catch (ArgumentException ex)
{
    Exception exceptionToRethrow;

    //Get instance of ExceptionManager using static method of
Enterprise Library Container
    ExceptionManager exManager = EnterpriseLibraryContainer.Current.Ge
tInstance<ExceptionManager>();

    //Call to HandleException method
    //Return value indicates whether to re-throw the exception
    bool rethrow = exManager.HandleException(ex, "General Policy", out
exceptionToRethrow);

    if (rethrow) throw exceptionToRethrow;
}
```

Process method

The `Process` method automatically performs exception management and throws the exception based on the configuration. It accepts the policy name and a delegate or a lambda expression; the application block manages any exception that occurs while executing the method or lambda expression, also if the `postHandlingAction` is set to `ThrowNewException` then the application block throws the exception as a result of the respective execution of the configured exception handlers.

Typical usage of the `Process` method will be similar to the code snippet given next:

```
//Get instance of ExceptionManager using static method of Enterprise
Library Container
ExceptionManager exManager = EnterpriseLibraryContainer.Current.GetIns
tance<ExceptionManager>();
```

```
BusinessLayer.BlogPost post = null;
BusinessLayer.BlogManager blogManager = new BusinessLayer.
BlogManager();

//try..catch block not required...
//Automatic Exception Management through Process method
post = exManager.Process<BusinessLayer.BlogPost>(
                                                    () => { return
blogManager.GetBlogPost(0); },
                                                    "Data Access Policy"
                                                    );
```

Stitching together: Exception Policy/Type/Handler

Let us stitch together the three fundamental elements **(exception policy**/**type**/**handler**) to put things in perspective and understand them better. Imagine that we want all the database-related exceptions to be replaced with a new exception to prevent disclosing the connection string information; additionally, we want to log the exception in a file to identify the root cause of the exception. To achieve this, we define a policy named **Data Access Exception Policy**. Now we can associate one or more data access-related exception types (`SqlException`, `SqlTypeException`, `DBConcurrencyException`, and so on) and configure one or more exception handlers for each exception type. As we want to replace the exception and also log the exception information, we can configure the Logging handler first and then the Replace handler.

Creating an Exception Handling block object

We have several options at hand while creating an Exception Handling object such as using the static `ExceptionPolicy` class, using Unity service locator, and using Unity container directly. A few approaches such as configuring the container through a configuration file or code are not listed here but the recommended approach is either to use the Unity service locator for applications with few dependencies or create objects using Unity container directly to leverage the benefits of this approach. Use of the static factory class is not recommended.

Using the ExceptionPolicy class

`ExceptionPolicy` is a static class and is part of the `Microsoft.Practices.EnterpriseLibrary.ExceptionHandling` namespace. This class contains static methods to handle exceptions. Internally, it leverages `EnterpriseLibraryContainer`, which is part of the `Microsoft.Practices.EnterpriseLibrary.Common.Configuration` namespace. This class is an entry point for the container infrastructure for the Enterprise Library. The `ExceptionPolicy` class was the default approach to handle exceptions in versions prior to 5.0. This approach is no longer recommended and is still available for backwards compatibility.

The following is the syntax to handle exceptions using the `ExceptionPolicy` static class:

```
try
{
    //Potentially exceptional area :)
}
catch (Exception ex)
{
    bool rethrow = ExceptionPolicy.HandleException(ex, "UI Policy");

    if (rethrow)
    {
        throw;
    }
}
```

Using Unity service locator

This approach is recommended for applications with few dependencies. The `EnterpriseLibraryContainer` class exposes a static property called `Current` of type `IServiceLocator`, which resolves and gets an instance of the specified type.

The following is the syntax to create an instance of `ExceptionManager` using Unity service locator:

```
//Get instance of ExceptionManager using static method of Enterprise
Library Container
ExceptionManager exManager = EnterpriseLibraryContainer.Current.GetIns
tance<ExceptionManager>();
```

Using Unity container directly

Larger complex applications demand looser coupling. This approach leverages the dependency injection mechanism to create objects instead of explicitly creating instances of concrete implementations. Unity container resolves objects using type registrations and mappings; these can be configured programmatically or through a configuration file. Based on the configuration, it resolves the appropriate type whenever requested. The following example instantiates a new Unity container object and adds the Enterprise Library Core Extension. This loads the configuration and makes registrations and mappings of Enterprise Library available.

The following is the syntax to create an instance of `ExceptionManager` directly using Unity Container:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
ExceptionManager exManager = container.Resolve<ExceptionManager>();
```

Wrapping an exception using Wrap handler

The Exception Handling block provides an out-of-the-box handler called **Wrap Handler**, which allows us to configure the wrap exception type and the exception message. We can also load the exception message from a resource file by specifying the message resource name and resource type. Based on the configuration, the exception is wrapped using the new exception type with the specified exception message. The new exception object contains the original exception as part of the **InnerException**.

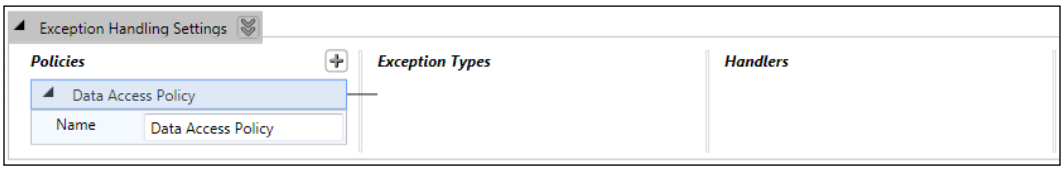
Wrapping the original exception with a new exception type is useful in the scenarios given next:

- Updating the error message of the original exception with a more meaningful message while maintaining the original context
- Throwing a specific exception (`DataLayerException`, `BusinessLayerException`, `FatalException`, `NonFatalException`, and so on) across layers/boundaries while maintaining the original context

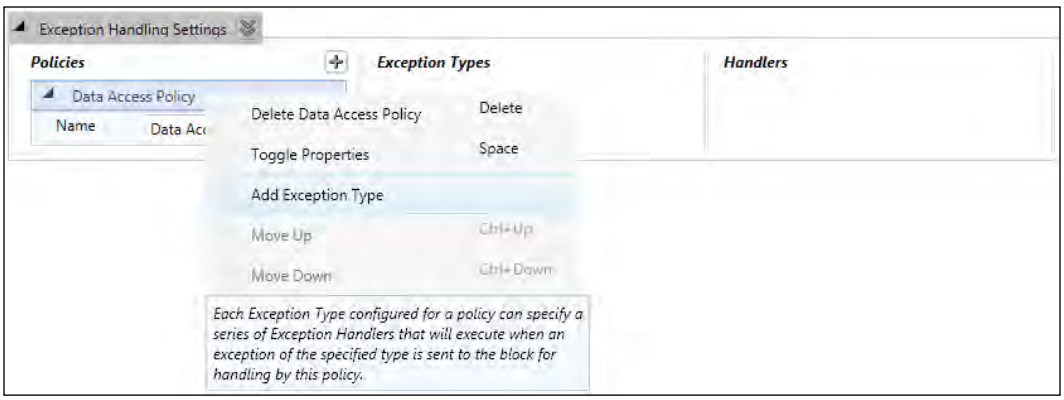
Configuring a Wrap exception handler

We currently have the default settings in the configuration file; to understand the configuration we will delete the default policy named **Policy**. The steps to configure Wrap Handler are given as follows:

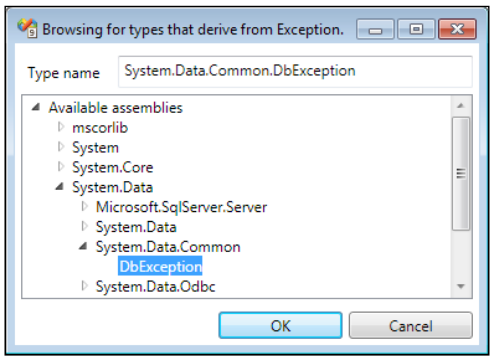
1. Add a new policy in the policies section and name it **Data Access Policy** as shown in the following screenshot.



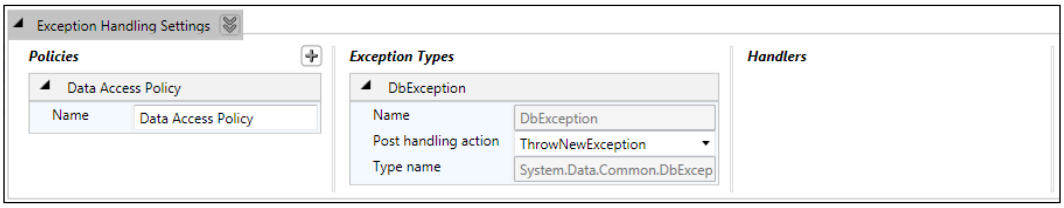
2. Right-click on the **Data Access Policy** and click on **Add Exception Type**. This will pop up a new exception type selection dialog.



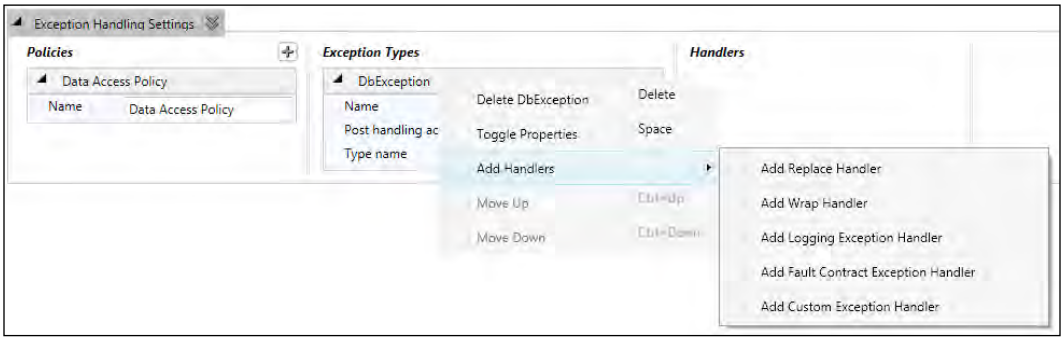
3. Specify the type name as **DbException** by keying in the type name.



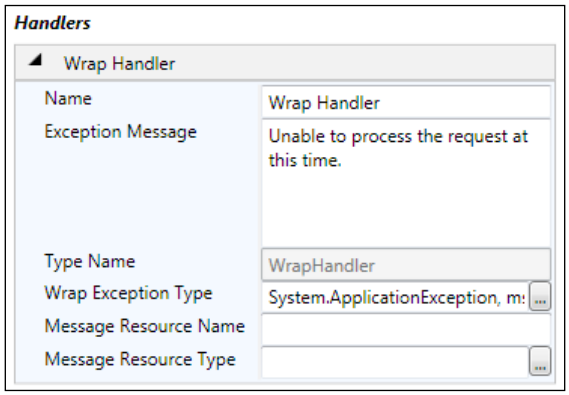
4. Set the **Post handling action** attribute of the exception type **DbException** to **ThrowNewException**.



5. So far we have added the policy and the exception type, now let's add the Wrap Handler. Right-click on the exception type **DbException** and click on **Add Handlers | Add Wrap Handler**.



6. Set the exception message and also set the **Wrap Exception Type** to **System.ApplicationException**.



So far we have successfully configured the application with a policy that will process an exception of type `DbException` or any matching exception in the class hierarchy. It also wraps the exception with a new `System.ApplicationException` with the specified exception message. Once the exception is wrapped with the new exception object, the new exception is thrown as configured in the post-handling action.

The following is the execution result of **Exception Wrapping Demo** provided as part of the sample application with this book:

```
Exception Message:
==================
Wrapped Exception: Database operation failed due to concurrency issue.
Error code: f294419a-b4b5-47ad-9e9e-ec62362965f2


Inner Exception Message:
========================
Original Exception: Concurrency violation: the UpdateCommand affected 0
records.
```

We can see in the given result that the original exception is wrapped with a new exception message. This helps in retaining the context yet providing more meaningful information to the application user.

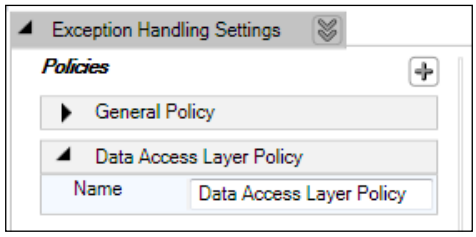
Replacing an exception using Replace handler

Replacing an exception is one of the common requirements to avoid exposing more than the required information especially while dealing with sensitive information. Imagine a scenario where a component throws the original exception, which might contain sensitive information such as connection string, stack trace, and so on, to the consumer. The Exception Handling Application Block provides the Replace handler to replace the exception with a custom exception and message.

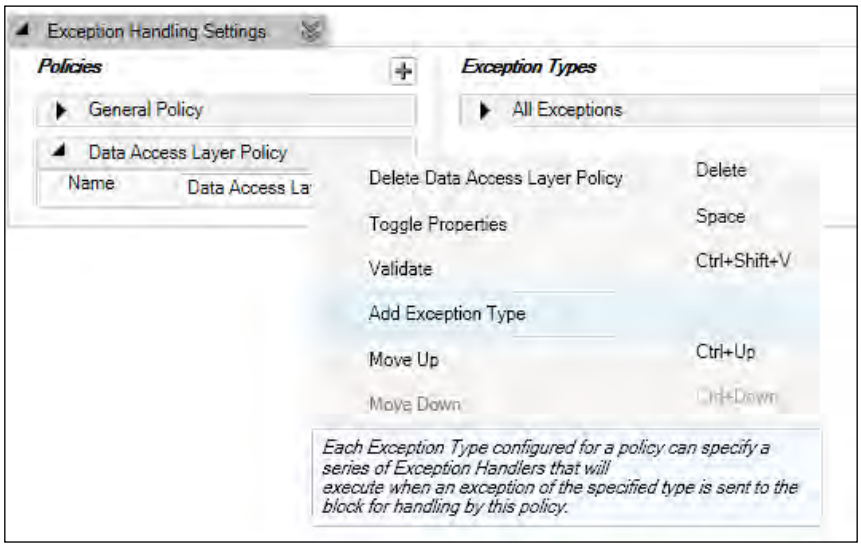
Configuring a Replace handler

Let us update the existing configuration file with a Replace handler. In this scenario, we will replace a `DbException` with an `ApplicationException` and set a custom message. This prevents the **Data Access Layer** from exposing sensitive information such as connection string and so on.

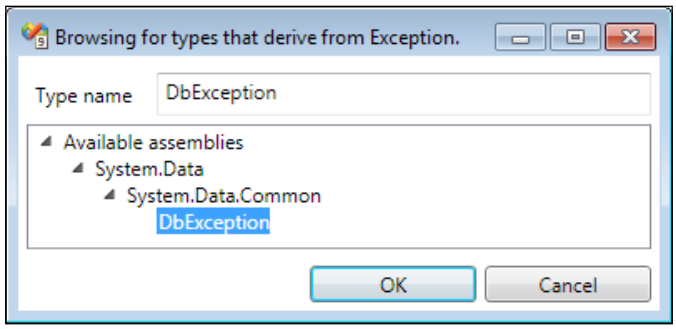
1. Add a new policy in the policies section and name it **Data Access Layer Policy** as shown in the following screenshot:



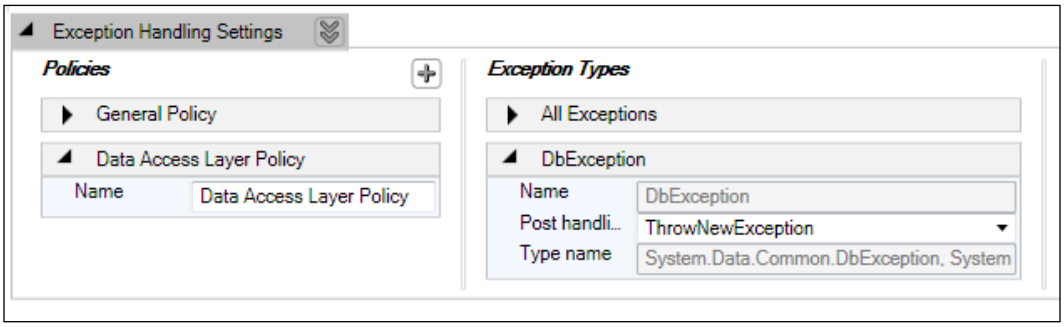
2. Right-click on the **Data Access Layer Policy** and click on **Add Exception Type**; this will pop up a new exception type selection dialog.



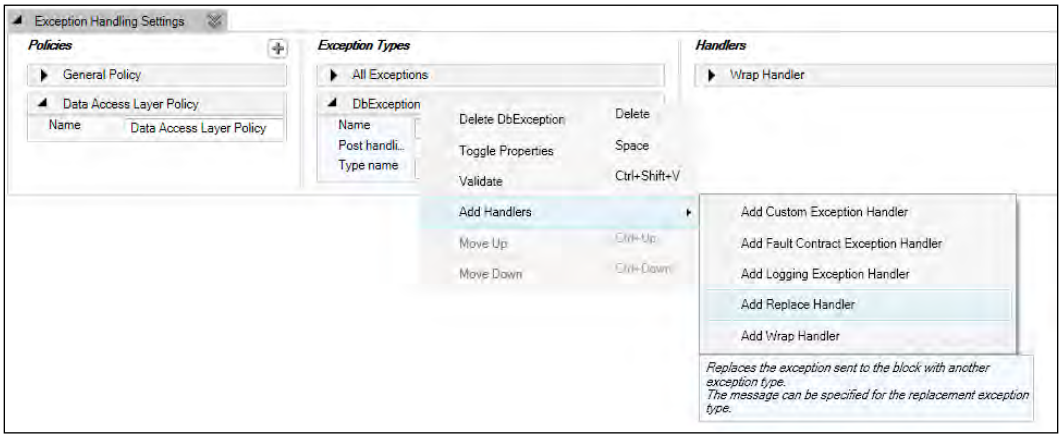
3. Specify the type name as **DbException** by keying in the type name.



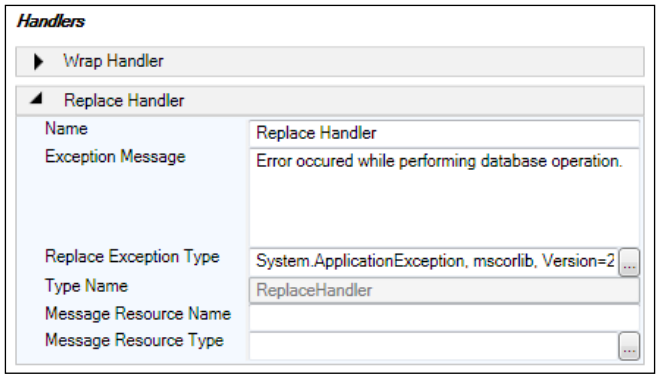
4. Set the **Post handling action** attribute of the exception type **DbException** to **ThrowNewException**.



5. So far we have added the policy and the exception type. Now let's add the Replace exception handler. Right-click on the exception type **DbException** and click on **Add Handlers | Add Replace Handler**.



6. Set the exception message and also set the replace exception type to **System.ApplicationException**.



So far we have successfully configured the application with a policy that will process an exception of type `DbException` or any matching exception in the class hierarchy. It also replaces the exception with a new `System.ApplicationException` with the specified exception message; a new exception is thrown as configured in the post-handling action.

The following is the execution result of **Exception Replace Demo** provided as part of the sample application with this book:

```
Exception Message:
==================
Replaced Exception: Application exception occured. Error code: 6a45a1e8-
e131-421d-a1f4-7a73cdb16198
```

As we can see in the above result, the original `DbException` has been replaced with `ApplicationException`. Additionally, the message also provides a unique error code to trace the root of the exception. This helps in protecting sensitive data from being exposed to other layers or users.

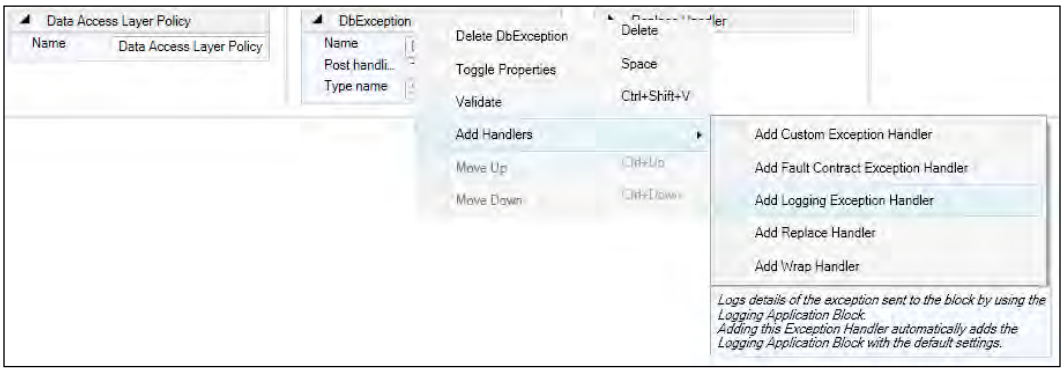
Logging an exception using Logging handler

Logging exceptions/errors provides valuable information; this information can be analyzed and issues can be resolved proactively. The Logging handler leverages the Logging block to log exception information. As mentioned earlier, every exception type can have one or more handlers and generally the Logging handler is used in combination with a Wrap or Replace handler.

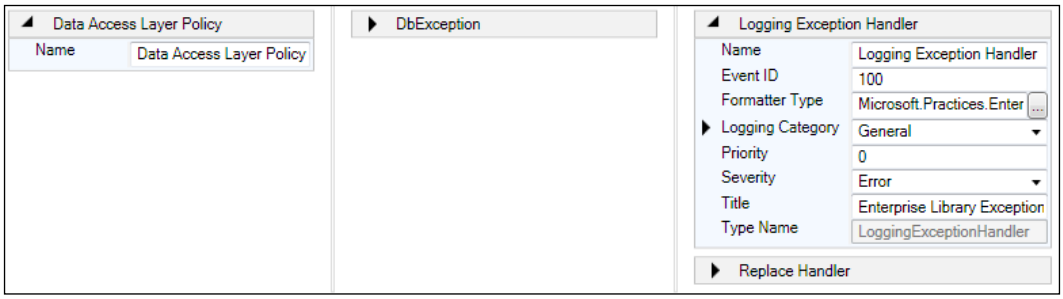
Configuring a Logging handler

We will update the existing configuration and add a Logging handler for the `DbException` type associated to the policy named **Data Access Layer Policy**.

Right-click on the **DbException** type in the Exception Types section of the **Data Access Layer Policy** and click on **Add Handlers | Add Logging Exception Handler** as shown in the following screenshot:

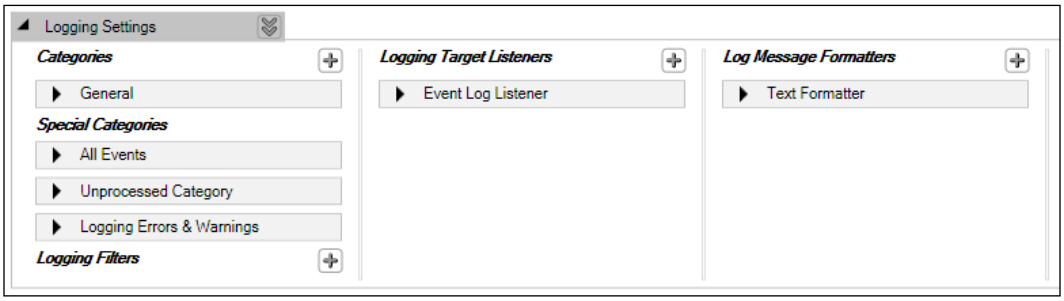


The following screenshot shows the default configuration of **Logging Exception Handler**:



Notice the Logging Exception Handler is moved up and is the first handler in the hierarchy; this is explicitly done to log the original exception and not the replaced or wrapped exception.

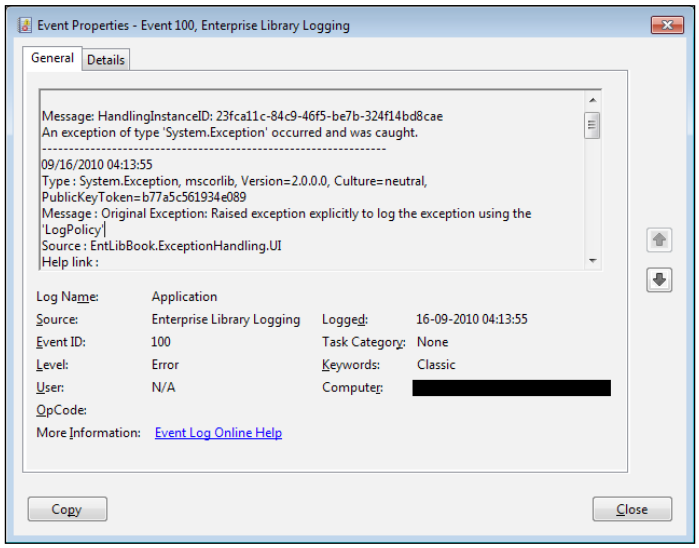
The following screenshot shows the **Logging Settings** added to the configuration editor.



The Logging Application Block is explained in more detail in the respective chapter. For this particular functionality, we will be using the default logging configuration.

Now, in our code when we encounter an exception of type `DbException` either managed through the `Process` or `HandleException` method of the `ExceptionManager` class, the application block will first log the exception in Windows Application Event Log as it is the default configuration. Later, the second handler will be invoked, which replaces the original exception with a new exception.

The following screenshot shows the exception logged in Windows Event Log; this is the execution result of **Exception Logging Demo** provided as part of the sample application with this book:



WCF fault contract exception handler

The Exception Handling Application Block also provides a handler to shield exceptions for **Windows Communication Foundation** (**WCF**) services. It is very important to implement an Exception Shielding pattern at service boundary level and this handler makes it very easy to prevent any sensitive information crossing the service boundary. This is implemented as part of the `Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.WCF` assembly, which needs to be referenced to leverage the functionality. We have to create a fault contract for our WCF service, configure the exception handling policy to use the fault contract exception handler, and map the created fault contract type.

Imagine we have a WCF service called `BlogService` and we want to prevent all original exceptions from being thrown to the service consumer. We also want to replace such exceptions with a generic fault contract providing a generic error message and an error code. In order to satisfy such a requirement, we will implement a fault contract called `GenericFaultContract` and configure the Exception Policy to replace all exceptions with an instance of `GenericFaultContract`.

Generic fault contract creation

We will create a simple fault contract to hold the error code and the message. A simple `GenericFaultContract` class is given next. This fault contract will be used to configure the exception handlers section mapped to an exception type.

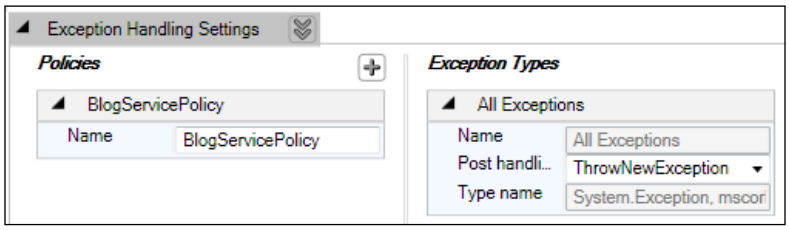
The following code snippet shows the `GenericFaultContract` class:

```
using System;
using System.Runtime.Serialization;

namespace EntLibBook.ExceptionHandling.ServiceLayer
{
    [DataContract]
    public class GenericFaultContract
    {
        [DataMember]
        public Guid FaultID { get; set; }

        [DataMember]
        public string FaultMessage { get; set; }
    }
}
```

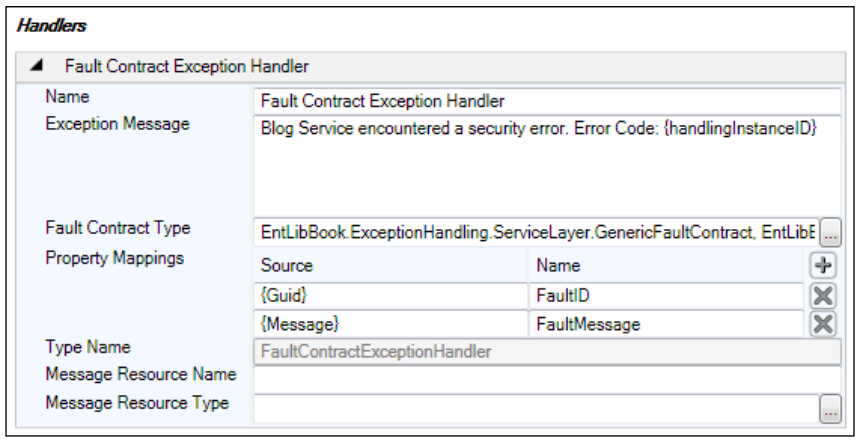
Configuring a fault contract exception handler

As we have the fault contract ready, we will use the configuration editor to edit the WCF service configuration file and add a **BlogServicePolicy** policy that handles all exceptions with a post-handling action to throw new exception.

The following screenshot shows the **Exception Handling Settings** with a configured policy named **BlogServicePolicy** and exception type as **All Exceptions**:



Alright, now we have the exception policy and type in place, let us add the fault contract exception handler and configure it to use the `GenericFaultContract` class. The following screenshot shows the configured handler called **Fault Contract Exception Handler**.



There are two important things to notice in the handler configuration. The exception message contains a token **{handlingInstanceID}**, which is replaced by a `Guid` generated by the application block. This `Guid` can be very useful if the exceptions are logged, since the support staff can look into the configured log store for more information on what went wrong using the handling instance identifier. Another important aspect in the configuration is the property mappings; we are mapping the tokens to the property of our `GenericFaultContract` class. The generated fault contract will have both the properties populated based on the configured tokens.

Applying the ExceptionShielding attribute

Now the final task in the WCF service is to apply the `ExceptionShielding` attribute either to the `ServiceContract` interface or to the class. The `ExceptionShielding` attribute instructs the Exception Handling Application Block to handle exceptions based on the configured exception policy.

The following code snippet shows the `ExceptionShielding` attribute in action:

```
using EntLibBook.ExceptionHandling.BusinessEntities;
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.WCF;

namespace EntLibBook.ExceptionHandling.ServiceLayer
{
    [ExceptionShielding("BlogServicePolicy")]
    public class BlogService : IBlogService
    {
        public BlogPost GetBlogPost(int id)
        {
            // Code to get Blog Post
            return null;
        }
    }
}
```

Exception handling: WCF Service consumer

The WCF Service consumer will be able to handle the Fault Exception by specifying the `GenericFaultContract` type and will be able to access the `FaultID` and `FaultMessage` properties.

The following code snippet shows the `catch` block with the `FaultException` as `GenericFaultContract` type:

```
try
{
    ServiceProxy.BlogServiceClient client = new ServiceProxy.
BlogServiceClient();

    client.GetBlogPost(1);
}
catch (FaultException<ServiceProxy.GenericFaultContract>
faultException)
{
    // Note: Just to demonstrate this scenario we are assigning the
             property
```

```
        // We can use Exception Handling Application Block to manage the
           exception here as well.

        // Retrieving FaultID and FaultMessage
        Guid handlingInstanceID = faultException.Detail.FaultID;
        string faultMessage = faultException.Detail.FaultMessage;
    }
```

Whenever the call to `GetBlogPost` throws an exception, the client code will receive a `FaultException` of `GenericFaultContract` type, which will be handled by the client code as part of the structured exception handling.

Implementing custom exception handler

Extensibility is the key feature of any Enterprise Library Application Block and hence, this block is extensible too. The Exception Handling block provides two areas for extensions, Exception Handler and Exception Formatter. We have already understood the concept of an exception handler and have also used many out-of-the-box handlers. Implementing a custom handler is very easy: we have to implement the `IExceptionHandler` interface and provide our custom implementation of the `HandleException` method. To understand and learn to implement a custom exception handler, let us create an exception handler that displays a message box to the user in a Windows Forms application.

The following code snippet provides the implementation of a custom exception handler that displays a message box whenever it receives a request to handle exception:

```
[ConfigurationElementType(typeof(CustomHandlerData))]
public class WindowsMessageExceptionHandler : IExceptionHandler
{
    public WindowsMessageExceptionHandler(NameValueCollection ignore)
    {
    }

    public Exception HandleException(Exception exception, Guid
handlingInstanceId)
    {
        MessageBox.Show(exception.Message, "Error", MessageBoxButtons.
OK, MessageBoxIcon.Error);

        return exception;
    }
}
```

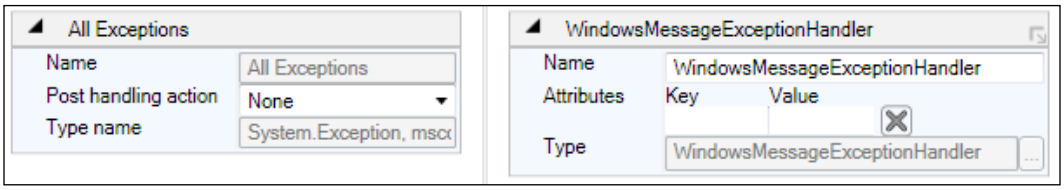
The `WindowsMessageExceptionHandler` class inherits from the `IExceptionHandler` interface and provides implementation for the `HandleException` method; this method shows a message box with the exception message. The custom exception handler implementation is decorated with the `ConfigurationElementType` attribute with the `CustomHandlerData` type as parameter; this essentially indicates the configuration object type.

Configuring custom exception handler

Configuration for the custom exception handler is similar to that for other handlers; we just need to add the `WindowsMessageExceptionHandler` in the configuration editor. Right-click on the exception type and click on **Add Handlers | Add Custom Exception Handler**. In the selection dialog, select the `WindowsMessageExceptionHandler` class.

The following screenshot shows the configuration screen for **Add Custom Exception Handler**:

The following screenshot shows `WindowsMessageExceptionHandler` added to the configuration editor:



We are done with our implementation and configuration of the custom exception handler. While handling exceptions in code using the specific policy mapped with the custom handler, it will display a message box to the user with the error message.

Summary

In this chapter, we have learned about the fundamental elements of the Exception Handling Application Block such as Exception Policy, Exception Types, and Exception Handler. We have learned about the required and optional assemblies, the initial infrastructure configuration, and the individual feature-level configuration. We have also explored and learned to leverage different Exception Handlers and implemented a custom Exception Handler.

5

Caching Application Block

Performance, **Scalability**, and **Availability** are three key design elements that are considered while designing enterprise-class applications. Judicious use of caching techniques goes a long way in improving and strengthening these elements. **Caching** is not rocket science but judiciously caching data involves some thoughtfulness. Data involving enormous processing/computation, expensive-to-retrieve data, and data that changes infrequently and/or is consumed quite often are great candidates for caching. Caching helps in improving performance by storing data either in-memory or to some persistent storage for quicker retrieval compared to the original source.

Caching is an important aspect of any enterprise application but it is a daunting task to develop a caching library that satisfies the requirements of different projects. The **Caching Application Block** fills that gap by providing a ready-to-use infrastructure for caching. It supports both in-memory caching as well as backing storage (Database or Isolated Storage); customization is also possible through extension points. The Caching block provides all the common functionality to add, retrieve, remove, and flush cached data. Also, cache expiration and scavenging policy can be controlled through configuration.

The following are the key features of the Caching Application Block:

- Manage configuration settings through Enterprise Library configuration tool
- In-memory, isolated, or database persistent cache storage location can be configured
- Policy-based expiration and scavenging, both configurable
- Support for custom expiration policies and storage location
- Extensibility points to implement custom backing store, expiration policy, storage encryption provider, and cache manager

The Enterprise Library Caching Application Block will be deprecated in future releases. Caching functionality is available in .NET 4.0 as part of `System.Runtime.Caching` namespace; this implementation is not dependent on the `System.Web` assembly and it can be used by other .NET applications, not just ASP.NET.

The Caching Application Block can be used with any of the following application types:

- Console Application
- Windows Forms
- ASP.NET Web Application or Web Service
- Windows Communication Foundation (WCF)
- Windows Presentation Foundation (WPF)
- Windows Service

Caching Application Block operations are both thread safe and exception safe.

The Caching Application Block can be leveraged for several different scenarios. The key scenarios for the Caching block are as follows:

- Consistent approach to caching across different application environments. Basically, it doesn't matter whether it's a Web application, Windows Forms application, WCF, WPF, and so on.
- Requires a configuration-based caching where the key elements can be modified during production deployment if required.
- Option to cache in a persistent backing store: The Caching Application Block provides support to store the cache data to both database and isolated storage. Additionally, cached data can be encrypted before persisting into a backing store. The backing store can be extended by creating a custom backing store provider.

In this chapter, you will:

- Be introduced to the Caching Application Block
- Understand the scenarios for the Caching Application Block
- Understand the concepts behind the Caching Application Block
- Learn about different backing stores such as `NullBackingStore`, `IsolatedStorageBackingStore`, and `DataBackingStore`
- Learn about referencing the required assemblies
- Learn to set up the initial infrastructure configuration using the configuration editor
- Learn to cache, retrieve, remove, and flush cached data using the Caching Application Block
- Understand and implement the cache item refresh action using `ICacheItemRefreshAction`
- Learn to configure `IsolatedStorageBackingStore` and `DataBackingStore`
- Learn to configure an encryption provider to encrypt cached data

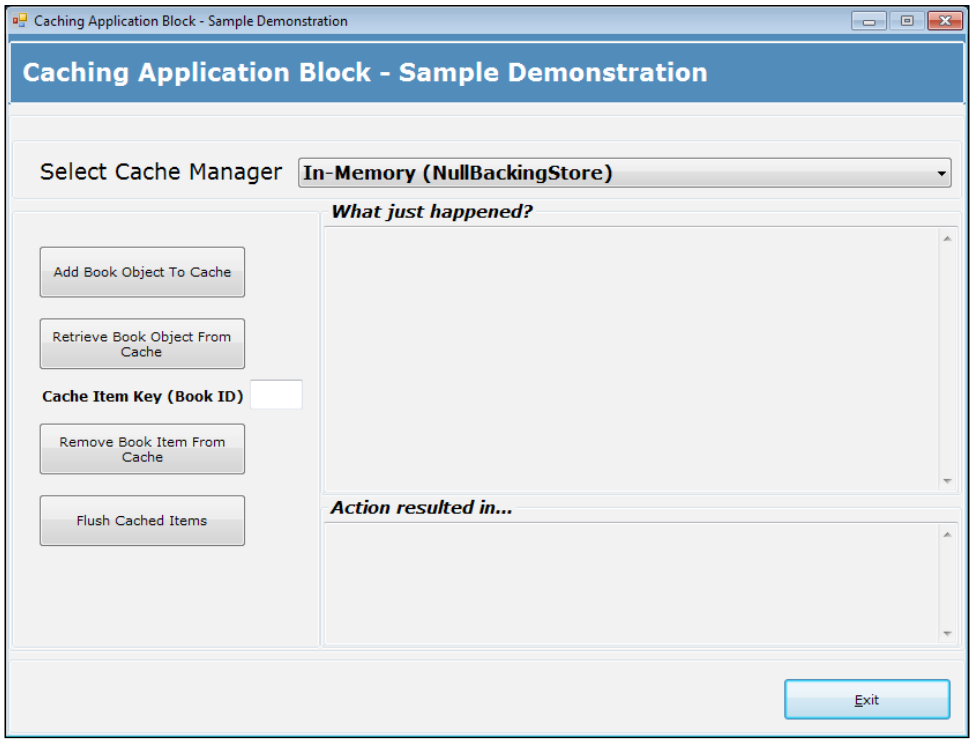
Developing an application

Before we dig deeper into individual features of the Caching block, we will touch upon the basic elements by creating a sample Windows Forms Application project. This will help us to get up-to-speed with the basics; in this section, we will do the following:

- Reference the Caching block assemblies
- Set up the initial configuration
- Write code to add items to the cache

To complement the concepts and sample code of this book and allow you to gain quick hands-on experience of different features of the Caching Application Block, we have created a sample demonstration application, which provides implementation of Add/Retrieve/Remove/Flush operations utilizing an in-memory and Isolated Storage Cache Manager (configured to encrypt cached data).

The following is a screenshot of the sample application:



Referencing the required assemblies

For the purposes of this demonstration, we will be referencing non-strong-named assemblies but based on individual requirements, Microsoft strong-named assemblies, or a modified set of custom assemblies can be referenced as well. Since we will also be exploring storage of cached items to a database and encryption of cached items feature in this chapter, we need to include references to the database and cryptography-related assemblies in the project.

The following table lists the required/optional assemblies.

Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required

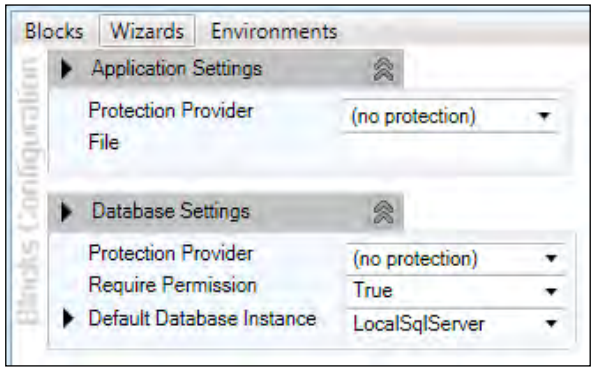
Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Caching.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Caching.Database.dll`	Optional.
`Microsoft.Practices.EnterpriseLibrary.Data.dll`	Only if database caching is required.
`Microsoft.Practices.EnterpriseLibrary.Security.Cryptography.dll`	Optional.
`Microsoft.Practices.EnterpriseLibrary.Caching.Cryptography.dll`	Only if data encryption is required for cached data.

Adding the initial Caching Settings

Before we can leverage the features of the Caching Application Block we have to add the initial **Caching Settings** to the configuration. The following steps will add the settings to the configuration file:

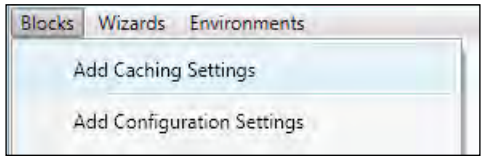
1. Open the **Enterprise Library configuration editor** either using the shortcut available in **Start** | **All Programs** | **Microsoft patterns & practices** | **Enterprise Library 5.0** | **Enterprise Library Configuration** or just by right-click-the configuration file in the **Solution Explorer** window of **Visual Studio IDE**.
2. Next, click on **Edit Enterprise Library V5 Configuration**. Initially, we will have a blank configuration file with default **Application Settings** and **Database Settings**.

The following screenshot shows the default configuration settings:

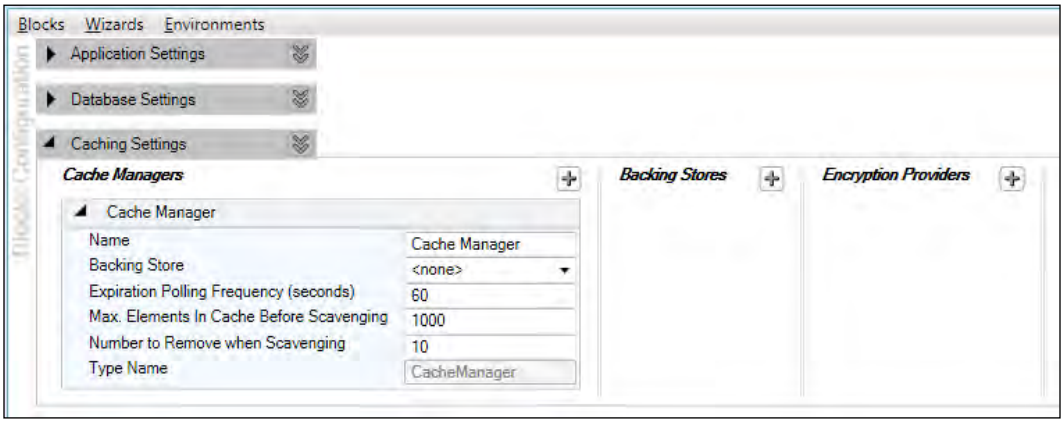


3. Now let us add the **Caching Settings** in the configuration file. Select the menu option **Blocks**, which lists many different settings to be added to the configuration, and click on the **Add Caching Settings** menu item to add the caching configuration settings.

   The following screenshot shows the **Add Caching Settings** menu item in the **Blocks** menu:



4. Once we click on **Add Caching Settings**, the configuration editor will display the default **Caching Settings** as shown in the following screenshot:



Notice that the settings consist of three parts: **Cache Managers**, **Backing Stores,** and **Encryption Providers**. By default, the setting is configured to use the default **CacheManager** provider and also the other attributes are set with the default values. We will change the default configuration further but for now, we are in good shape with regards to the initial infrastructure configuration.

Adding namespaces

We definitely don't want to get bored by fully qualifying the type on every instance of its usage, so to make our life easy we can add the given namespaces to the Windows Form's source code file to use the Caching block elements without fully qualifying the reference. Although we will be using **EnterpriseLibraryContainer** to instantiate objects (so we will also add `Microsoft.Practices.EnterpriseLibrary.Common.Configuration` namespace to the source file), the **Unity Namespace** section is listed to make you aware of the availability of the alternative approach of instantiating objects.

Core Namespaces:

- `Microsoft.Practices.EnterpriseLibrary.Caching`
- `Microsoft.Practices.EnterpriseLibrary.Caching.Expirations`

Configuration Namespace (Optional): Required while using the `EnterpriseLibraryContainer` to instantiate objects.

- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`

Unity Namespace (Optional): Required while instantiating objects using **UnityContainer**.

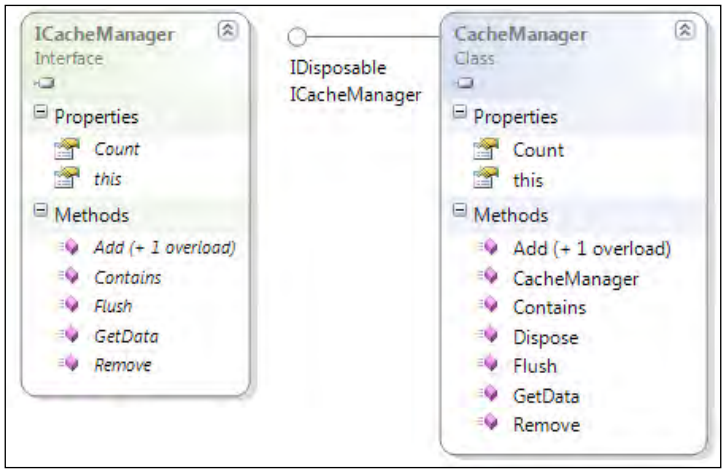
- `System.Configuration`
- `Microsoft.Practices.Unity`
- `Microsoft.Practices.Unity.Configuration`

Creating the CacheManager instance

The `CacheManager` class is the default implementation of the **ICacheManager** interface, which resides in the `Microsoft.Practices.EnterpriseLibrary.Caching` namespace. As the name implies, it acts as a manager and manages all the caching operations. `CacheManager` internally creates a **Cache** object during initialization and this `Cache` object holds the real cache; all the requests (add, retrieve, remove, and so on) are forwarded to the `Cache` object.

Operations performed using the default `CacheManager` object are thread safe.

The following diagram shows the definitions of the `ICacheManager` interface and of the `CacheManager` class.



We have several options at hand while creating a `CacheManager` object such as using the static **CacheFactory** class, using Unity service locator and using Unity container directly. A few approaches such as configuring the container through a configuration file or code are not listed here but the recommended approach is either to use the Unity service locator for applications with few dependencies or create objects using Unity container directly to leverage the benefits of this approach. Use of the static factory class is not recommended.

Using the static factory class

Static factory classes were the default approach to creating objects with versions prior to 5.0. This approach is no longer recommended and is still available for backwards compatibility. The Caching Application Block provides a static class called `CacheFactory` available in the `Microsoft.Practices.EnterpriseLibrary.Caching` namespace. Once the `CacheManager` object is created it in turn creates a `CacheManagerFactory` object, which in turn creates a `Cache` object.

The following is the syntax to create a deafult `CacheManager` instance using the static factory class:

```
ICacheManager cacheManager = CacheFactory.GetCacheManager();
```

The following is the syntax to create a named `CacheManager` instance using the static factory class:

```
ICacheManager cacheManager = CacheFactory.GetCacheManager("Isolated
Storage Cache Manager");
```

Using the Unity Service Locator

This approach is recommended for applications with few dependencies. The `EnterpriseLibraryContainer` class exposes a static property called `Current` of type **IServiceLocator**, which resolves and gets an instance of the specified type.

The following is the syntax to create a default `CacheManager` instance using Unity Service Locator:

```
ICacheManager cacheManager = EnterpriseLibraryContainer.Current.
GetInstance<ICacheManager>();
```

The following is the syntax to create a named `CacheManager` instance using Unity Service Locator:

```
ICacheManager cacheManager = EnterpriseLibraryContainer.Current.GetIns
tance<ICacheManager>("Isolated Storage Cache Manager");
```

Using the Unity container directly

Larger complex applications demand looser coupling; this approach leverages the dependency injection mechanism to create objects instead of explicitly creating instances of concrete implementations. Unity container resolves objects using type registrations and mappings; these can configured programmatically or through a configuration file. Based on the configuration, it resolves the appropriate type whenever requested. The following example instantiates a new Unity container object and adds the Enterprise Library Core Extension. This loads the configuration and makes registrations and mappings of Enterprise Library available.

The following is the syntax to create a default `CacheManager` instance directly using Unity container:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
ICacheManager cacheManager = container.Resolve<ICacheManager>();
```

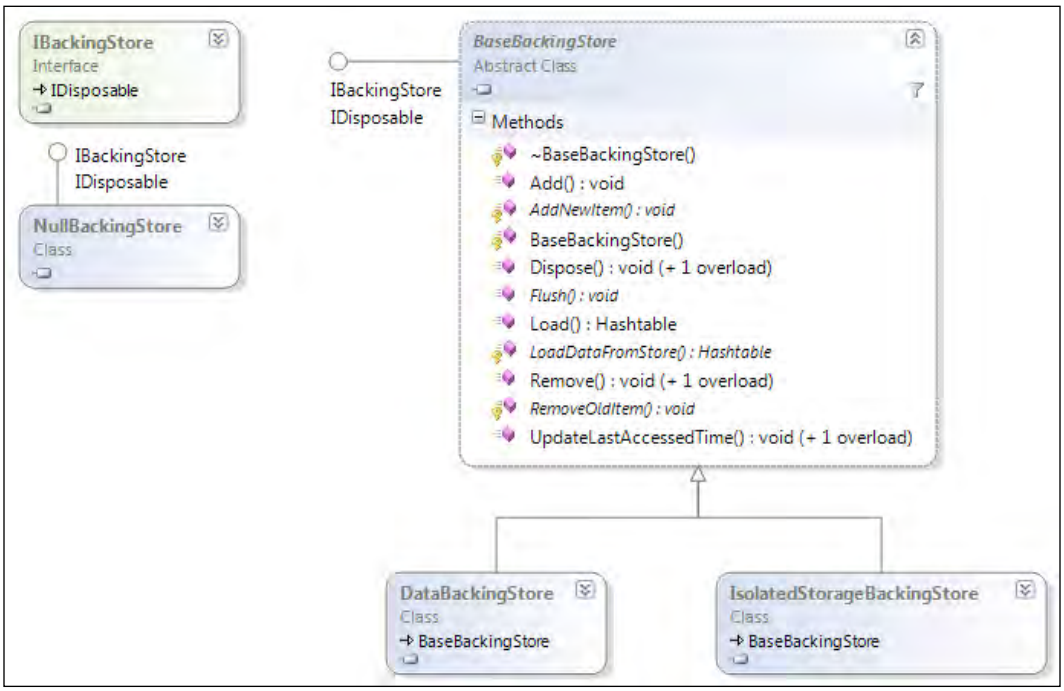
The following is the syntax to create a named `CacheManager` instance directly using Unity container:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
ICacheManager cacheManager = container.Resolve<ICacheManager>("Isolate
d Storage Cache Manager");
```

Configuring the in-memory backing store

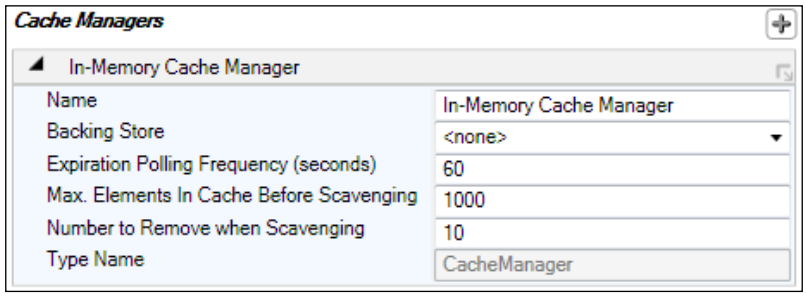
Cache manager stores the cached data in-memory and optionally it can also store the data in a configured persistent storage. The Caching block provides three backing stores out of the box and a custom backing store can be implemented using either the **IBackingStore** interface or **BaseBackingStore** class. The `IBackingStore` interface is part of the `Microsoft.Practices.EnterpriseLibrary.Caching` namespace; this interface provides the contract for backing store implementation. The `BaseBackingStore` class is part of the `Microsoft.Practices.EnterpriseLibrary.Caching.BackingStoreImplementations` namespace; this class provides implementation of common policies and utilities such as argument validations, which are useful to all backing store implementations.

The following diagram shows the members and inheritance hierarchy of the respective class and interface:



NullBackingStore is the default backing store, which is used by the Caching block while no backing store is configured. This implementation of the backing store inherits the `IBackingStore` interface but the implementation does nothing. It is surprising but the reason is pretty clear, this backing store allows the `Cache` class to store the data in-memory only. `NullBackingStore` is part of the `Microsoft.Practices.EnterpriseLibrary.Caching.BackingStoreImplementations` namespace. As discussed previously, in-memory is ideally a cache manager configuration without any backing store; in other words, a dummy implementation (`NullBackingStore`) is used as default backing store. This ideally means that the caching will be in-memory only.

The following screenshot shows the configuration options of the cache manager:



Adding items to cache

`CacheManager` provides two overloaded methods to add items to cache; the simplest overload accepts a key and a value of the cached item. This method sets the cache item priority to `Normal` and it also sets the refresh action and expiration policy to null. The other overloaded `Add` method provides finer control over the cached item. It not only allows setting the cache item priority for scavenging but it also allows setting the refresh action and multiple expiration policies. Both methods will throw an `ArgumentNullException` if the key is null or an `ArgumentException` if the key is an empty string. Apart from the above-mentioned exceptions, specific exceptions might be thrown by the configured `BackingStore` implementation.

The following is the syntax for caching an item with default settings:

```
this.cacheManager.Add(book.ID.ToString(), book);
```

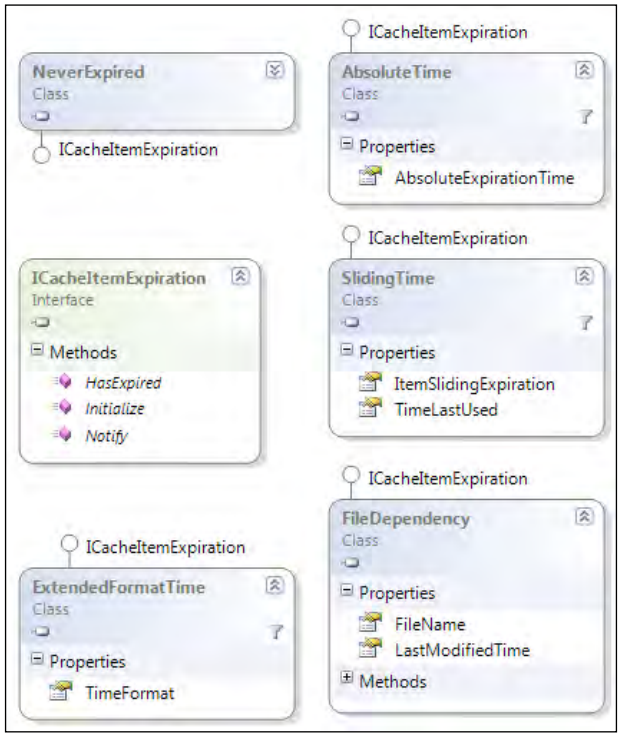
The following is the syntax for setting priority and expiration policy for the cache item:

```
//Cache Item Priority = High
//Cache Refresh Action is not set
//Cache Expiration Policy is set to sliding time of 2 minutes
this.cacheManager.Add(book.ID.ToString(), book, CacheItemPriority.
High, null, new SlidingTime(TimeSpan.FromMinutes(2)));
```

Understanding the expiration process

Cached data often needs a configurable expiration mechanism, which removes the cached item after a specified interval; this can be easily achieved by associating an expiration policy with the cached item. The Caching Application Block periodically evaluates the internal hash table to identify the expired cached items. The **BackgroundScheduler** class performs the expiration process based on the **Expiration Polling Frequency (seconds)** configured in the Cache Manager configuration settings.

The following diagram shows the members and inheritance relationship of the respective classes related to the expiration process:



Expiration policies

Expiration policy dictates when the cached item expires; we have three types of expiration policies:

- No expiration
- Time-based expiration
- Notification-based expiration

The following is the list of available expiration policies:

- **NeverExpired**: Cache item never expires but may be removed by the block if lack of memory is detected. This falls under the "No Expiration" category.
- **AbsoluteTime**: Cache item expires at a specified absolute time as specified in the **AsoluteExpirationTime** property. This falls under the "Time-based Expiration" category.
- **SlidingTime**: Cache item expires after the specified time has elapsed from when the item was last accessed. By default it is 2 minutes. This falls under the "Time-based Expiration" category.
- **ExtendedFormatTime**: Provides the ability to specify detailed expiration conditions like cache item expires every day at 5:00 PM or on Friday of each week. This falls under the "Time-based Expiration" category.
- **FileDependency**: Cache item expires when the specified file is modified. This falls under "Notification-based Expiration" category.

`NeverExpired` is the default expiration policy that will be assigned while using the given method of `CacheManager`:

```
public void Add(string key, object value);
```

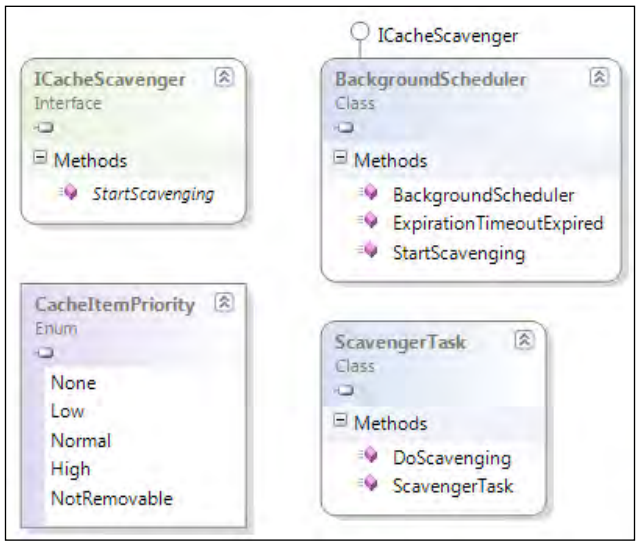
The expiration process performs marking and sweeping as a two-part process.

Understanding the Scavenging process

Every time an item is added to the cache, the `BackgroundScheduler` object checks whether the total items in the cache have reached the configured limit (**Maximum Elements in Cache before Scavenging**) provided in the **Cache Manager** configuration settings. Also, another setting, **Number to remove when scavenging**, determines the number of cached items removed from the cache after scavenging begins. Cached items are removed based on the priority (`Low`, `Normal`, `High` or `NotRemovable`) specified while adding the cached item; the default value is `Normal`.

Scavenging process performs marking and sweeping in a single pass.

The following diagram shows the members of the **Interface**, **Class**, and **Enum** related to scavenging:



Reading cached items

`CacheManager` exposes a method called `GetData`, which accepts the key of a cached item; this operation will return `null` if the cached item does not exist. It will throw an `ArgumentNullException` if the key is null or `ArgumentException` if the key is an empty string. Apart from the above-mentioned exceptions, specific exceptions might be thrown by the configured `BackingStore` implementation.

The following syntax gets the cached item using the key:

```
Book book = this.cacheManager.GetData("1") as Book;
```

Never use the `Contains` method of the cache manager as this method might not return an accurate result. The `Contains` method might return `true` indicating the cache item with the specified key exists but the `GetData` method may not fetch the item as the cached item might be expired, removed, or marked for removal.

Removing cached items

Removing cached items is a very simple affair; `CacheManager` exposes a method called `Remove`, which accepts a string representing the key of a cached item. It does nothing if no item exists with that key. It will throw an `ArgumentNullException` if the key is null or `ArgumentException` if the key is an empty string. Apart from the above-mentioned exceptions, specific exceptions might be thrown by the configured `BackingStore` implementation.

The following syntax removes the cached item with the specified key:

```
this.cacheManager.Remove("1");
```

Flushing cached items

Flushing removes all items from the cache and the cache items are left unchanged if an error is encountered during the removal process. If the `CacheManager` is configured to use either the out-of-the-box or a custom `BackingStore`, then an exception might be thrown by the configured `BackingStore` implementation.

The following syntax removes all cached items from the cache manager:

```
this.cacheManager.Flush();
```

Reloading expired items

The Caching Application Block provides extensibility points at every level; imagine a scenario where a cache item has to be reloaded as soon as it expires. The **ICacheItemRefreshAction** interface defines the contract to cater to such requirements; we can implement a custom refresh action and pass it while adding the item to the cache.

The following code snippet shows the definition of `ICacheItemRefreshAction` interface:

```
namespace Microsoft.Practices.EnterpriseLibrary.Caching
{
    public interface ICacheItemRefreshAction
    {
        void Refresh(string removedKey, object expiredValue,
CacheItemRemovedReason removalReason);
    }
}
```

The following code snippet is a sample skeleton structure to reload the expired item by implementing the `ICacheItemRefreshAction` interface:

```
[Serializable]
public class BookCacheItemRefreshAction : ICacheItemRefreshAction
{
    public void Refresh(string removedKey, object expiredValue,
CacheItemRemovedReason removalReason)
    {
        //Item removed from cache with the specified removal reason
        //Refresh the cached item
    }
}
```

The `BookCacheItemRefreshAction` class implements the `ICacheItemRefreshAction` interface. We have to provide the custom reload logic for the expired item in the `Refresh` method.

The following code snippet shows how to leverage the `BookCacheItemRefreshAction` class while adding items to cache:

```
BookCacheItemRefreshAction refreshAction = new
BookCacheItemRefreshAction();
this.cacheManager.Add(book.ID.ToString(), book, CacheItemPriority.
High, refreshAction, new SlidingTime(TimeSpan.FromMinutes(2)));
```

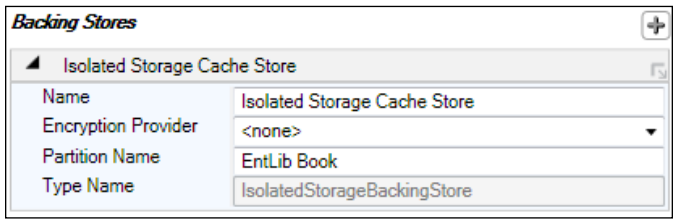
We are creating an instance of `BookCacheItemRefreshAction` and passing this object while invoking the `Add` method. Now, whenever the cached item expires the `Refresh` method will be invoked, this allows us to identify the cached item using the key. Additionally, it also provides the value of the expired item and the removal reason.

Configuring Isolated Cache Storage Backing Store

IsolatedStorageBackingStore stores cached data in a data storage mechanism called "Isolated Storage", which provides isolation and safety. `IsolatedStorageBackingStore` inherits from the `BaseBackingStore` class, which provides implementation of common policies and utilities useful to all backing store implementations. It leverages the `System.IO.IsolatedStorage.IsolatedStorageFile` class to store the cached data in a tree structured storage schema. Performance is optimized by storing the cached item in its own subdirectory and by creating separate files representing different elements of `CacheItem`. A storage encryption provider can be configured to encrypt data before storing it in persistent form. `IsolatedStorageBackingStore` is part of the `Microsoft.Practices.EnterpriseLibrary.Caching.BackingStoreImplementations` namespace.

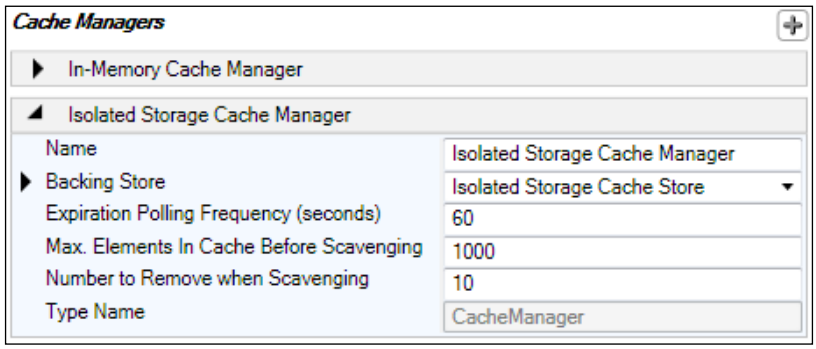
Configuration of the **Isolated Storage Cache Store** backing store is pretty straight-forward: we need to add an **Isolated Storage Cache Store** backing store with a unique **Partition Name** and then map the backing store in the Cache Manager configuration. Additionally, an encryption provider can be configured to store the cached data in encrypted form.

The following screenshot shows the configuration options of the **Isolated Storage Cache Store**.



Isolated Cache Storage stores the cached data in the user's `IsolatedStorage` folder located at `C:\Users\<<user name>>\AppData\Local\IsolatedStorage`. The partition name helps in partitioning cached data of different cache managers or even different applications.

The following screenshot shows the configuration options of the cache manager with the **Backing Store** configured as **Isolated Storage Cache Store**.

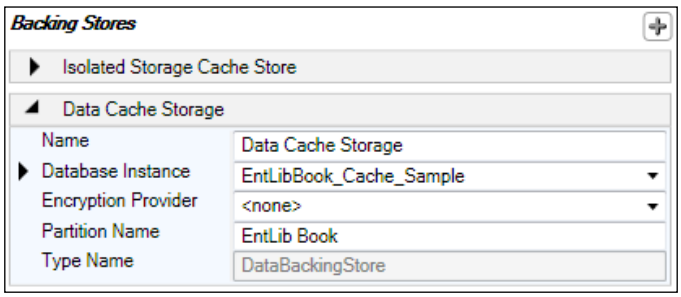


Configuring Database Cache Storage

DataBackingStore is an implementation that stores cached items in a database leveraging the **Data Access Application Block**. This application block provides the script to create the necessary database schema for SQL Server for storing cached items. A storage encryption provider can be configured to encrypt data before storing data in persistent form. `DataBackingStore` is part of the `Microsoft.Practices.EnterpriseLibrary.Caching.Database` namespace.

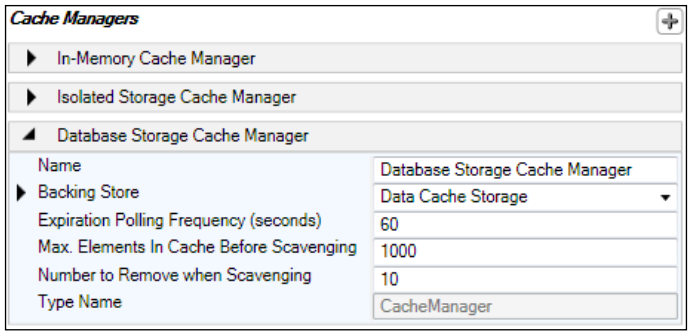
Configuration of **Data Cache Storage** backing store is similar to that for **Isolated Storage**: we need to add a **Data Cache Storage** backing store with a unique partition name and then map the backing store in the Cache Manager configuration. We also need to provide the "Connection String Key" in the **Database Instance** attribute; this connection string points to the database that contains the caching tables and stored procedures created using the `CreateCachingDatabase.sql` script located at `\Blocks\Caching\Src\Database\Scripts` folder. This script creates a database named **Caching**; we can modify the script to create the required tables, stored procedures, and so on in an existing or in a different database as well. Additionally, an encryption provider can be configured to store the cached data in encrypted form.

The following screenshot shows the configuration options of **Data Cache Storage** with the configured **Database Instance** (configuring Data Access block settings is covered in detail in the *Adding data access settings* section of *Chapter 2*, *Data Access Application Block*):



Data Cache Storage stores the cached data in the configured database. This database can be shared across different applications or different cache managers, and the partition name helps in identifying the partition to be used by the respective cache managers.

The following screenshot shows the configuration options of **Database Storage Cache Manager** with the **Backing Store** configured as **Data Cache Storage**:



Configuring and encrypting cached data

The Caching Application Block provides the ability to encrypt the cache item before the data is cached in a backing store. `SymmetricStorageEncryptionProvider` implements the `IStorageEncryptionProvider` interface, which leverages the symmetric cryptographic implementations from the Cryptography Application Block. The Cryptography block is covered in detail in *Chapter 8*, *Cryptography Application Block*. The configuration tool helps in selecting the symmetric cryptography provider, generate a key, and associate the encryption provider to the backing store. `SymmetricStorageEncryptionProvider` is part of the `Microsoft.Practices.EnterpriseLibrary.Caching.Cryptography` namespace.

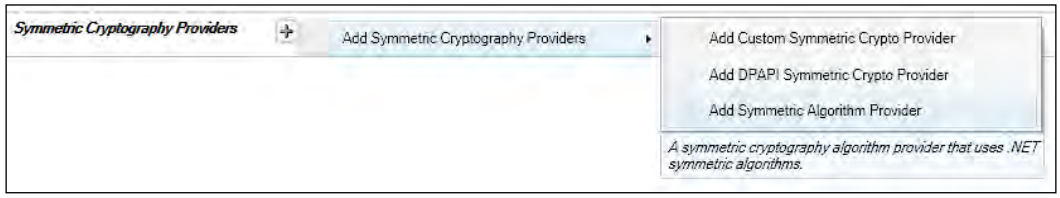
While using `NullBackingStore` to cache data in memory, the application block will not perform encryption even if encryption is configured. This behavior is intentional and so it is recommended not to store any sensitive data in the cache.

The **Caching Application Block** leverages the **Cryptography Application Block** to provide us with the encryption and decryption capabilities to securely store our cached data in a persistent backing store. It is to be noted that encryption configuration will not work with in-memory storage (`NullBackingStore`). Encryption/Decryption is a feature purely enabled through configuration, and there are no code changes required to leverage this functionality.

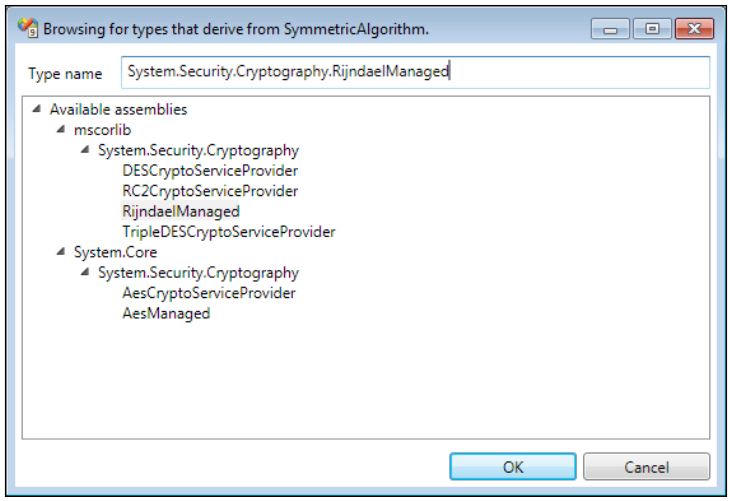
Configuration steps

1. **Caching Settings** contains a section called **Encryption Providers**. Click on the plus symbol available on the top right corner of that section and navigate and click **Add Encryption Providers | Add Symmetric Crypto Provider**. This action would add the default **Cryptography Settings** of the Cryptography Application Block.

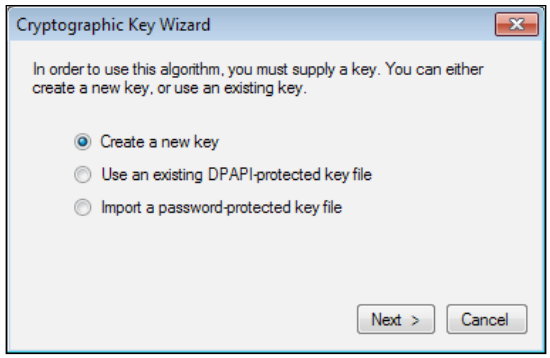
2. In the **Cryptography Settings**, click on the plus symbol of the **Symmetric Cryptography Providers** section and navigate and click **Add Symmetric Cryptography Providers | Add Symmetric Algorithm Provider**. We can also add **Data Protection API (DPAPI)** or **Custom Symmetric Crypto Provider** based on the requirements.



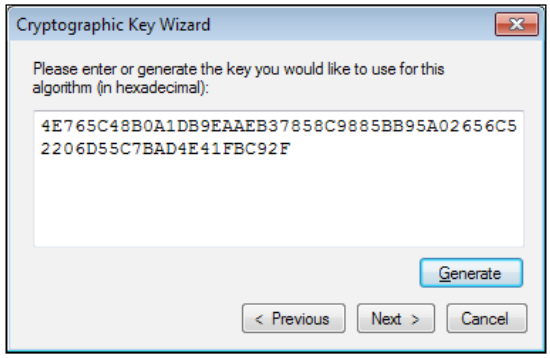
3. The previous action will show a symmetric algorithm selection dialog; for this demonstration, we will select `System.Security.Cryptography.RijndaelManaged` and hit **OK**.



4. The previous action will pop up the **Cryptographic Key Wizard** dialog; basically the algorithm requires a key that can be used to encrypt and decrypt data. We can either create a new key, use an existing **Data Protection API (DPAPI)** protected key file, or import a password-protected key file. For the purposes of this demonstration, we will opt to **Create a new key** and click **Next**.



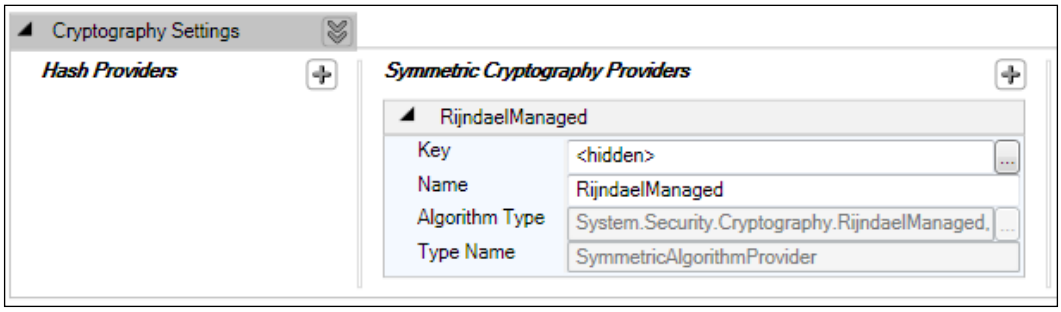
5. We are now prompted to either enter the key or generate the key using the **Generate** button. Click on **Generate** and a new key will be generated and displayed in the textbox. Click **Next** to move to the next step of the wizard.



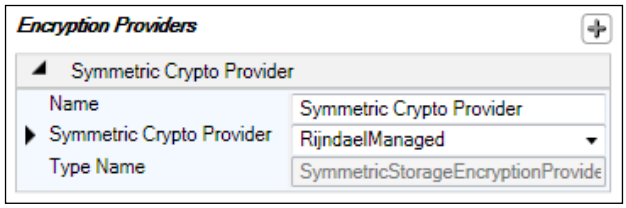
6. This step allows us to store the key in a file; provide the appropriate path and key filename by clicking the ellipsis "**...**" button. Click **Next** to move to the next step of the wizard.

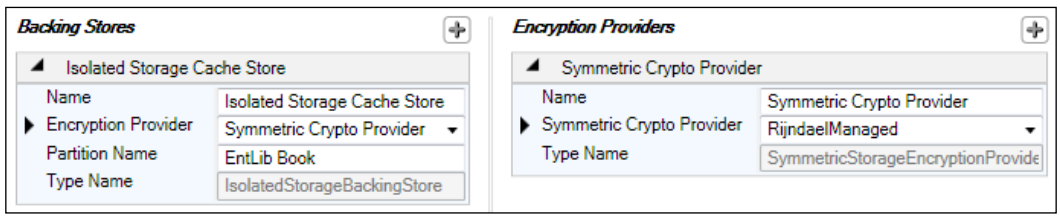
7. So far we have generated the key and specified the path and filename to store the key. But the key itself is not yet protected and vulnerable; this step prompts us to protect the key using the **Data Protection API (DPAPI)**. We have to select the data protection mode. **User mode** encrypts the key using the credentials of the currently logged-in user while the **Machine mode** allows any users on this computer to encrypt or decrypt the key. For the purposes of this demonstration, we will select **User mode**. Click the **Finish** button to close the wizard.

8. We will end up with the **Cryptography Settings** configuration as given next:



9. The next step is to associate the configured Symmetric Cryptography Provider in the Encryption Providers section of the Caching Settings. Also we have to associate the backing store with the Encryption Provider. The following screenshots depict the associations and configuration.





To summarize the configuration steps, we have configured an `IsolatedStorage` backing store with an Encryption Provider entry named **Symmetric Crypto Provider**, which leverages the Cryptography Application Block and is associated to a **Symmetric Cryptography Providers** entry named **RijndaelManaged,** with algorithm type `System.Security.Cryptography.RijndaelManaged`. While performing caching operations using the specified `IsolatedStorage` backing store, the cached data will be encrypted/decrypted using the provided encryption configuration.

Summary

In this chapter, we have learned the fundamental elements of the Caching Application Block such as the `CacheManager` class, expiration policy, scavenging process, backing stores, and encryption providers. We have explored the various required and optional assemblies, the initial infrastructure configuration and the individual feature-level configuration. We have also learned to initialize the `CacheManager` using the `CacheFactory` class, Unity service locator, and using Unity container directly and later we deep dived into the basics of adding, removing, reading, and flushing the cache items. We have further learned to configure an encryption provider to encrypt cached data while using a persistent backing store.

We often spend lot of effort on validating input and it often becomes challenging to perform the same validation across different layers of the application; in the next chapter we will explore the Validation block which makes validation a productive and easy affair.

6
Validation Application Block

While developing applications we always have to be distrustful of any input; be it from users or from other systems, it is very important to validate the input. Developers often spend the bulk of their development effort on validating input, yet we find ourselves struggling to manage the validation logic, which spreads like a plague into every nook and corner of the application code. To make our life more challenging we have to deal with several different validation mechanisms that are available for different types of applications (ASP.NET, Windows Forms, WCF, WPF); also within the same application we may have to validate input at multiple places across different layers or trust boundaries.

Started to feel dizzy? Let us do something about it.

The **Validation Application Block** is a structured, consistent, easy-to-maintain, flexible, and reusable component to perform validations. It provides commonly required **Validators** that can be leveraged to validate input. It can be used to prevent invalid input; also business rules validation can be implemented. As the approach is not focused towards any specific layer (for example the UI) the same validation rules can be used to validate the input at different layers in the application. The Validation Application Block includes adapters for technologies such as ASP.NET, Windows Forms, Windows Communication Foundation (WCF), and Windows Presentation Foundation (WPF).

The following are the key features of the Validation block:

- Validation rules are not limited to any specific layer of the application; validation logic/rules can be used to validate across different layers.
- Rule-based validation provides flexibility to validate an object for various scenarios through a **Rule Set**.
- It provides consistent and flexible validation mechanism.
- It allows validating objects using attributes.

- It allows validating values programmatically.
- It allows validating objects using self-validation.
- It allows validating objects using configuration.
- It provides several validator types for the most common validation scenarios.
- It provides adapters for integration with ASP.NET, Windows Forms, WPF, and WCF.

In this chapter, you will:

- Be introduced to the Validation Application Block
- Be introduced to concepts such as Validators, ValidatorFactory, ValidationResults, and so on
- Learn about referencing the required and optional assemblies
- Learn to set up the initial infrastructure configuration using the configuration editor
- Learn to validate objects using attributes
- Learn to validate values programmatically
- Learn to validate using self-validation
- Learn to validate objects using configuration
- Learn to integrate the Validation Application Block with Windows Forms based applications
- Learn to integrate the Validation Application Block with ASP.NET web applications
- Learn to implement a custom `Validator`

Validation Application Block features

Several Validation Application Block elements work together to fulfill the validation requirements. To start with, we have to decide on the validation method. The application block provides validation methods such as the following:

- Validating objects using attributes by decorating the properties with the required Validators
- Validating values programmatically
- Validating objects using self-validation by decorating the class with the `HasSelfValidation` attribute and providing a validation method

- Validating objects using configuration by providing type to be validated, rule set, validation targets (Properties, Fields and/or Methods), and validation rules

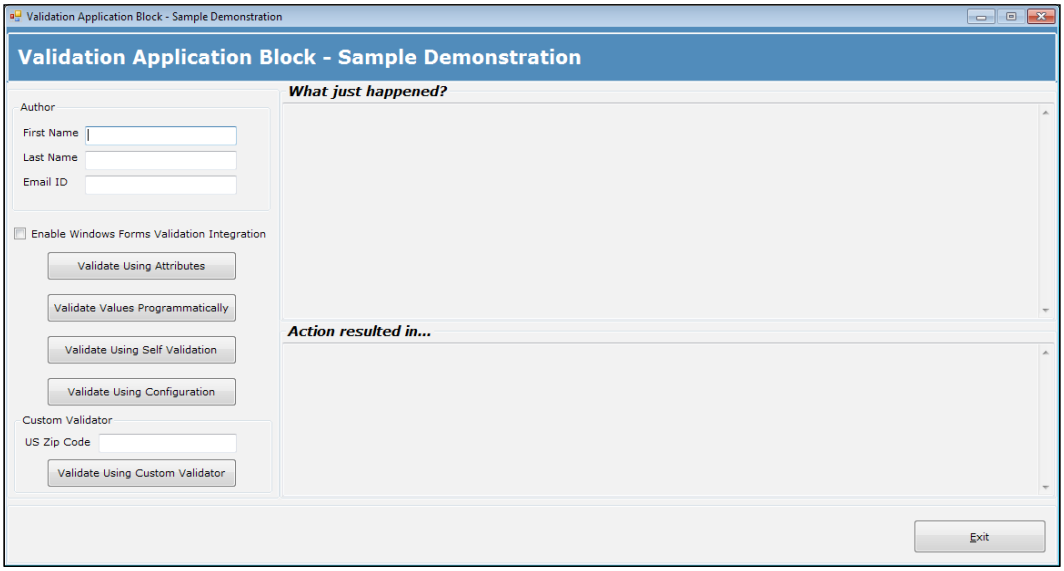
Each validation method is useful in its own way; we may opt for the validation method based on our needs. The Validation block provides several Validators (.NET classes), which can be grouped using Rule Sets, so while validating objects we may provide a Rule Set and all the validation rules are processed. These Rule Sets are mapped against properties, fields, and/or methods of the types to be validated. Now using the appropriate `ValidatorFactory` we may initiate the validation process by providing the object to be validated and the rule set to be used for validation.

Developing an application

We will explore each individual Validation block feature and along the way we will understand the concepts behind the individual elements. This will help us to get up to speed with the basics; to get started we will do the following:

- Reference the Validation block assemblies
- Add the required Namespaces

To complement the concepts and sample code of this book and allow you to gain quick hands-on experience of different features of the Validation Application Block, we have created a sample demonstration application. A screenshot of the sample application is shown as follows:



Referencing the required assemblies

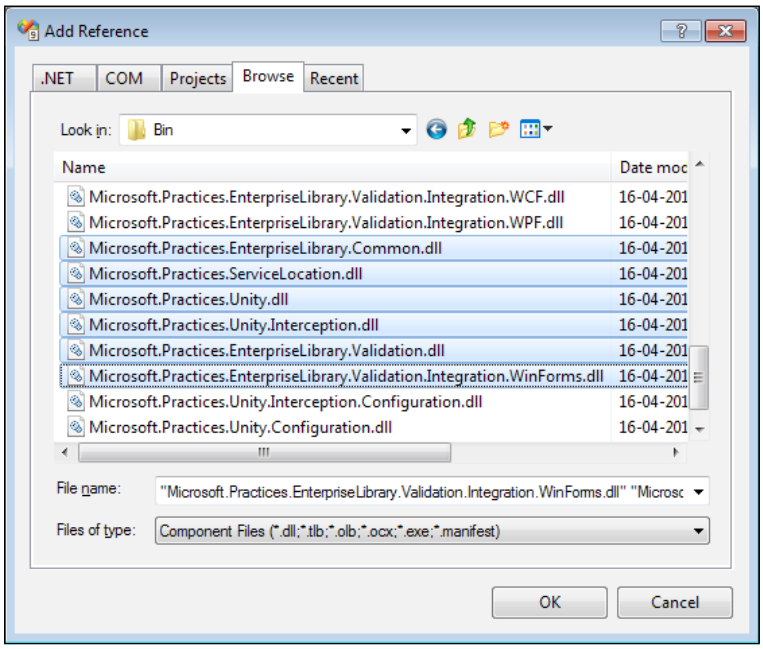
For the purposes of this demonstration we will be referencing non-strong-named assemblies but based on individual requirements Microsoft strong-named assemblies, or a modified set of custom assemblies can be referenced as well. We will also be exploring the features relating to ASP.NET in this chapter; for now, we will only include references to WinForms assemblies; adding assemblies for ASP.NET will be introduced in the *Integrating the Validation block with ASP.NET* section.

The following table lists the required/optional assemblies:

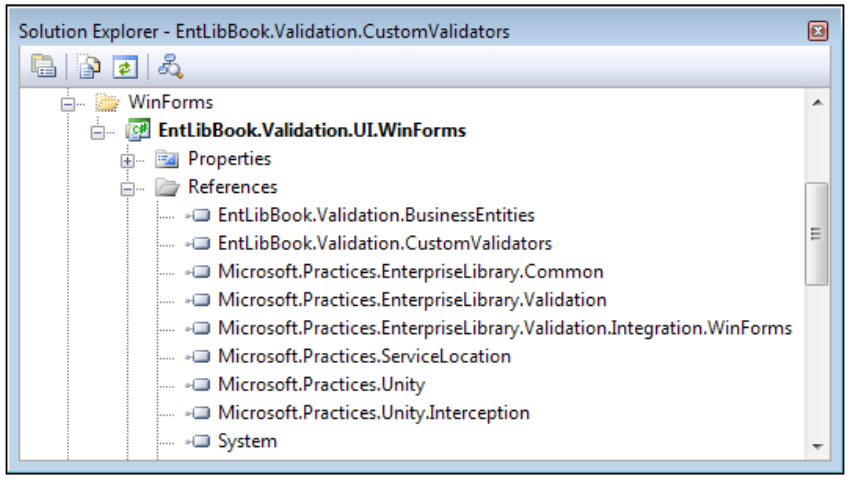
Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required
`Microsoft.Practices.Unity.Configuration.dll`	Optional Useful while utilizing Unity configuration classes in our code
`Microsoft.Practices.EnterpriseLibrary.Validation.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Validation.Integration.AspNet.dll`	Optional Used for integration with ASP.NET application
`Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms.dll`	Optional Used for integration with Windows Forms application
`Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF.dll`	Optional Used for integration with WCF service
`Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WPF.dll`	Optional Used for integration with WPF application

The following steps will add the references to the sample application:

1. Open Visual Studio 2008/2010 and create a new sample **Windows Forms Application** by selecting **File | New | Project | Windows Forms Application**, providing the appropriate name for the solution and the desired project location. Currently, the application will have a default form and assembly references. In the **Solution Explorer** right-click on the **References** section and click on **Add Reference** and go to the **Browse** tab.
2. Next, navigate to the Enterprise Library 5.0 installation location, the default install location is `%Program Files%\Microsoft Enterprise Library 5.0\Bin`.
3. Now select all the required assemblies listed in the previously given table and also the optional WinForms integration assembly. The final assembly selection will look similar to the following screenshot; note that the assemblies have been moved together for your reference.



4. After clicking the **OK** button the selected assemblies will be added to the references, the following screenshot displays the **Solution Explorer** listing all the added assemblies.



5. The next step is to add a configuration file to the project. Right-click on the project and navigate and click on the menu **Add | New Item**; this will display the **Add New Item** dialog. Select **Application Configuration File** and click on **Add**. This action will add a configuration file named `App.config` to the project. We can now add the Logging settings to the configuration file. This configuration file will be leveraged while validating using the rules configured in the configuration file.

Adding namespaces

We need to add the given namespaces to the source code file to use the Validation Application Block elements without fully qualifying each reference.

Core Namespaces:

- `Microsoft.Practices.EnterpriseLibrary.Validation`
- `Microsoft.Practices.EnterpriseLibrary.Validation.Validators`

Configuration Namespace (Optional): Required while using the **EnterpriseLibraryContainer** to instantiate objects.

- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`

Unity Namespaces (Optional): Required while instantiating objects using Unity container.

- `System.Configuration`
- `Microsoft.Practices.Unity`
- `Microsoft.Practices.Unity.Configuration`

WCF Namespace (Optional): Required while leveraging Validation Application Block in a WCF Service.

- `Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WCF`

Understanding Validators

The Validation Application Block provides several validation classes, which inherit from the abstract `Validator` class and these are called Validators. Each `Validator` class is associated with a specific data type; the `Validator` validates whether the input is valid or not. Validators can be associated with data types in several ways; association can be made through configuration, attributes, a combination of configuration, and attributes, and using self-validation. They can also be instantiated within our code without associating them with a data type. The concrete implementation of the `Validator` class holds the validation logic; the block also provides `Validator<T>`, which is a generic abstract class to validate the type represented by `T`. The Validation Application Block provides the following Validators.

Value Validators

Value Validators as the name suggests perform validations on the value of their respective data type. These are implemented using the abstract class `ValueValidator<T>`.

Validator Class	Description
`StringLengthValidator`	The `StringLengthValidator` class checks whether the length of the string is within the specified lower and upper bound range.
`ContainsCharactersValidator`	The `ContainsCharactersValidator` class checks whether an arbitrary string input contains any or all of the characters specified by the `CharacterSet` property.

Validator Class	Description
`DateTimeRangeValidator`	`DateTimeRangeValidator` class validates whether a `DateTime` object is within the specified lower and upper bound range.
`DomainValidator<T>`	`DomainValidator<T>` class checks whether the input value is one of the specified values in the set of acceptable values specified as part of the `Domain` property.
`EnumConversionValidator`	`EnumConversionValidator` class checks whether the input string can be converted to a value of the `enum` type specified in the `EnumType` property.
`NotNullValidator`	`NotNullValidator` class checks that the value is not null.
`PropertyComparisonValidator`	`PropertyComparisonValidator` class compares the value to be verified with the value of the property on the target object property (`PropertyToCompare`) using the specified comparison operator (`ComparisonOperator`).
`RangeValidator<T>`	`RangeValidator<T>` class checks whether the value is within the specified lower and upper bound range. This generic implementation can be used with any type implementing the `IComparable` interface.
`RegexValidator`	`RegexValidator` class validates whether the value matches the pattern specified by a regular expression using `System.Text.RegularExpressions.Regex`.
`RelativeDateTimeValidator`	`RelativeDateTimeValidator` class verifies whether the `DateTime` value is within the specified lower and upper bound range using relative times and dates. Additionally the `LowerUnit` and `UpperUnit` properties set the unit of time for the respective lower and upper boundaries.
`TypeConversionValidator`	`TypeConversionValidator` class validates whether the input value string can be converted to the target type specified in the `TargetType` property.

Object Validators

Object Validators performs validations on an object reference. Object Validator and Object Collection Validator fall under this category.

Validator Class	Description
`ObjectValidator`	`ObjectValidator` class invokes all Validators defined for the object's type and causes validation to occur on an object reference. Validation is ignored if the object is `null` (C#) or `Nothing` (Visual Basic).
`ObjectCollectionValidator`	`ObjectCollectionValidator` class verifies whether the object is a collection of the specified type; validation is invoked for each object in the collection using the defined Validators.

Single Member Validators

Instead of validating the entire data type using the defined Validators, Single Member Validators gives us the flexibility to validate the individual members of types. The Validation Application Block provides three different Validators: `FieldValueValidator`, `MethodReturnValueValidator`, and `PropertyValueValidator`.

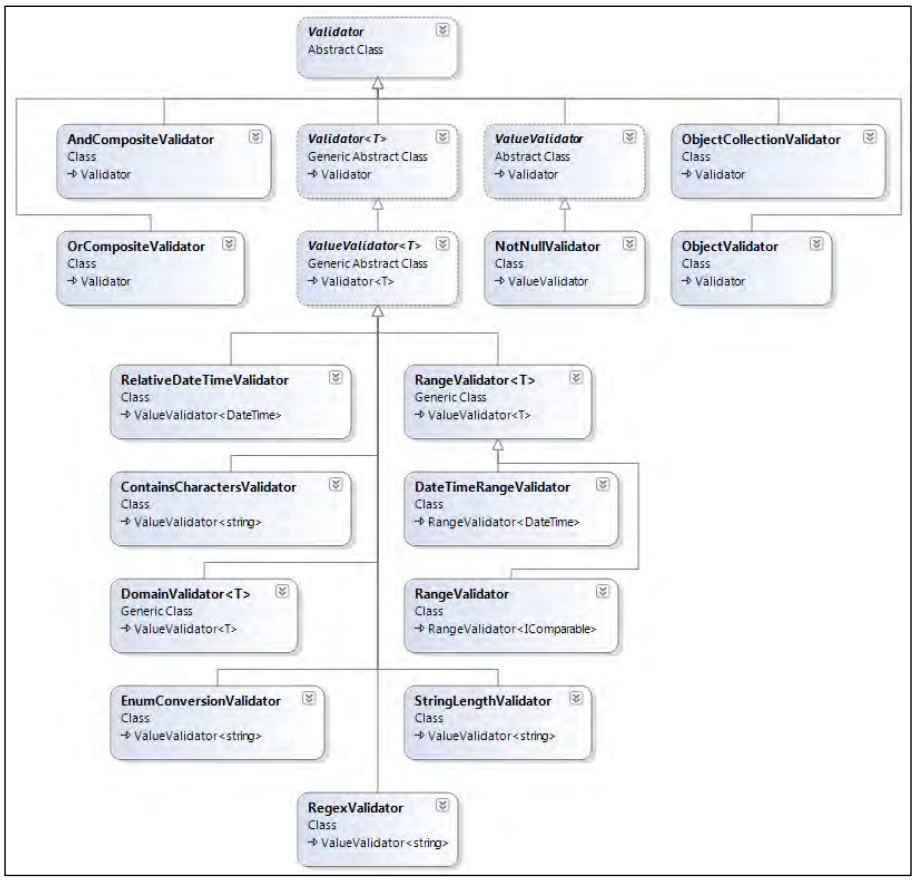
Validator Class	Description
`FieldValueValidator<T>`	`FieldValueValidator<T>` class provides the flexibility to validate a field of a type; the constructor accepts a field name and the `Validator` to validate the value of the field.
`MethodReturnValueValidator<T>`	`MethodReturnValueValidator<T>` class is similar to `FieldValueValidator`, instead of validating the field this validator accepts a method name and the `Validator` instance to validate the return value. `MethodReturnValueValidator` invokes the method and performs validation of the return value using the specified `Validator`.
`PropertyValueValidator<T>`	`PropertyValueValidator<T>` class validates the value of the specified property of a type; the constructor signature is same as for other Single Member Validators. This validator accepts a property name and the `Validator` type to be used to validate the value of the property.

Composite Validators

Composite Validators provide the flexibility to combine multiple Validators. This category consists of "And" and "Or" Composite Validators.

Validator Class	Description
`AndCompositeValidator`	The `AndCompositeValidator` class performs validation on all the specified Validators; only if all the Validators are valid will the outcome be valid. The constructor accepts a variable number of `Validator` objects as parameters.
`OrCompositeValidator`	Similar to `AndCompositeValidator` the `OrCompositeValidator` class also performs validation of all the specified Validators.

The following diagram lists the available `Validator` classes and the inheritance hierarchy:



Understanding Rule Sets

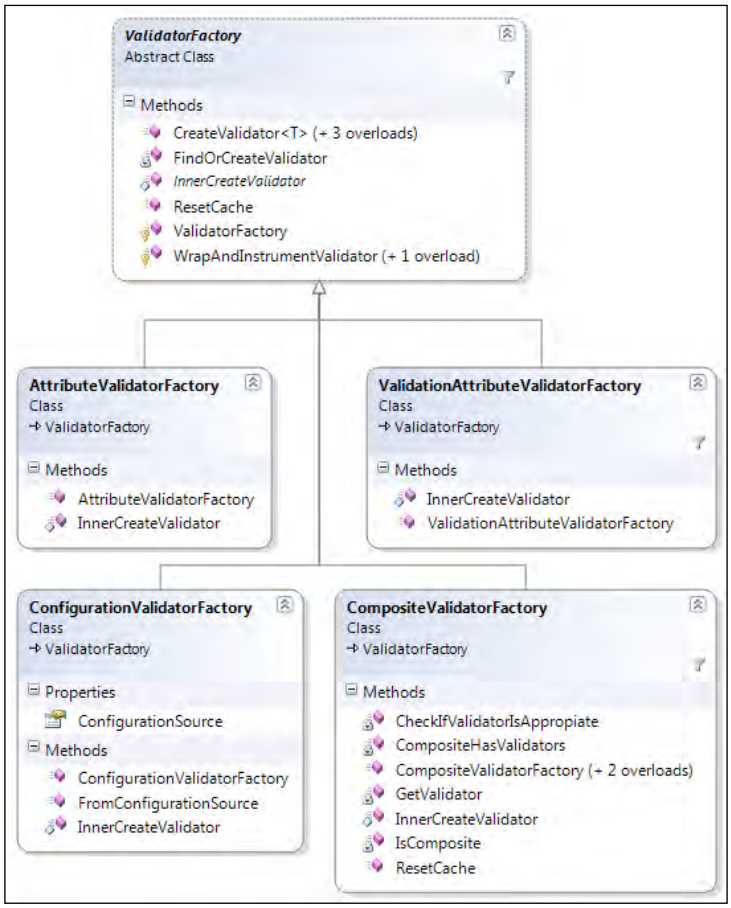
Consider Rule Sets as a way of grouping validation rules for a specific scenario. For example, while updating a `Product` record we might want to validate whether `ProductID` is available as part of the `Product` object. This rule will not apply while adding a new `Product` to the data store. Rule sets can be applied as part of attributes to properties of a class or through configuration.

Understanding ValidatorFactory

`ValidatorFactory` is an abstract class for creating Validators for a specific type; this class applies the factory pattern and helps in creating `Validator` objects using `CreateValidator` method. Since we have several validation methods, the `ValidatorFactory` class has several concrete implementations each for a specific validation method.

- The `AttributeValidatorFactory` class produces Validators based on the `Validator` attributes specified and the Rule Set in the type to be validated.
- The `ConfigurationValidatorFactory` class produces Validators based on the configuration specified and the Rule Set in the type to be validated.
- The `CompositeValidatorFactory` class composes one or more concrete implementations of ValidatorFactory classes.

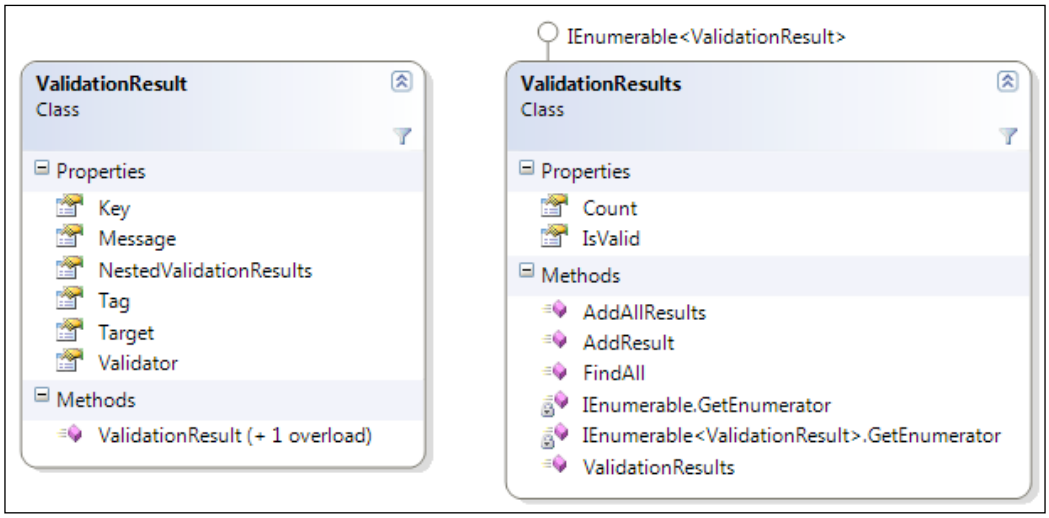
The following diagram shows the members and inheritance hierarchy of the all the `ValidatorFactory` classes:



Understanding ValidationResults

The `ValidationResults` class holds one or more `ValidationResult` objects based on the outcome of the validation. It has two useful properties; the `IsValid` property specifies whether the validation is successful and the `Count` property gets the results count. The `ValidationResult` class holds a single result with information such as validation message, key, tag, and so on.

The following diagram shows the members of the `ValidationResult` and `ValidationResults` classes:



Validating objects using attributes

Let us scratch the surface with a simple `Author` class, which consists of `ID`, `FirstName`, and `EmailID` properties. We would like to validate based on the given criteria:

- First name of the author should not be null and should be between 1 and 30 characters.
- `EmailID` should not be null or empty and should be a valid E-mail ID.

The `Author` class marked with the respective Validator attributes is given next:

```
public class Author
{
    public int ID { get; set; }

    [NotNullValidator(MessageTemplate = "First Name cannot be null")]
    [StringLengthValidator(1, 30, MessageTemplate = "First Name must
be between 1 and 30 characters")]
    public string FirstName { get; set; }

    [RegexValidator(@"\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*",
MessageTemplate = "Invalid Email ID")]
    public string EmailID { get; set; }
}
```

The given `Author` class is marked with the `Validator` attributes to let the Validation Application Block know that validation has to be done based on the given criteria. So we have specified the criteria; now we have to validate the object in our application. Assume that we receive the First Name and Email ID from the user while registering as a new author. We have to use that input and construct the object and then validate the object to verify whether the input meets the validation criteria. The following code creates a `ValidatorFactory` and creates a `Validator` instance by passing the type `Author`.

```
AttributeValidatorFactory validatoryFactory =
EnterpriseLibraryContainer.Current.GetInstance<AttributeValidatorFact
ory>();
Validator<Author> validator = validatoryFactory.
CreateValidator<Author>();

Author author = new Author();
author.FirstName = null;
author.EmailID = "some invalid email id";

ValidationResults results = validator.Validate(author);

foreach (ValidationResult result in results)
{
    Console.WriteLine(result.Message);
}
```

Since we are dealing with attribute-based validation we have created an instance of the `AttributeValidatorFactory` class. Alternatively, `ValidatorFactory` can also be instantiated to validate rules defined in attributes, configuration, and .NET Data Annotations validation attributes.

The previous code block will result in validation failure and display the following error messages in the console.

- First Name cannot be null
- First Name must be between 1 and 30 characters
- Invalid Email ID

Validating values programmatically

Validating objects using attributes works well while we own the source code for the class we wish to validate; unfortunately, there are cases where this approach will not work. We might only have the binary or the proxy of a web service and we might also wish to validate individual values instead of the entire object. To cater to these scenarios the Validation Application Block provides a `Validator` class that can be used to validate values against the specified validation criteria.

Let us assume that we have a web service proxy with `Author` class and we have to validate the "First Name" and "Email ID". The given code will validate against the same set of criteria as defined in the attribute-based validation example.

```
ValidationResults validationResults = null;

Author author = new Author();
author.FirstName = null;
author.EmailID = "some invalid email id";

Validator firstNameValidator = new AndCompositeValidator(new
NotNullValidator(), new StringLengthValidator(1, 30));
validationResults = firstNameValidator.Validate(author.FirstName);

Validator<string> emailIDValidator = new RegexValidator(@"\w+([-+.']\
w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*");
emailIDValidator.Validate(author.EmailID, validationResults);

foreach (ValidationResult result in validationResults)
{
    Console.WriteLine(result.Message);
}
```

This code block validates individual values based on the specific validation criteria provided by the `Validator` object. The `firstNameValidator` object consists of an `AndCompositeValidator`, which has `NotNullValidator` and `StringLengthValidator`; basically this says that both the validators should be `true` for a successful validation. Email ID validation is performed by instantiating `RegexValidator` (Regular Expression Validator) with the valid Email ID pattern. The `Validate` method provides an overload that accepts an existing `ValidationResults` object and adds validation errors to the list.

Validating objects using self-validation

Self-validation provides the flexibility of implementing validation logic within the class; this approach is very useful to quickly implement validation logic for complex scenarios.

The given `Author` class is marked for self-validation and provides its own validation logic:

```
[HasSelfValidation]
public class Author
{
    public int ID { get; set; }

    public string FirstName { get; set; }

    public string EmailID { get; set; }

    [SelfValidation]
    public void Validate(ValidationResults results)
    {
        if (this.FirstName == null)
            results.AddResult(new ValidationResult("First Name cannot
                                be null", this, null, null, null));
        else if((this.FirstName.Length < 1) ||
                                        (this.FirstName.Length > 30))
            results.AddResult(new ValidationResult("First Name must be
               between 1 and 30 characters", this, null, null, null));

        Validator<string> emailIDValidator = new RegexValidator
                    (@"\w+([-+.']\w+)*@\w+([-.]\w+)*\.\w+([-.]\w+)*");
        emailIDValidator.Validate(this.EmailID, results);
    }
}
```

In the given code the `Author` class is marked with the `HasSelfValidation` attribute to notify the Validation Application Block that the class implements its own validation logic. Also, we have marked the `Validate` method (the method signature should accept a single parameter of type `ValidationResults`) with the `SelfValidation` attribute. The method marked with the `SelfValidation` attribute is invoked by the Validation Application Block to validate the object.

One or more methods can be marked with the `SelfValidation` attribute for different Rule Sets.

Now when the following code is invoked, it displays all the validation errors in the console.

```
AttributeValidatorFactory validatoryFactory =
EnterpriseLibraryContainer.Current.GetInstance<AttributeValidatorFact
ory>();
Validator<Author> validator = validatoryFactory.
CreateValidator<Author>();

Author author = new Author();
author.FirstName = null;
author.EmailID = "some invalid email id";

ValidationResults results = validator.Validate(author);
foreach (ValidationResult result in results)
{
    Console.WriteLine(result.Message);
}
```

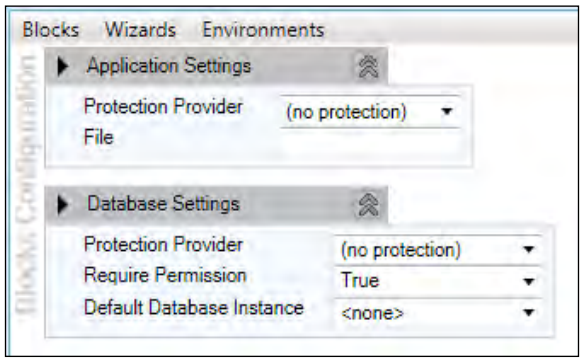
Validating objects using configuration

So far we have specified validation rules in attributes or written our own validation logic but validation rules are often dynamic in nature. Validation rules might change over a period of time and would require code changes and recompilation. Configuration-based validation provides flexibility to change validation rules without re-compiling the code. Validation rules can be configured and stored in configuration file for several types. The following steps will add the settings to the configuration file:

1. Open the **Enterprise Library configuration editor** either using the shortcut available in **Start | All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration** or just by right-clicking the configuration file in the **Solution Explorer** window of **Visual Studio IDE**.

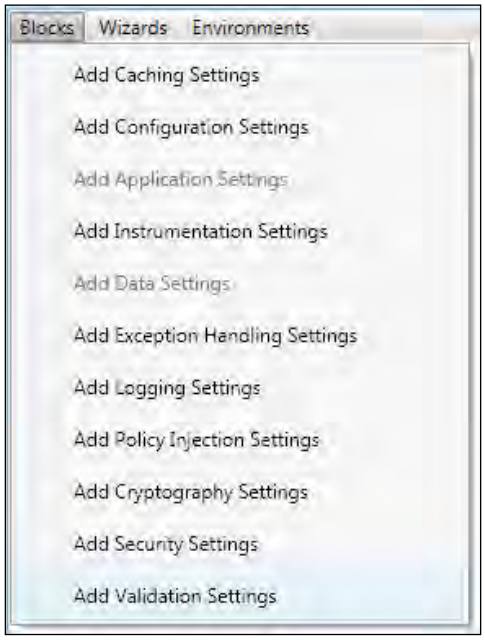
2. Next click on **Edit Enterprise Library V5 Configuration**; initially we will have a blank configuration file with default **Application Settings** and **Database Settings**.

   The following screenshot displays the default configuration:



3. Now let us add the **Validation Settings** in the configuration file. Select the menu option **Blocks**, which lists many different settings to be added to the configuration, and click on the **Add Validation Settings** menu item to add the validation configuration settings.

The following screenshot shows the menu listing several settings options:



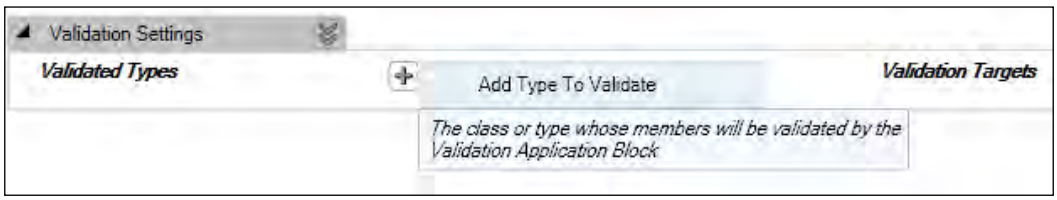
4. Once we click on the **Add Validation Settings** menu item, the **Validation Settings** section is added as shown in the given screenshot:

Notice that the setting consists of four parts: **Validated Types**, **Rule Sets**, **Validation Targets**, and **Rules**.

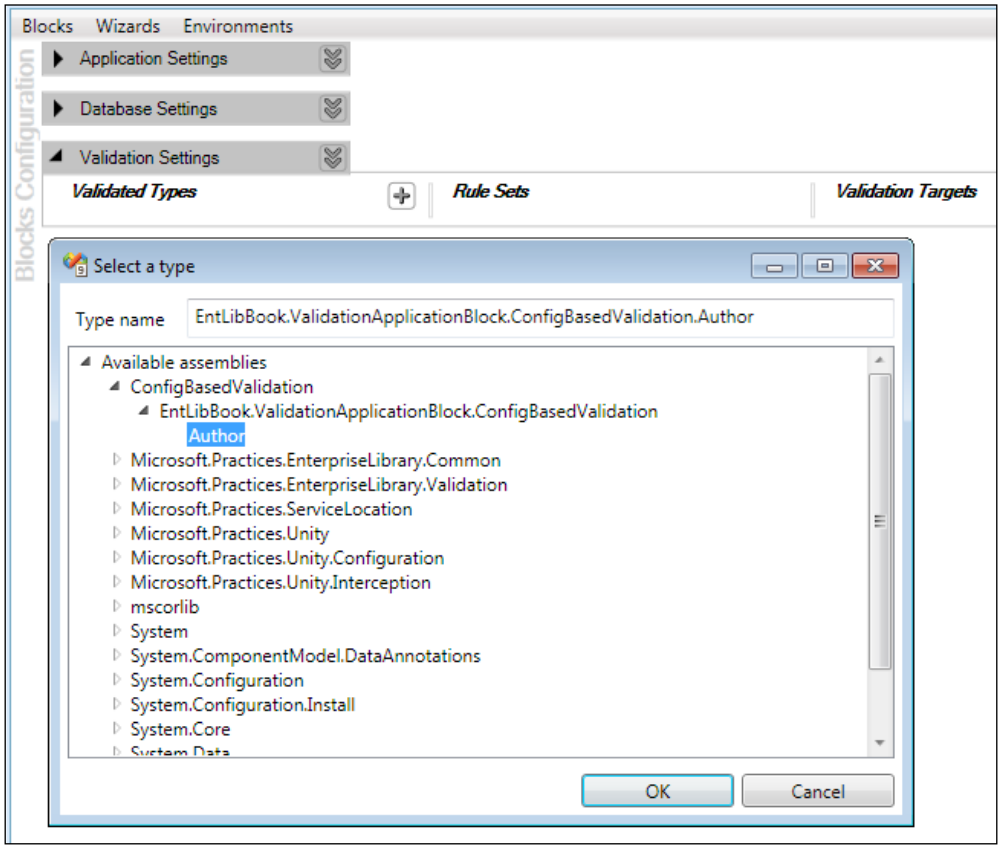
Before proceeding any further, let us look at the code of the `Author` class that will be used to demonstrate the configuration-based validation approach. The given `Author` class is similar to the `Author` class used during the attribute-based approach; we have removed all the validator attributes as these rules will now be configured in the configuration.

```
public class Author
{
    public int ID { get; set; }

    public string FirstName { get; set; }

    public string EmailID { get; set; }
}
```

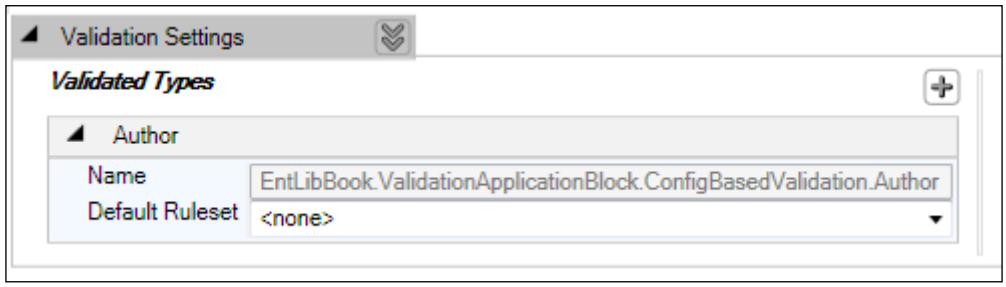
Now let us add the `Author` class whose members will be validated. Click the plus symbol in the **Validated Types** section and then click on **Add Type To Validate**. A dialog will appear with the list of available assemblies; select the `Author` class from our application assembly. The following screenshot displays the menu option to add the type to validate:



After we click the **Add Type To Validate** menu item, the type selection dialog will be shown. The following screenshot displays the type selection dialog with the selected type as `Author`.

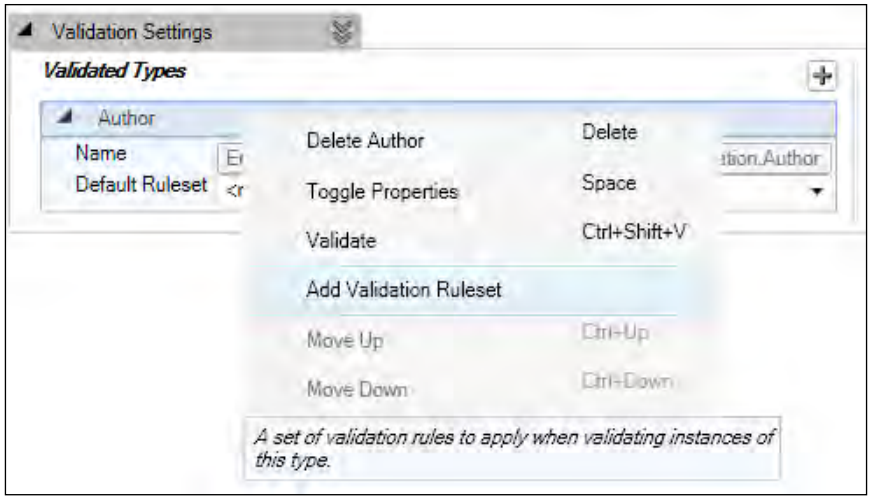


The following screenshot shows the type `Author` added in the **Validated Types** section in **Validation Settings**.

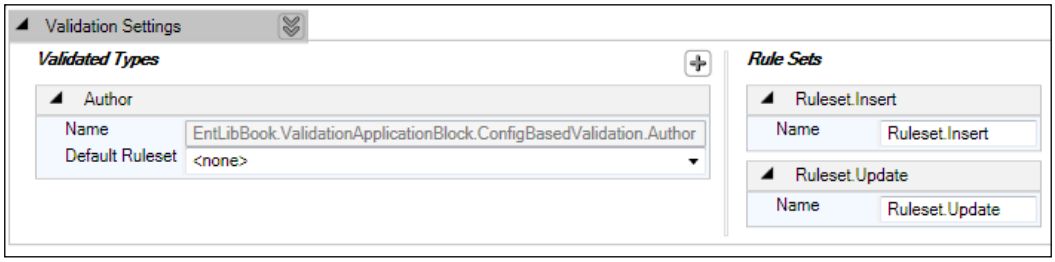


Now that we have configured the **Validated Types** section, let us move ahead with the **Rule Sets** section, which will hold the set of validation rules to be applied while validating instances of the `Author` class.

The following screenshot shows the menu option to add the validation Rule Set:

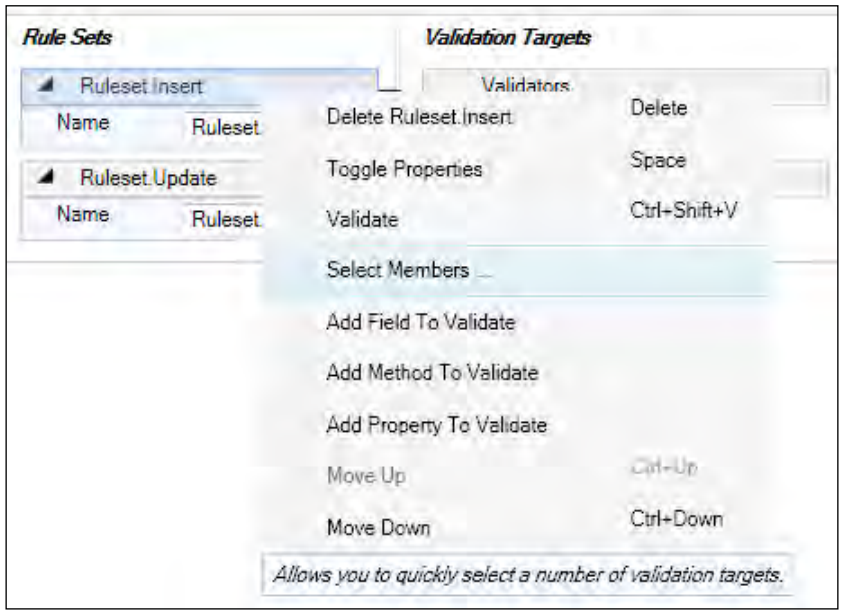


We will add two **Rule Sets** to validate the properties of the `Author` object while inserting and updating the author in the application. The following screenshot shows the configuration after adding two **Rule Sets**. Note that the `Name` has been updated to reflect the purpose of the rule set.

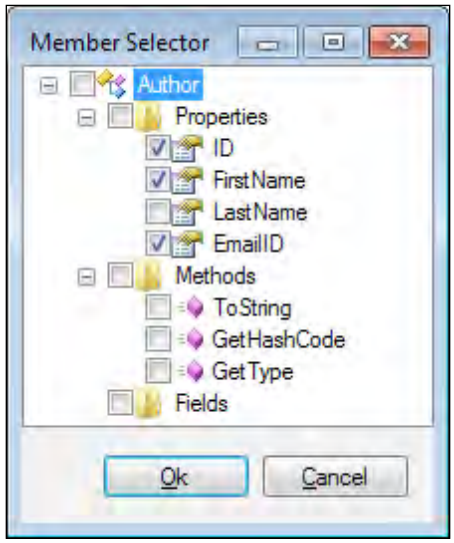


We have the Rule Sets in place; now we will add the **Validation Targets**, which are properties for both the **Rule Sets**. Right-click on each rule set and click on the **Select Members...** menu to select the members of the `Author` class that need to be validated. Alternatively, we can manually add each member using the menu options such as **Add Field To Validate**, **Add Method To Validate**, and **Add Property To Validate**.

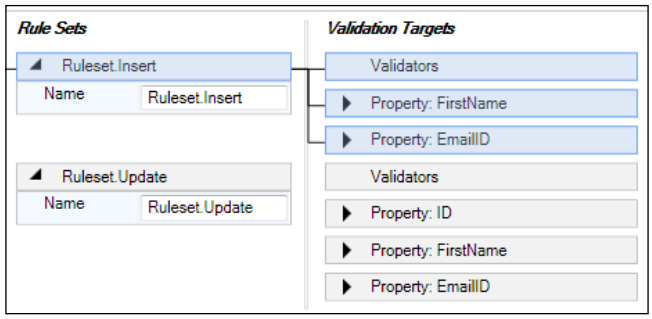
The following screenshot shows the **Select Members...** menu option:



After clicking on **Select Members...** a member selector dialog will be displayed with the list of available properties, methods, and fields. The given screenshot displays the **Member Selector** dialog for the `Author` class:

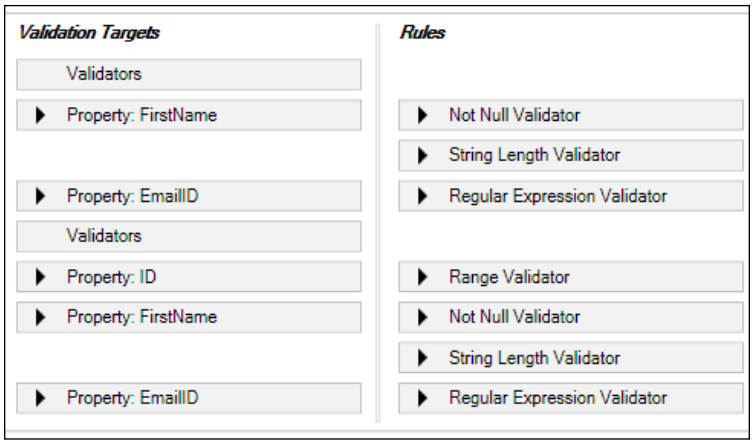


Once the members are selected, in our case we are going to select `FirstName` and `EmailID` for Rule Set **Ruleset.Insert** and `ID`, `FirstName`, and `EmailID` for Rule Set **Ruleset.Update**. The following screenshot displays the members added to the **Validation Targets** section:



Once the properties are selected for validation, we have to specify one or more validator for each member based on the validation needs. We will be using the same validators as used during the attribute-based approach; right-click on each member to navigate to **Add Validators** and to the respective validator. The given screenshot shows list of available validators for configuration:

The following screenshot shows the configured validators for each member. Several validators require setting of one or more properties and these are self explanatory as they share the same characteristic as the attribute-based approach. The following screenshot shows the configured validators for each member under the **Rules** section.



The following is a summary of the configuration steps we have performed so far:

1. We configured the `Author` class as a type whose members will be validated.
2. We configured two `Rule Sets`, one each for adding and modifying an `Author`.
3. Next, we selected the properties for both of the Rule Sets.
4. Finally, we added validation rules for the selected properties of the `Author` class.

We are done with the configuration part; now we will use the configured Rule Set to perform validation. The following code block validates the `Author` object using the specified Rule Set:

```
ConfigurationValidatorFactory validatoryFactory =
EnterpriseLibraryContainer.Current.GetInstance<ConfigurationValidator
Factory>();
Validator<Author> validator = validatoryFactory.CreateValidator<Author
>("Ruleset.Update");

Author author = new Author();
author.FirstName = null;
author.EmailID = "some invalid email id";
```

```
ValidationResults results = validator.Validate(author);

foreach (ValidationResult result in results)
{
    Console.WriteLine(result.Message);
}
```

In this code, we are instantiating a `ConfigurationValidatorFactory` object using the `EnterpriseLibraryContainer` class. Then, the next step is to create the `Validator` object; notice the highlighted code: while creating the `Validator` we can specify the Rule Set name to apply specific Rule Set-based validations. In our case, while adding the `Author` to the application we will use **Ruleset.Insert** and while modifying we will use the **Ruleset.Update**. The previous code block validates the `Author` object with **Ruleset.Update**, which validates the `ID`, `FirstName`, and `EmailID` properties based on the configured rules for each property.

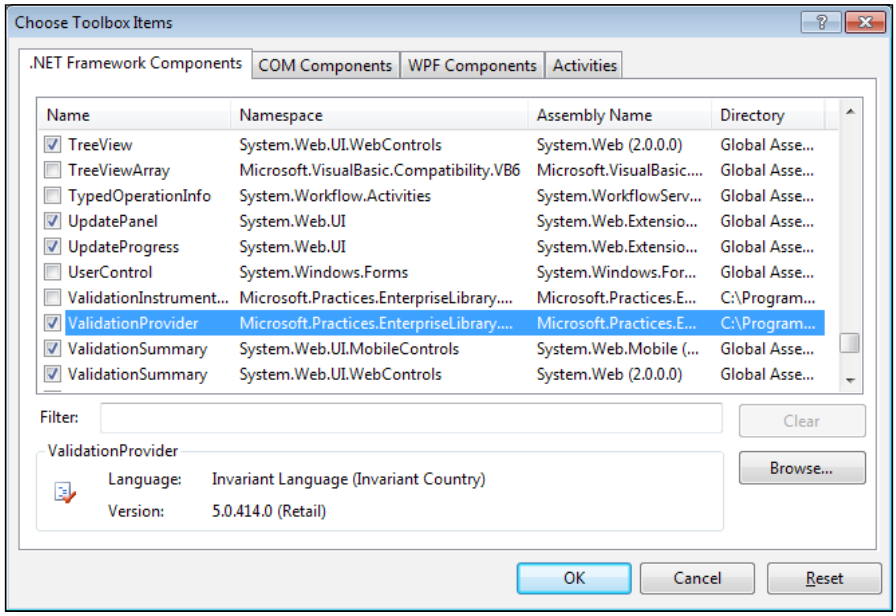
Integrating with Windows Forms-based applications

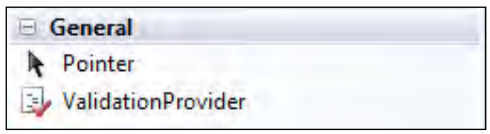
The Validation Application Block provides integration with Windows Forms applications and validates user input. The `ValidationProvider` component part of the `Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms` assembly is an extender provider that adds additional properties to Windows Forms controls. Validation can be performed by using the `Control.Validating` event or it can be invoked manually in our code using the `ValidationProvider.PerformValidation(Control)` method. Additionally, it provides integration with Windows Forms `ErrorProvider` component to display visual indication to the user of the error.

We must add reference to the `Microsoft.Practices.EnterpriseLibrary.Validation.Integration.WinForms.dll` to leverage the integration features.

Steps to leverage ValidationProvider

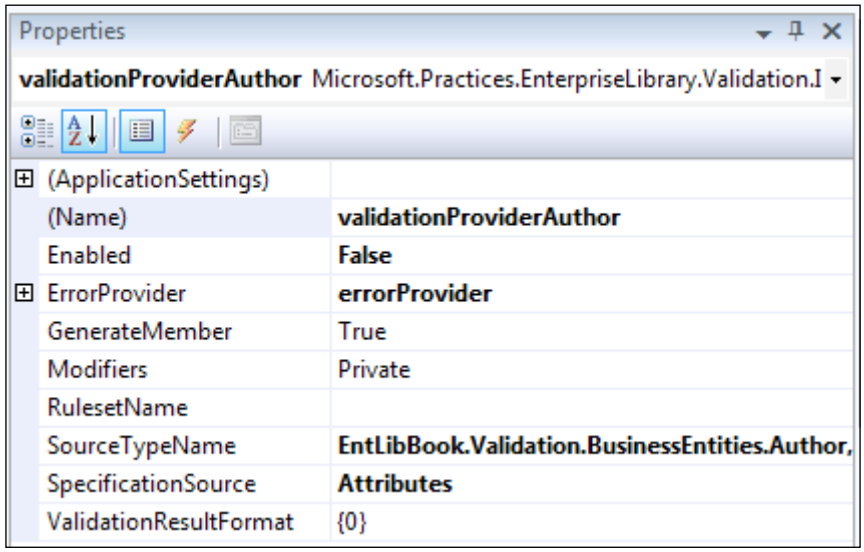
1. Add `ValidationProvider` to the Toolbox.
2. Right-click on the **Toolbox** and click **Choose Items...** menu, this will load the **Choose Toolbox Items** dialog. In the .NET Framework Components tab, select the **ValidationProvider** component.



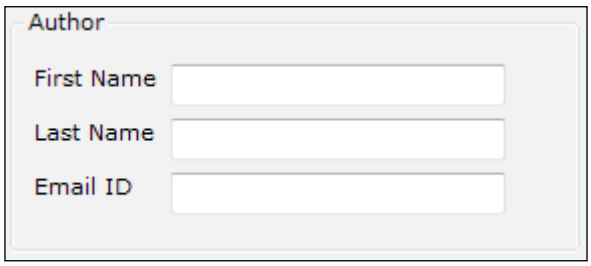


3. Add **ErrorProvider** and a `ValidationProvider` component to the Windows Forms.

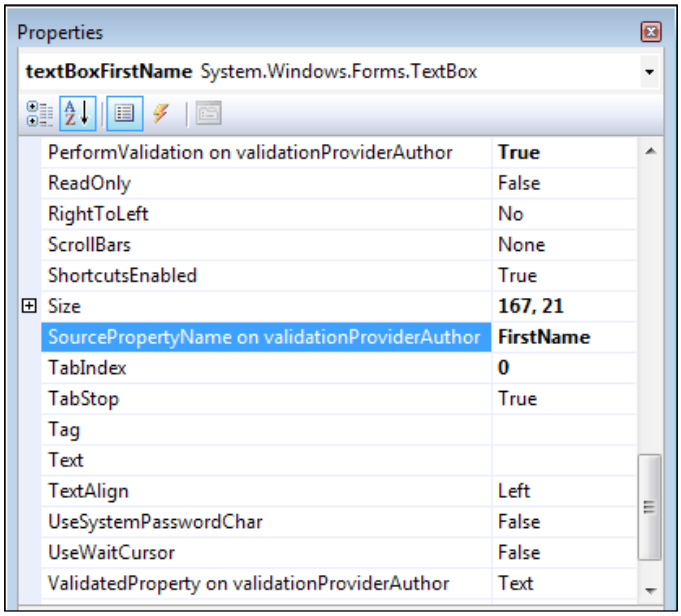
4. Configure `ValidationProvider` by selecting the `ErrorProvider` component and by assigning the `SourceTypeName` to the fully qualified name of the type to be validated. Optionally, `RulesetName` can be configured to use a specific Rule Set; also the component can be enabled or disabled by setting the `Enabled` property.



5. Configure controls for validation.
6. Assuming we want to validate the `Author` class, which consists of **First Name**, **Last Name**, and **Email ID**, since `ValidationProvider` adds additional properties to the controls, we can configure `SourcePropertyName` to the respective property name in the `Author` class. `ValidatedProperty` is set to `Text` by default for a `TextBox` control and `PerformValidation` is set to `True` by default.



7. The following screenshot shows the extended properties that have to be configured. `SourcePropertyName` determines the property to be used of the type configured in the `ValidationProvider`.

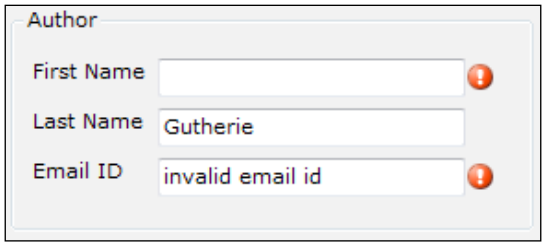


8. Perform validation either using `ValidateChildren` or the `ValidationProvider.PerformValidation(Control)` method.
9. The following code snippet shows the validation call using the Windows Forms `ValidateChildren` method:

```
this.ValidateChildren(ValidationConstraints.Visible);
```

10. The following code snippet shows the validation call using the `ValidationProvider`:

```
validationProviderAuthor.PerformValidation(textBoxFirstName);
```



Integrating the Validation block with ASP.NET

The Validation Application Block provides the `PropertyProxyValidator` control to validate user input by associating the existing validation rules of a particular type by mapping it to an **ASP.NET** server control. Apart from the common assembly references and Validation Application Block reference, we have to add the Enterprise Library Validation Application Block ASP.NET Integration assembly to leverage and integrate the Validation Application Block with ASP.NET.

We must include the integration assembly using the `@Register` directive:

```
<%@ Register Assembly="Microsoft.Practices.EnterpriseLibrary.
Validation.Integration.AspNet"
    Namespace="Microsoft.Practices.EnterpriseLibrary.Validation.
Integration.AspNet"
    TagPrefix="vabaspnet" %>
```

The `PropertyProxyValidator` control works like the **ASP.NET Validator** control but under the hood it acts as a wrapper that uses the existing validation rules. The four basic properties of this control are as follows:

- `ControlToValidate`: ID of the input control to validate
- `SourceTypeName`: Fully qualified type name whose property will be validated
- `PropertyName` : Property to be validated
- `RulesetName` : Rule Set to be applied for validation

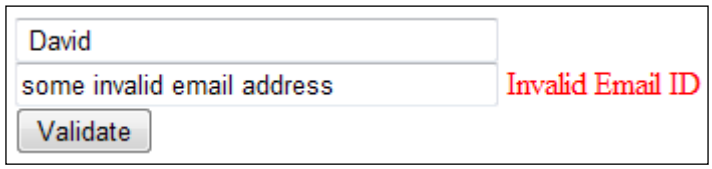
The ASP.NET syntax to associate the Server control with the `PropertyProxyValidator` control and the corresponding class and property mapping are shown next:

```
<asp:TextBox ID="txtFirstName" runat="server" Width="235px"></
asp:TextBox>
<vabaspnet:PropertyProxyValidator id="firstNameValidator"
runat="server"
        ControlToValidate="txtFirstName" PropertyName="FirstName"
        RulesetName="Ruleset.Insert" SourceTypeName="VAB_ASPNET_
Integration.Author"
        OnValueConvert="firstNameValidator_ValueConvert"></vabaspnet:P
ropertyProxyValidator>
```

This code will display the error message if the First Name does not satisfy any rules of the `FirstName` property of the `Author` class. Also, it exposes a `ValueConvert` event which can be used to convert the string representation value to the required type. The given code block converts the First Name to `null` value if the First Name is empty.

```
protected void firstNameValidator_ValueConvert(object sender,
Microsoft.Practices.EnterpriseLibrary.Validation.Integration.
ValueConvertEventArgs e)
{
    string firstName = e.ValueToConvert as string;

    if (firstName == string.Empty) e.ConvertedValue = null;
}
```

The following screenshot shows the validation result with the error message **Invalid Email ID** for the Email ID field.



Implementing a Custom Validator

The Validation Application Block provides extension points to implement custom validators; both loosely and strongly typed validators can be implemented using the abstract classes `Validator` and `Validator<T>` respectively. We may also inherit from an existing Validator class to extend the functionality. Additionally, we can also implement a custom Validator Attribute to allow our custom validator to be used with the attribute-based validation approach.

Let us implement a simple US Zip Code validator to understand the implementation details of a custom `Validator`. The steps to implement it are as follows:

1. The very first step is to add the required assembly references. We need the given assemblies for the implementation:
    - `System.Configuration.dll`
    - `Microsoft.Practices.EnterpriseLibrary.Common.dll`
    - `Microsoft.Practices.EnterpriseLibrary.Validation.dll`

2. Add a class and name the class `USZipCodeValidator`; this class will be decorated with the `ConfigurationElementType` attribute and we will use the `CustomValidatorData` as the configuration object. `CustomValidatorData` describes an instance of a custom `Validator` class.

```
[ConfigurationElementType(typeof(CustomValidatorData))]
public class USZipCodeValidator
{
}
```

3. Next, we can inherit using the strongly typed `Validator` and implement the abstract members, additionally the default message template, and the required constructors. We also have to provide our US Zip Code validation logic in the `DoValidate` method.

```
[ConfigurationElementType(typeof(CustomValidatorData))]
public class USZipCodeValidator: Validator<string>
{
    public USZipCodeValidator() : base(null, null) { }

    public USZipCodeValidator(string messageTemplate, string tag)
: base(messageTemplate, tag) { }

    protected override void DoValidate(string objectToValidate,
object currentTarget, string key, ValidationResults
validationResults)
    {
        string zipCodePattern = @"\d{5}(-\d{4})?";

        Regex regex = new Regex(zipCodePattern);

        if (!regex.IsMatch(objectToValidate))
        {
            string message = string.Format(this.MessageTemplate,
objectToValidate);
            this.LogValidationResult(validationResults, message,
currentTarget, key);
        }
    }

    protected override string DefaultMessageTemplate
    {
        get { return "Value {0} is not a valid US Zip Code"; }
    }
}
```

We have implemented our custom validator that validates a string for a valid US Zip Code. The `USZipCodeValidator` class can now be consumed either through configuration by adding the custom validator or through programmatic validation. Let us now see how we consume the Validator in our application through programmatic validation to validate the user input.

The following code snippet demonstrates the usage of the implemented custom validator, which can also be leveraged using the configuration-based approach by adding the `USZipCodeValidator` using the **Add Custom Validator** menu item while adding validators.

```
USZipCodeValidator customValidator = new USZipCodeValidator();

ValidationResults results = customValidator.Validate(textBoxUSZipCode.
Text);
```

Summary

In this chapter, we have learned about the key features and fundamental elements of the Validation Application Block such as Validators, `ValidatorFactory`, `ValidationResults`, and Rule Sets. We have explored the various required and optional assemblies, the initial infrastructure configuration, and the individual feature-level configuration. We have also learned to validate objects using various approaches such as attributes, self-validation, programmatically, and through configuration. We have also seen how the Validation Application Block can be integrated with Windows Forms-based applications and ASP.NET web applications. Finally, we learned to implement a custom validator with a simple implementation of a US Zip Code Validator. In the next chapter, we will deep dive into the **Security Application Block** and learn to leverage **Authorization Rule Provider** and **Security Cache Provider** to authorize and cache security credentials.

7
Security Application Block

Security is of prime importance for any application, especially enterprise applications where the business impact is potentially high. At the very core, security is a two step mechanism. The first step is the process of validating an identity against a store (Active Directory, Database, and so on); this is commonly called as **Authentication**. The second step is the process of verifying whether the validated identity is allowed to perform certain actions; this is commonly known **Authorization**. These two security mechanisms take care of allowing only known identities to access the application and perform their respective actions. Although, with the advent of new tools and technologies, it is not difficult to safeguard the application, utilizing these authentication and authorization mechanisms and implementing security correctly across different types of applications, or across different layers and in a consistent manner is pretty challenging for developers. Also, while security is an important factor, it's of no use if the application's performance is dismal. So, a good design should also consider performance and cache the outcome of authentication and authorization for repeated use.

The Security Application Block provides a very simple and consistent way to implement authorization and credential caching functionality in our applications. Authorization doesn't belong to one particular layer; it is a best practice to authorize user action not only in the UI layer but also in the business logic layer. As Enterprise Library application blocks are layer-agnostic, we can leverage the same authorization rules and expect the same outcome across different layers bringing consistency. Authorization of user actions can be performed using an Authorization Provider; the block provides **Authorization Rule Provider** or **AzMan Authorization Provider**; it also provides the flexibility of implementing a custom authorization provider. Caching of security credentials is provided by the **SecurityCacheProvider** by leveraging the **Caching Application Block** and a custom caching provider can also be implemented using extension points. Both Authorization and Security cache providers are configured in the configuration file; this allows changing of provider any time without re-compilation.

The following are the key features of the Security block:

- The Security Application Block provides a simple and consistent API to implement authorization.
- It abstracts the application code from security providers through configuration.
- It provides the Authorization Rule Provider to store rules in a configuration file and **Windows Authorization Manager** (**AzMan**) **Authorization Provider** to authorize against Active Directory, XML file, or database.
- Flexibility to implement custom Authorization Providers.
- It provides token generation and caching of authenticated **IIdentity**, **IPrincipal** and **Profile** objects.
- It provides User identity cache management, which improves performance while repeatedly authenticating users using cached security credentials.
- Flexibility to extend and implement custom Security Cache Providers.

In this chapter, you will:

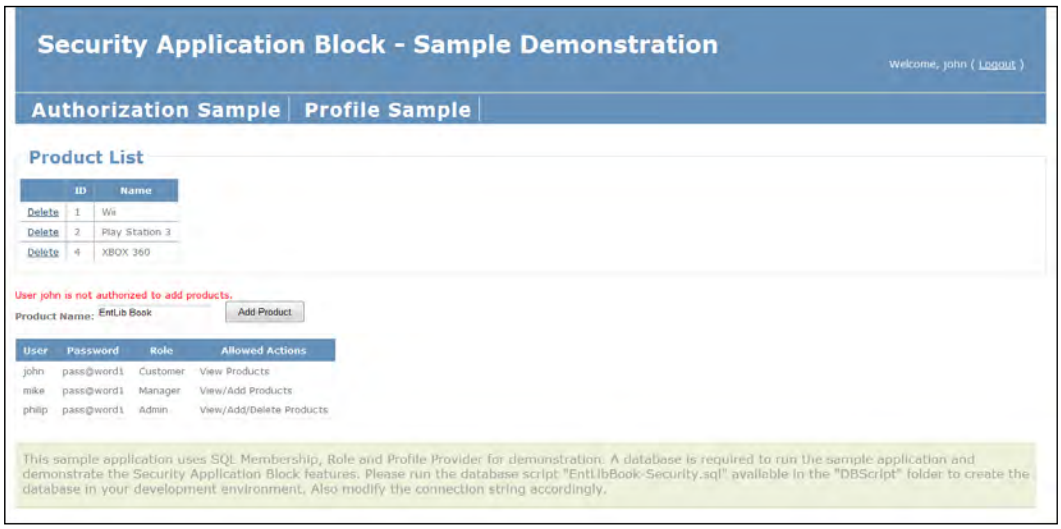
- Be introduced to the Security Application Block
- Be introduced to Authorization Providers such as Authorization Rule Provider and AzMan Authorization Provider
- Be introduced to the Security Cache Provider
- Learn about referencing the required assemblies
- Learn about the required and optional namespaces to avoid fully qualifying types
- Learn to authorize user actions based on rules
- Learn to save user Identity in cache and obtain a temporary token for an Authenticated User
- Learn to retrieve a token from cache and authenticate user
- Learn to terminate a User session by expiring cached identity
- Learn to implement a custom authorization provider

Developing an application

We will explore each individual Security block feature and along the way we will understand the concepts behind the individual elements. This will help us to get up to speed with the basics. To get started, we will do the following:

- Reference the Validation block assemblies
- Add the required Namespaces
- Set up the initial configuration

To complement the concepts and sample code of this book and allow you to gain quick hands-on experience of different features of the Security Application Block, we have created a sample web application project with three additional projects, `DataProvider`, `BusinessLayer`, and `BusinessEntities`, to demonstrate the features. The application leverages SQL Membership, Role, and Profile provider for authentication, role management, and profiling needs. Before running the web application you will have to run the database generation script provided in the **DBScript** folder of the solution, and update the connection string in `web.config` appropriately. You might have to open the solution in "Administrator" mode based on your development environment. Also, create an application pool with an identity that has the required privileges to access the development SQL Server database, and map the application pool to the website. A screenshot of the sample application is shown as follows:



Referencing required/optional assemblies

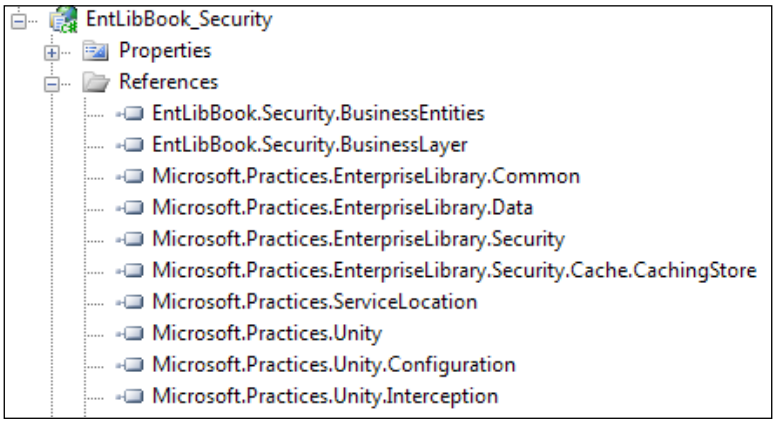
For the purposes of this demonstration we will be referencing non-strong-named assemblies but based on individual requirements Microsoft strong-named assemblies, or a modified set of custom assemblies can be referenced as well. The list of Enterprise Library assemblies that are required to leverage the Security Application Block functionality is given next. A few assemblies are optional based on the Authorization Provider and cache storage mechanism used. Use the Microsoft strong-named, or the non-strong-named, or a modified set of custom assemblies based on your referencing needs.

The following table lists the required/optional assemblies:

Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required
`Microsoft.Practices.Unity.Configuration.dll`	Optional Useful while utilizing Unity configuration classes in our code
`Microsoft.Practices.EnterpriseLibrary.Security.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Security.AzMan.dll`	**Optional** Used for Windows Authorization Manager Provider
`Microsoft.Practices.EnterpriseLibrary.Security.Cache.CachingStore.dll`	**Optional** Used for caching the User identity
`Microsoft.Practices.EnterpriseLibrary.Data.dll`	**Optional** Used for caching in Database Cache Storage

Open Visual Studio 2008/2010 and create a new ASP.NET Web Application Project by selecting **File | New | Project | ASP.NET Web Application**; provide the appropriate name for the solution and the desired project location. Currently, the application will have a default web form and assembly references. In the **Solution Explorer**, right-click on the **References** section and click on **Add Reference** and go to the **Browse** tab. Next, navigate to the Enterprise Library 5.0 installation location; the default install location is `%Program Files%\Microsoft Enterprise Library 5.0\Bin`. Now select all the assemblies listed in the previous table, excluding the AzMan-related assembly (`Microsoft.Practices.EnterpriseLibrary.Security.AzMan.dll`).

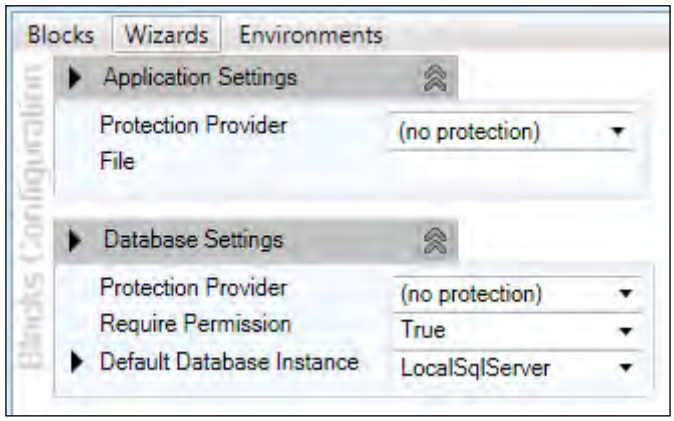
The final assembly selection will look similar to the following screenshot:



Adding initial security settings

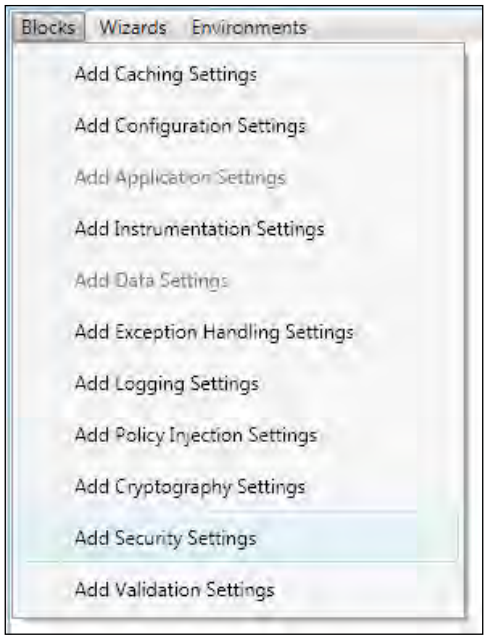
Before we can leverage the features of the Security Application Block, we have to add the initial **Security Settings** to the configuration. Open the Enterprise Library configuration editor either using the shortcut available in **Start | All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration** or just by right-clicking the configuration file in the **Solution Explorer** window of **Visual Studio IDE** and clicking on **Edit Enterprise Library V5 Configuration**. Initially, we will have a blank configuration file with default **Application Settings** and **Database Settings**.

The following screenshot shows the default configuration settings:

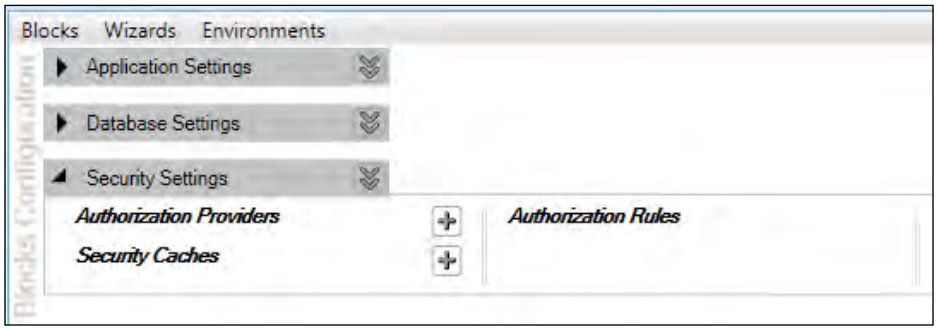


Let us go ahead and add the **Security Settings** in the configuration file. Select the menu option **Blocks**, which lists many different settings to be added to the configuration, and click on the **Add Security Settings** menu item to add the security configuration settings.

The following screenshot shows the available options in the **Blocks** menu:



Once we click on the **Add Security Settings** the configuration editor will display the default **Security Settings** as shown in the following screenshot.



The **Security Settings** consist of **Authorization Providers**, **Security Caches** and **Authorization Rules**. Authorization Rules can be configured only while using Authorization Rule Provider. We will change the configuration further but for now, we are in good shape with regards to the initial infrastructure configuration.

Adding namespaces

We will be leveraging types from several different namespaces and so to make our life easy we can add the given namespace to the source code file to use the Security block elements without fully qualifying the references. Although we will be using `EnterpriseLibraryContainer` to instantiate objects (we will also add `Microsoft.Practices.EnterpriseLibrary.Common.Configuration` namespace to the source file), the Unity Namespace section is listed to make you aware of the availability of the alternative approach of instantiating objects.

- **Core Namespace:**
  - `Microsoft.Practices.EnterpriseLibrary.Security`
- **Configuration Namespace (Optional):** Required while using the **EnterpriseLibraryContainer** to instantiate objects.
  - `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`
- **Unity Namespace (Optional):** Required while instantiating objects using **UnityContainer**.
  - `System.Configuration`
  - `Microsoft.Practices.Unity`
  - `Microsoft.Practices.Unity.Configuration`

Creating security application block objects

We have several options at hand while creating objects, such as using a static factory class, using Unity service locator and using Unity container directly. A few approaches such as configuring the container through a configuration file or code are not listed here but the recommended approach is either to use the Unity Service Locator for applications with few dependencies or create objects using Unity container directly to leverage the benefits of this approach. Use of a static factory class is not recommended.

Using the static factory class

Static factory classes were the default approach for creating objects with versions prior to 5.0. This approach is no longer recommended but is still available for backward compatibility.

The following is the syntax to create deafult and named Authorization Provider instances using the static `AuthorizationFactory` class:

```
//Instantiating Using Static Factory - Default Authorization Provider
IAuthorizationProvider defaultAuthorizationProvider =
AuthorizationFactory.GetAuthorizationProvider();

//Instantiating Using Static Factory - Named Authorization Provider
IAuthorizationProvider namedAuthorizationProvider =
AuthorizationFactory.GetAuthorizationProvider("AuthzProvider");
```

The following is the syntax to create deafult and named Security Cache Provider instances using the static `SecurityCacheFactory` class:

```
//Instantiating Using Static Factory - Default Security Cache Provider
ISecurityCacheProvider defaultSecurityCacheProvider =
SecurityCacheFactory.GetSecurityCacheProvider();

//Instantiating Using Static Factory - Named Security Cache Provider
ISecurityCacheProvider namedSecurityCacheProvider =
SecurityCacheFactory.GetSecurityCacheProvider("SecurityCache");
```

Using Unity service locator

This approach is recommended for applications with few dependencies. The `EnterpriseLibraryContainer` class exposes a static property called **Current** of type `IServiceLocator`, which resolves and gets an instance of the specified type.

The following is the syntax to create default and named Authorization Provider instances using the `EnterpriseLibraryContainer` class:

```
//Instantiating Using Unity Service Locator - Default Authorization
Provider
IAuthorizationProvider defaultAuthorizationProvider =
EnterpriseLibraryContainer.Current.GetInstance<IAuthorizationProvid
er>();

//Instantiating Using Unity Service Locator - Named Authorization
Provider
IAuthorizationProvider namedAuthorizationProvider =
EnterpriseLibraryContainer.Current.GetInstance<IAuthorizationProvider>
("AuthzProvider");
```

The following is the syntax to create default and named Security Cache Provider instances using the `EnterpriseLibraryContainer` class:

```
//Instantiating Using Unity Service Locator - Default Security Cache
Provider
ISecurityCacheProvider defaultSecurityCacheProvider =
EnterpriseLibraryContainer.Current.GetInstance<ISecurityCacheProvid
er>();

//Instantiating Using Unity Service Locator - Named Security Cache
Provider
ISecurityCacheProvider namedSecurityCacheProvider =
EnterpriseLibraryContainer.Current.GetInstance<ISecurityCacheProvider>
("SecurityCache");
```

Using Unity container directly

Larger complex applications demand looser coupling; this approach leverages the dependency injection mechanism to create objects instead of explicitly creating instances of concrete implementations. Unity container stores the type registrations and mappings in the configuration file and instantiates the appropriate type whenever requested. This allows us to change the type in the configuration without re-compiling the code and essentially to change the behavior from outside.

The following is the syntax to create default and named Authorization Provider instances using the `UnityContainer` class:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();

//Instantiating Using Unity Container Directly - Default Authorization
Provider
```

```
IAuthorizationProvider defaultAuthorizationProvider = container.Resolv
e<IAuthorizationProvider>();

//Instantiating Using Unity Container Directly - Named Authorization
Provider
IAuthorizationProvider namedAuthorizationProvider = container.Resolve<
IAuthorizationProvider>("AuthzProvider");
```

The following is the syntax to create default and named Security Cache Provider instances using the `UnityContainer` class:

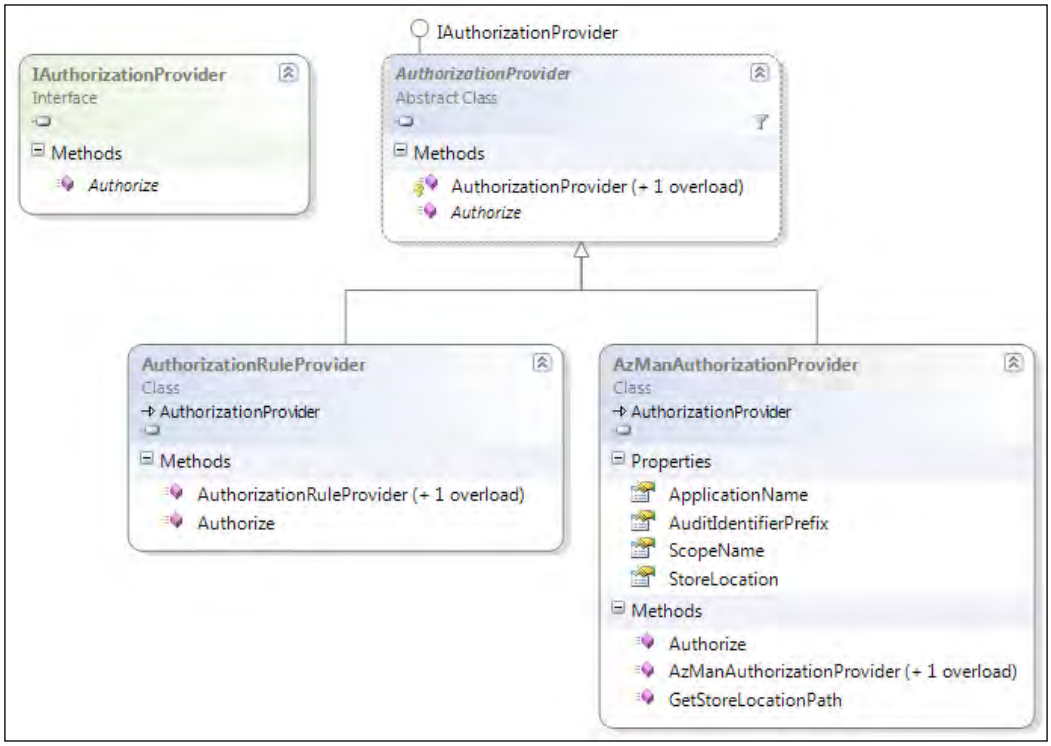
```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();

//Instantiating Using Unity Container Directly - Default Security
Cache Provider
ISecurityCacheProvider defaultSecurityCacheProvider = container.Resolv
e<ISecurityCacheProvider>();

//Instantiating Using Unity Container Directly - Named Security Cache
Provider
ISecurityCacheProvider namedSecurityCacheProvider = container.Resolve<
ISecurityCacheProvider>("SecurityCache");
```

Understanding Authorization Providers

An Authorization Provider is simply a class that provides authorization logic; technically it implements either an `IAuthorizationProvider` interface or an abstract class named `AuthorizationProvider` and provides authorization logic in the `Authorize` method. As mentioned previously, the Security Application Block provides two Authorization Providers out of the box, `AuthorizationRuleProvider` and `AzManAuthorizationProvider` both implementing the abstract class `AuthorizationProvider` available in the `Microsoft.Practices.EnterpriseLibrary.Security` namespace. This abstract class in turn implements the `IAuthorizationProvider` interface, which defines the basic functionality of an Authorization Provider; it exposes a single method named `Authorize`, which accepts an instance of the `IPrincipal` object and the name of the rule to evaluate. Custom providers can be implemented either by implementing the `IAuthorizationProvider` interface or an abstract class named `AuthorizationProvider`.

An `IPrincipal` instance (`GenericPrincipal`, `WindowsPrincipal`, `PassportPrincipal`, and so on) represents the security context of the user on whose behalf the code is running; it also includes the user's identity represented as an instance of `IIdentity` (`GenericIdentity`, `FormsIdentity`, `WindowsIdentity`, `PassportIdentity`, and so on).

The following diagram shows the members and inheritance hierarchy of the respective class and interface:



Authorization Rule Provider

The `AuthorizationRuleProvider` class is an implementation that evaluates Boolean expressions to determine whether the objects are authorized; these expressions or rules are stored in the configuration file. We can create authorization rules using the **Rule Expression Editor** part of the Enterprise Library configuration tool and validate them using the `Authorize` method of the `Authorization Provider`. This authorization provider is part of the `Microsoft.Practices.EnterpriseLibrary.Security` namespace.

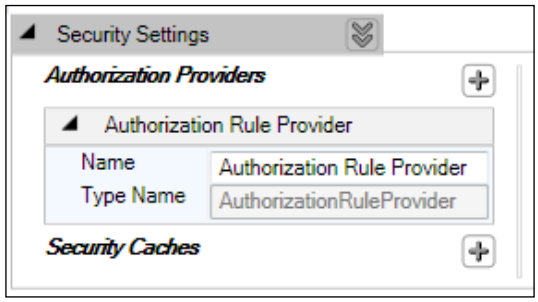
Authorizing using Authorization Rule Provider

As discussed earlier, Authorization Rule Provider stores authorization rules in the configuration and this is one of the simplest ways to perform authorization. Basically, we need to configure to use the Authorization Rule Provider and provide authorization rules based on which the authorization will be performed.

Let us add Authorization Rule Provider as our Authorization Provider; click on the plus symbol on the right side of the Authorization Providers and navigate to the **Add Authorization Rule Provider** menu item.

The following screenshot shows the configuration options of the **Add Authorization Rule Provider** menu item:

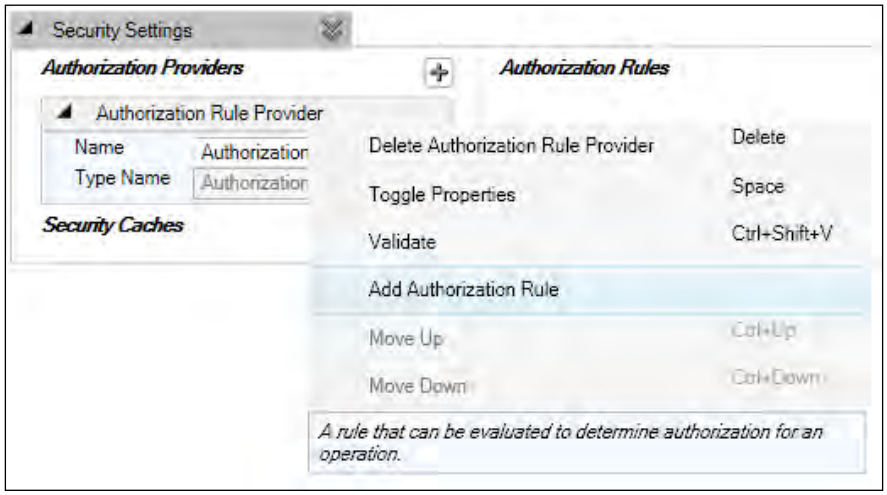
The following screenshot shows the default configuration of the newly added Authorization Provider; in this case, it is **Authorization Rule Provider**:



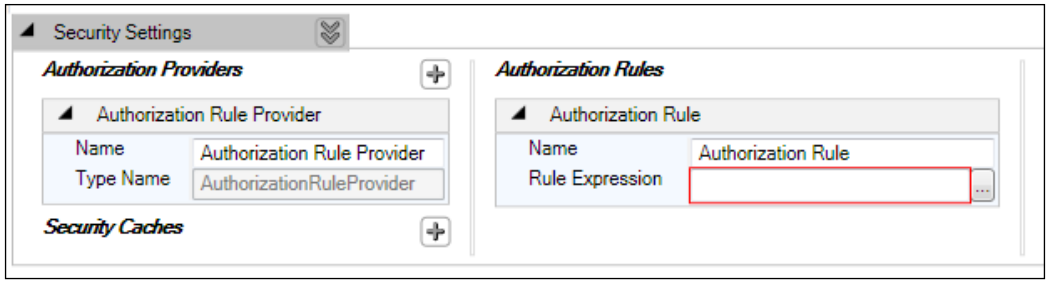
Now we have the Authorization Rule Provider added to the configuration but we still need to add the authorization rules. Imagine that we have a business scenario where:

- We have to allow only users belonging to the administrator's role to add or delete products.
- We should allow all authenticated customers to view the products.

This scenario is quite common where certain operations can be performed only by specific roles, basically role-based authorization. To fulfill this requirement, we will have to add three different rules for add, delete, and view operations. Right-click on the Authorization Rule Provider and click on the **Add Authorization Rule** menu item as shown on the following screenshot.



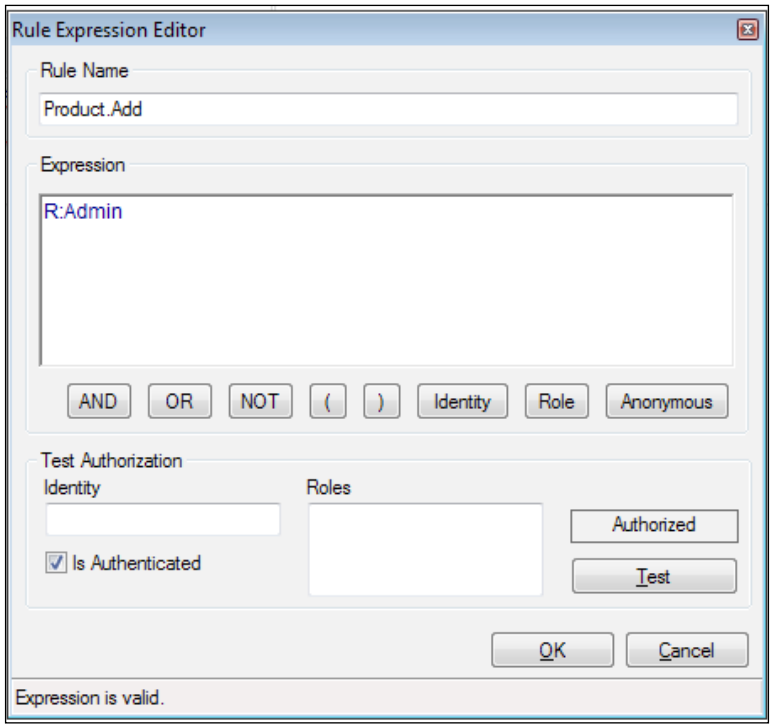
The following screenshot shows the newly added Authorization Rule:



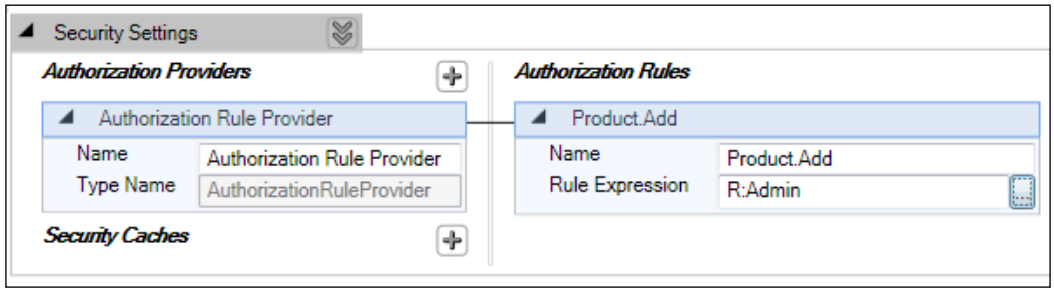
Let us update the name of the rule to "**Product.Add**" to represent the operation for which the rule is configured. We will provide the rule using the **Rule Expression Editor**; click on the right corner button to open the Rule Expression Editor. The requirement is to allow only the administrator role to perform this action. The following action needs to be performed to configure the rule:

1. Click on the **Role** button to add the Role expression: **R**.
2. Enter the role name next to the role expression: **R:Admin**.
3. Select the checkbox Is **Authenticated** to allow only authenticated users.

The following screenshot displays the **Rule Expression Editor** dialog box with the expression configured to **R:Admin**.

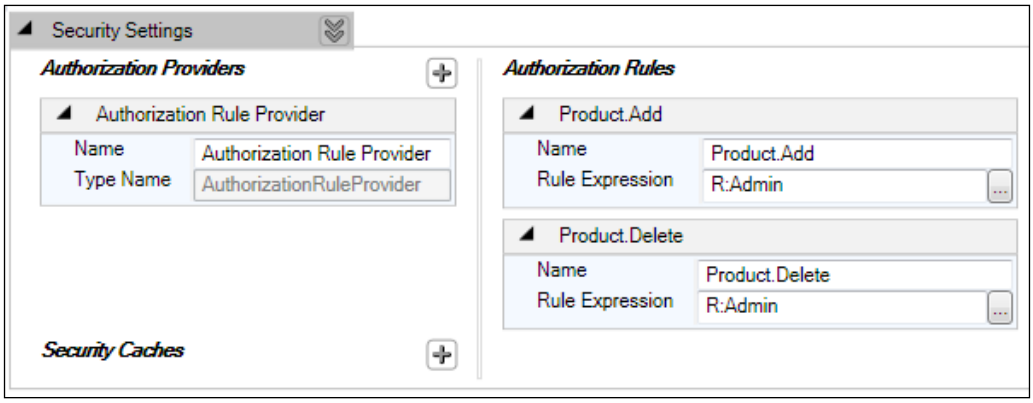


The following screenshot shows the **Rule Expression** property set to **R:Admin**.



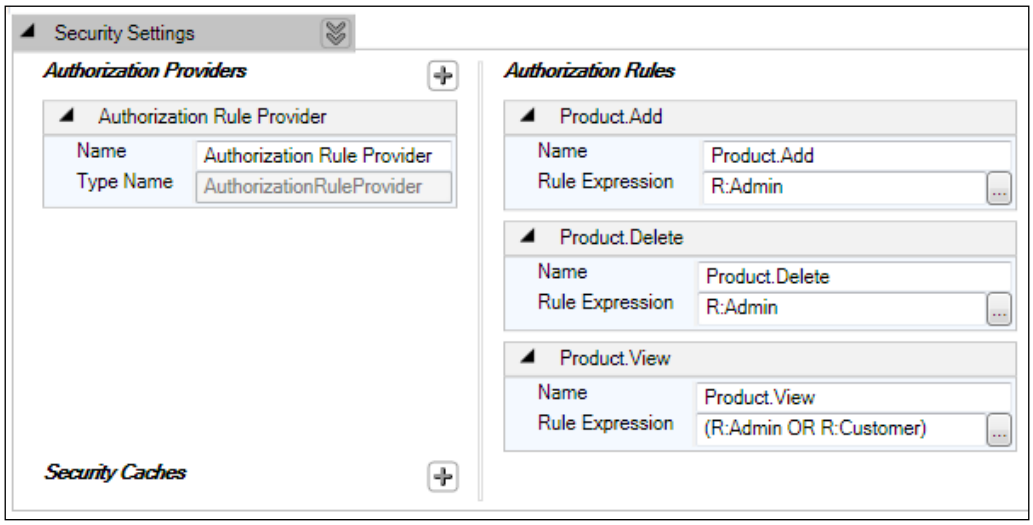
Now let us add the rule for the product delete operation. This rule is configured in a similar fashion. The resulting configuration will be similar to the configuration shown.

The following screenshot displays the added authorization rule named **Product.Delete** with the configured **Rule Expression**:



Alright, we now have to allow all authenticated customers to view the products. Basically we want the authorization to pass if the user is either of role Customer; also Admin role should have permission, only then the user will be able to view products. We will add another rule called **Product.View** and configure the rule expression using the **Rule Expression Editor** as given next. While configuring the rule, use the **OR** operator to specify that either Admin or Customer can perform this operation.

The following screenshot displays the added authorization rule named **Product.View** with the configured **Rule Expression**:



Now that we have the configuration ready, let us get our hands dirty with some code. Before authorizing we need to authenticate the user; based on the authentication requirement we could be using either out-of-the-box authentication mechanism or we might use custom authentication. Assuming that we are using the current Windows identity, the following steps will allow us to authorize specific operations by passing the Windows principal while invoking the `Authorize` method of the Authorization Provider.

1. The first step is to get the `IIdentity` and `IPrincipal` based on the authentication mechanism. We are using current Windows identity for this sample.

   ```
   WindowsIdentity windowsIdentity = WindowsIdentity.GetCurrent();
   WindowsPrincipal windowsPrincipal = new WindowsPrincipal(windowsId
   entity);
   ```

2. Create an instance of the configured Authorization Provider using the `AuthorizationFactory.GetAuthorizationProvider` method; in our case we will get an instance of Authorization Rule Provider.

   ```
   IAuthorizationProvider authzProvider = AuthorizationFactory.GetAut
   horizationProvider("Authorization Rule Provider");
   ```

3. Now use the instance of Authorization Provider to authorize the operation by passing the `IPrincipal` instance and the rule name.

   ```
   bool result = authzProvider.Authorize(windowsPrincipal, "Product.
   Add");
   ```

`AuthorizationFactory.GetAuthorizationProvider` also has an overloaded alternative without any parameter, which gets the default authorization provider configured in the configuration.

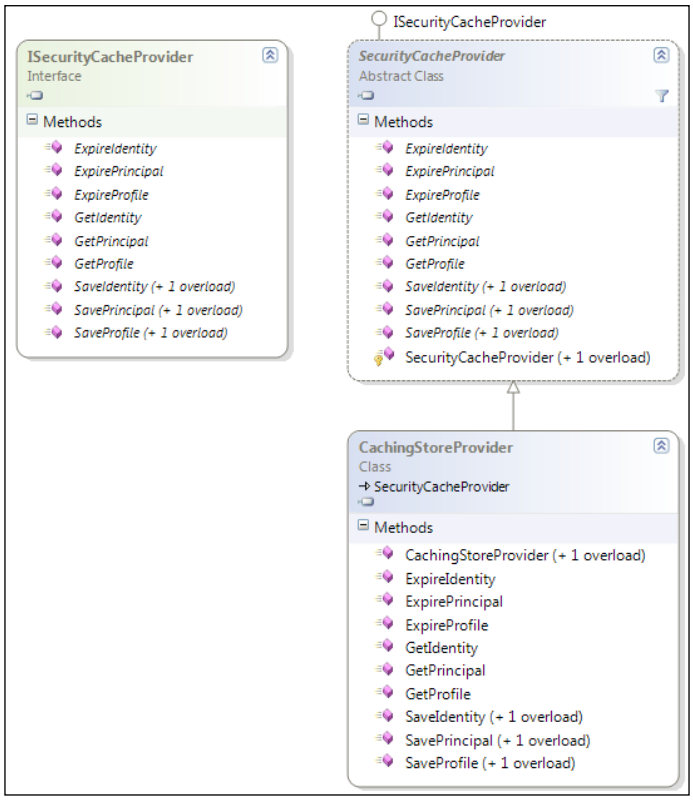
AzMan Authorization Provider

The `AzManAuthorizationProvider` class provides us the ability to define individual operations of an application, which then can be grouped together to form a task. Each individual operation or task can then be assigned roles to perform those operations or tasks. The best part of Authorization Manager is that it provides an administration tool as a **Microsoft Management Console** (**MMC**) snap-in to manage users, roles, operations, and tasks. Policy administrators can configure an **Authorization Manager Policy** store in an **Active Directory**, **Active Directory Application Mode** (**ADAM**) store, or in an XML file. This authorization provider is part of the `Microsoft.Practices.EnterpriseLibrary.Security` namespace.

Understanding Security Cache Provider

Security Cache Provider allows us to cache, retrieve instances of **IIdentity**, **IPrincipal**, or **Profile** objects (such as the **ASP.NET Profile** object), and additionally purge/expire the same. It also generates a token of type **IToken** and this token can be used to purge/expire the cache. The `SecurityCacheProvider` class is an abstract implementation of the `ISecurityCacheProvider` interface; both are part of the `Microsoft.Practices.EnterpriseLibrary.Security` namespace. The `ISecurityCacheProvider` interface consists of methods such as `SaveIdentity`, `SavePrincipal`, and `SaveProfile`; all three methods have their overloaded counterparts to accept an instance of **IToken** to group each of these objects with the same token. It also provides methods such as `GetIdentity`, `GetPrincipal`, and `GetProfile` to retrieve cached credentials; these methods accept instance of **IToken**. Apart from saving and retrieving, `ISecurityCacheProvider` also exposes methods to expire cached items; `ExpireIdentity`, `ExpirePrincipal`, and `ExpireProfile`. These methods accept an instance of **IToken** to expire the respective cached item.

The following diagram shows the members and inheritance relationship of the respective classes related to Security Cache Provider.



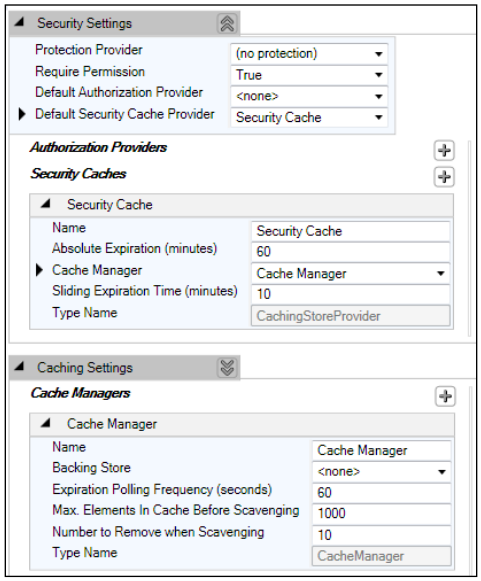
CachingStoreProvider class

The `CachingStoreProvider` class is a concrete implementation of the `SecurityCacheProvider` class; it leverages the Caching Application Block for its caching needs. This class provides the logic to obtain a token for an authenticated user and manage caching for authenticated `IIdentity`, `IPrincipal`, or `Profile` objects (such as the ASP.NET Profile object). The `CachingStoreProvider` class is part of the `Microsoft.Practices.EnterpriseLibrary.Security.Cache.CachingStore` namespace.

Configuring Security Cache Provider

To leverage security caching related functionality, let us add the built-in `CachingStoreProvider` Security Cache Provider in the configuration. This provider uses the caching mechanism implemented by the Caching Application Block. In the configuration file, click on the plus symbol of the **Security Caches** section and navigate to the **Add Security Cache** menu item as shown in the following screenshot:

The following screenshot shows the default configuration of **Security Cache**:



We have configured the Security Cache Provider and are ready to use it in our code to perform various actions against the Security Cache Provider.

Caching and generating a token for an authenticated user

Frequent authentication of user during a single session may lead to performance degradation of the application; we can obtain a temporary token by saving a user principal or a user identity in the security cache. We can save user identity, principal and/or profile; one or more objects can be combined using the same token. Caching an `IIdentity`, `IPrincipal`, or `Profile` is just a two-step process; everything else is taken care of by the configuration. As mentioned earlier, Security Cache Provider uses the Caching Application Block for caching, which gives us all the flexibility of configuration to select the storage mechanism, encryption, and expiration policy. Also, the generated **IToken** can be used to retrieve cached items or mark them for expiration.

The following code snippet gets the current Windows identity and checks whether the identity is authenticated. Upon validation, the instance of Security Cache Provider is used to save the identity and generate the token:

```
//Get current Windows Identity
WindowsIdentity identity = WindowsIdentity.GetCurrent();

if (identity.IsAuthenticated)
{
    ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

    //Cache User Identity and generate token
    IToken token = cacheProvider.SaveIdentity(identity);
}
```

For ASP.NET Web Applications, User Identity can be obtained by accessing the property `Page.User.Identity`.

```
ISecurityCacheProvider cacheProvider =
EnterpriseLibraryContainer.Current.GetInstance
<ISecurityCacheProvider>();
cacheProvider.SaveIdentity(Page.User.Identity);
```

The following code snippet gets the current Windows identity and for the purposes of the demonstration, creates a `GenericPrincipal` object with **Manager** role. The instance of Security Cache Provider is used to save the principal and generate the token:

```
//Get current Windows Identity
WindowsIdentity identity = WindowsIdentity.GetCurrent();

//Constructing dummy Principal Object for demonstration
GenericPrincipal principal = new GenericPrincipal(identity, new
string[] { "Manager" });

if (identity.IsAuthenticated)
{
    ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

    //Cache IPrincipal and generate token
    IToken token = cacheProvider.SavePrincipal(principal);
}
```

For ASP.NET Web Applications, the respective `IPrincipal` instance can be obtained by accessing the property `Page.User`.

The following code snippet demonstrates the Profile caching feature; the `SaveProfile` method of Security Cache Provider is used to save the profile object and generate the token:

```
ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();
IToken token = cacheProvider.SaveProfile(HttpContext.Current.Profile);
```

Associating a token with User Identity, Principal and Profile objects

We can associate an existing token while caching instead of generating a new token, which allows grouping of Identity, Principal, and Profile objects. To utilize this grouping functionality, we have to use the respective overloaded save method and pass the instance of the token as the second parameter.

The following code snippet demonstrates how to associate the generated token while saving Identity, Principal, and Profile objects:

```
//Constructing dummy Principal Object for demonstration
GenericPrincipal principal = new GenericPrincipal(Page.User.Identity,
new string[] { "Manager" });

if (Page.User.Identity.IsAuthenticated)
{
    ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

    //Cache IIdentity and generate token
    IToken token = cacheProvider.SaveIdentity(Page.User.Identity);

    //Cache IPrincipal and group token with related items
    cacheProvider.SavePrincipal(principal, token);

    //Cache Profile object and group token with related items
    cacheProvider.SaveProfile(HttpContext.Current.Profile, token);
}
```

Retrieving User Identity, User Principal, and Profile objects

The following code block first creates an instance of the Security Cache Provider and then saves the respective items, which generates an **IToken** instance, which can be used to retrieve the respective item. Currently **IToken** is an instance of **GuidToken**, which generates a **Guid**; this can be stored for the user's session in the appropriate location based on the application type. The token can be re-generated using the **Guid** and authentication information can be validated as well as authorization being performed by retrieving the **IPrincipal** instance from the security cache.

The following code snippet demonstrates how to retrieve the Identity object using the generated token:

```
//Get current Windows Identity
IIdentity identity = WindowsIdentity.GetCurrent();

ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

//Cache Identity and generate token
IToken token = cacheProvider.SaveIdentity(identity);

//Retrieve Identity using token
IIdentity cachedIdentity = cacheProvider.GetIdentity(token);
```

The following code snippet demonstrates how to retrieve the Principal object using the generated token:

```
//Constructing dummy Principal Object for demonstration
GenericPrincipal principal = new GenericPrincipal(Page.User.Identity,
new string[] { "Manager" });

if (Page.User.Identity.IsAuthenticated)
{
    ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

    //Cache IIdentity and generate token
    IToken token = cacheProvider.SaveIdentity(Page.User.Identity);

    //Cache IPrincipal and group token with related items
    cacheProvider.SavePrincipal(principal, token);

    //Retrieve cached Principal using token
    cacheProvider.GetPrincipal(token);
}
```

The following code snippet demonstrates how to retrieve the Profile object using the generated token:

```
ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

//Cache Profile object and generate token
IToken token = cacheProvider.SaveProfile(HttpContext.Current.Profile);

//Retrieve cached Profile using token
ProfileBase profile = cacheProvider.GetProfile(token) as ProfileBase;
```

Expiring User Identity, User Principal, and Profile objects

Security Cache Provider also provides the ability to expire the cached item when the user logs out of the system or the session ends so that the token cannot be misused. This functionality is part of the `ExpireIdentity`, `ExpirePrincipal`, and `ExpireProfile` methods of Security Cache Provider. In the given code blocks, we are creating an instance of the Security Cache Provider and then saving the respective items, which generates an **IToken** instance. The same token is used to force expiration of the cached item. Please note we are deliberately performing the cache and immediately forcing expiration in the next line just to give you the full picture.

The following code snippet demonstrates how to purge/expire the saved Identity using the generated token:

```
ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

//Cache IIdentity and generate token
IToken token = cacheProvider.SaveIdentity(Page.User.Identity);

//Purge/Expire an existing cached Identity using token
cacheProvider.ExpireIdentity(token);
```

The following code snippet demonstrates how to purge/expire the saved Principal using the generated token:

```
//Constructing dummy Principal Object for demonstration
GenericPrincipal principal = new GenericPrincipal(Page.User.Identity,
new string[] { "Manager" });

if (Page.User.Identity.IsAuthenticated)
{
    ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

    //Cache IIdentity and generate token
    IToken token = cacheProvider.SaveIdentity(Page.User.Identity);

    //Cache IPrincipal and group token with related items
    cacheProvider.SavePrincipal(principal, token);

    //Purge/Expire the existing cached Principal using token
    cacheProvider.ExpirePrincipal(token);
}
```

The following code snippet demonstrates how to purge/expire the saved Profile object using the generated token:

```
ISecurityCacheProvider cacheProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISecurityCacheProvider>();

//Cache Profile object and generate token
IToken token = cacheProvider.SaveProfile(HttpContext.Current.Profile);

//Purge/Expire the cached Profile using token
cacheProvider.ExpireProfile(token);
```

Implementing a custom Authorization Provider

The Security Application Block provides extension points to implement a custom authorization provider; we may extend either the `IAuthorizationProvider` interface or the abstract class `AuthorizationProvider`. The **Authorize** method is where we need to provide our authorization logic. Both the extension points are part of the `Microsoft.Practices.EnterpriseLibrary.Security` namespace.

Following is the `IAuthorizationProvider` interface which exposes the `Authorize` method:

```
public interface IAuthorizationProvider
{
    bool Authorize(IPrincipal principal, string context);
}
```

The following code snippet shows the implementation of the `AuthorizationProvider` abstract class, which inherits the `IAuthorizationProvider` interface and provides wiring of the instrumentation provider for instrumentation purposes:

```
public abstract class AuthorizationProvider : IAuthorizationProvider
{
    IAuthorizationProviderInstrumentationProvider
instrumentationProvider;

    protected AuthorizationProvider()
        : this(new NullAuthorizationProviderInstrumentationProvider())
    {
    }

    protected
AuthorizationProvider(IAuthorizationProviderInstrumentationProvider
instrumentationProvider)
    {
        if (instrumentationProvider == null) throw new ArgumentNullExc
eption("instrumentationProvider");

        this.instrumentationProvider = instrumentationProvider;
    }

    public abstract bool Authorize(IPrincipal principal, string
context);

    protected IAuthorizationProviderInstrumentationProvider
InstrumentationProvider
    {
        get { return this.instrumentationProvider; }
    }
}
```

Custom XML Authorization Provider

Implementing a custom authorization provider is pretty straight-forward. As mentioned previously, we can inherit from the `AuthorizationProvider` class and provide an override the `Authorize` method to provide our authorization logic. Apart from that, we also have to decorate the class with the `ConfigurationElementType` attribute. To make our job easy, the application block provides the `CustomAuthorizationProviderData` class, which holds a configuration object for custom providers. This class is part of the `Microsoft.Practices.EnterpriseLibrary.Security.Configuration` namespace.

The following code snippet shows a typical custom Authorization Provider implementation:

```
[ConfigurationElementType(typeof(CustomAuthorizationProviderData))]
public class XmlAuthorizationProvider : AuthorizationProvider
{
    public XmlAuthorizationProvider(NameValueCollection
configurationItems) { }

    public override bool Authorize(IPrincipal principal, string
context)
    {
        // Custom authorization logic goes here

        // Return true or false based on the authorization outcome
        return false;
    }
}
```

Summary

In this chapter, we discussed the key features of the Security Application Block and have explored the elements of Authorization and Security Cache Providers. We have learned about the various required and optional assemblies. We saw how to configure the initial configuration and also the Authorization Rule Provider, Authorization Rules, as well as Security Cache Provider. We have also learned to authorize based on the configured rules and perform various operations such as saving, retrieving, and expiring instances of **IIdentity**, **IPrincipal**, and **Profile** objects using the Security Cache Provider. Finally, we observed how to implement a custom authorization provider.

8
Cryptography Application Block

Cryptography is an ancient art and science of hiding information to protect sensitive information from the bad guys. It was extensively used even before the computer age. During those times, cryptography was concerned solely with message confidentiality (encryption). **Encryption** is the process of converting information called plaintext into an unreadable form called cipher-text, and decryption is the opposite where cipher-text is converted back to plaintext. The most basic form of cipher is a transposition cipher, which involves rearranging the order of letters, for example "**attack today**" will become "**tatakc otdya**". Substitution cipher is another type of cipher, which replaces letters or group of letters with other letters or group of letters. Several interesting means of hiding information were introduced by imaginative and intelligent people/groups. Cryptography has evolved and modern-day cryptography in general involves three types of cryptographic algorithms: symmetric (secret key) algorithms, asymmetric (public key) algorithms, and hash functions. Symmetric algorithms use a single key for encryption and decryption, asymmetric algorithms uses two keys, one for encryption and the other for decryption, and hash functions are one-way cryptography and since the plaintext is not recoverable they do not require any key.

An application dealing with sensitive data available in memory, stored in a database, file, or any other storage medium is vulnerable to theft. Encryption provides protection from such threats by encrypting data using a key and reconstructs the original data by decrypting it using the same key. Similarly, **hashing** provides a mechanism through which we can maintain data integrity by creating and comparing the generated hash with the original input, and is generally used to save a password or check for message integrity.

The Cryptography Application Block simplifies implementation of hashing and symmetric encryption functionality in our application. As you might be aware, the .NET Framework provides the Cryptography API as part of the `System.Security.Cryptography` namespace for this very purpose. The application block takes it a step further by abstracting the application code from the intricacies of specific cryptography providers. It allows us to create and compare hashes, encrypt and decrypt data using the configured hashing and symmetric cryptography providers respectively. Hashing in cryptography is a mechanism through which an input is converted into fixed size string (hash value); this process is generally referred to as one-way hashing function as the hash value cannot be re-converted to the original input. This can be used to perform message integrity checks, store sensitive data such as password that doesn't need to be retrieved, digital signatures, and so on. Encryption in cryptography is a process transforming an input or plain text into an unreadable form called cipher text. This transformation is performed using an algorithm with a key.

The following are the key features of the Cryptography Application Block:

- Provides hashing functionality with a simple API to generate and compare hash values
- Several hash providers are available out of the box for common hashing algorithms
- Extension point to implement custom hash provider
- Provides symmetric cryptography functionality to encrypt/decrypt data
- Several symmetric cryptography providers are available out of the box for common encryption algorithms
- Configuration editor support to configure hashing and cryptography providers

In this chapter, you will:

- Be introduced to the Cryptography Application Block
- Be introduced to Hashing and Hash Providers
- Be introduced to Cryptography and Cryptography Providers
- Learn about referencing the required and optional assemblies
- Learn to set up the initial configuration
- Learn to configure the hash provider
- Learn to generate hash value for a given string
- Learn to compare hash value with a string
- Learn to implement a custom Hash Provider

- Learn to configure the symmetric cryptography provider
- Learn to encrypt data
- Learn to decrypt data
- Learn to implement custom Symmetric Cryptography Provider

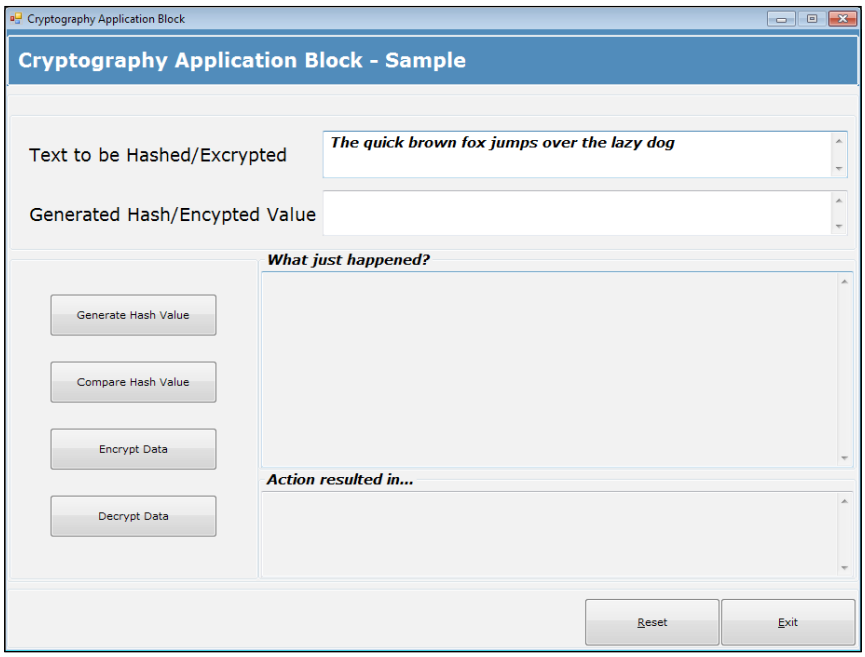
Developing an application

Before we leverage and dig deeper into individual features of the Cryptography block, we will create a simple application, which will help us to get up to speed with the basics; in this section we will do the following:

- Reference the Logging block assemblies
- Add Namespaces
- Set up the initial configuration

To complement the concepts and sample code snippet of this book and allow you to gain quick hands-on experience of different features of the Cryptography Application Block, we have created a sample demonstration application that provides implementation of generating and comparing hashes and encrypting/decrypting data.

The following is a screenshot of the sample application:



Referencing required and optional assemblies

For the purposes of this demonstration, we will be referencing non-strong-named assemblies but based on individual requirements, Microsoft strong-named assemblies or a modified set of custom assemblies can be referenced as well.

The following table lists the required/optional assemblies:

Assembly	Required/Optional
`Microsoft.Practices.EnterpriseLibrary.Common.dll`	Required
`Microsoft.Practices.ServiceLocation.dll`	Required
`Microsoft.Practices.Unity.dll`	Required
`Microsoft.Practices.Unity.Interception.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Security.Cryptography.dll`	Required
`Microsoft.Practices.EnterpriseLibrary.Caching.dll`	**Optional:** Used while leveraging **SerializationUtility** class for serializing and de-serializing objects to and from byte streams

Adding namespaces

Instead of fully qualifying the types on every instance of their usage, we can add the namespaces given next to the source code file to use the Cryptography block elements without fully qualifying each reference.

Core Namespace:

- `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography`

Configuration Namespace (Optional): Required while using the **EnterpriseLibraryContainer** to instantiate objects.

- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`

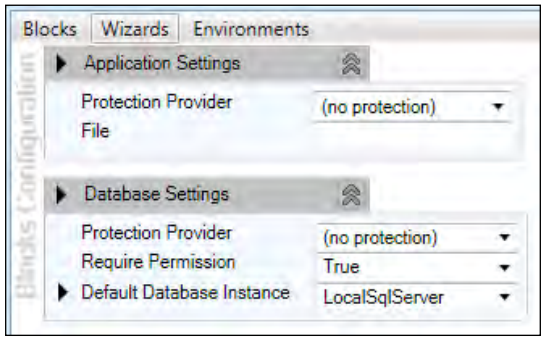
Unity Namespace (Optional): Required while instantiating objects using **UnityContainer**.

- `System.Configuration`
- `Microsoft.Practices.Unity`
- `Microsoft.Practices.Unity.Configuration`

Adding initial cryptography settings

Before we can leverage the features of the Cryptography Application Block, we have to add the initial **Cryptography Settings** to the configuration. Open the **Enterprise Library configuration editor** either using the shortcut available in **Start | All Programs | Microsoft patterns & practices | Enterprise Library 5.0 | Enterprise Library Configuration** or by just right-clicking the configuration file in the **Solution Explorer** window of **Visual Studio IDE** and clicking on **Edit Enterprise Library V5 Configuration**. Initially, we will have a blank configuration file with default **Application Settings** and **Database Settings**.

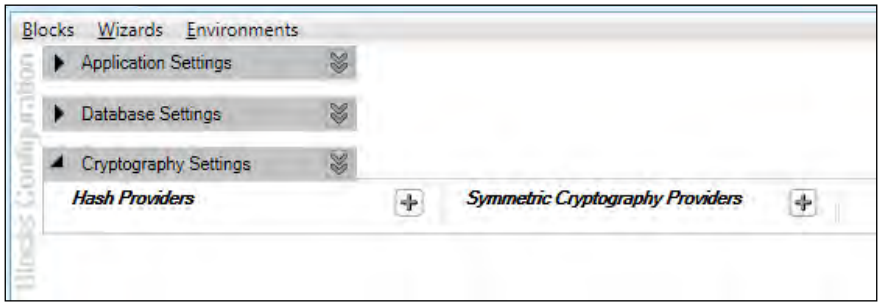
The following screenshot displays the default settings displayed in the configuration editor:



Let us go ahead and add the **Cryptography Settings** in the configuration file. Select the menu option **Blocks**, which lists many different settings to be added to the configuration. Click on the **Add Cryptography Settings** menu item to add the security configuration settings.

The following screenshot shows the menu option **Add Cryptography Settings**:

Once we click on **Add Cryptography Settings**, the configuration editor will display the default **Cryptography Settings** as shown in the following screenshot:



Notice that the setting consists of two sections: **Hash Providers** and **Symmetric Cryptography Providers**. We will change the configuration further, but for now, we are in good shape with regards to the initial infrastructure configuration.

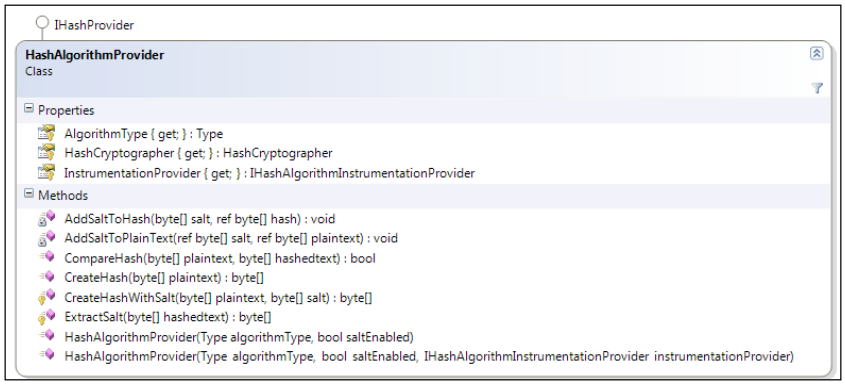
Working of Hash Provider

The Cryptography Application Block is developed with same principles as for other application blocks—it separates the implementation from the usage. So it means the hash provider configuration can be updated without impacting the application code. But how does this work? The application block provides an interface called `IHashProvider`, which is part of `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography` namespace; this contract defines two methods, `CreateHash`, and `CompareHash` to create and compare hashes respectively. These methods accept both input and hash value as byte arrays; any class implementing this interface is required to provide the implementation for both the `CreateHash` and `CompareHash` methods.

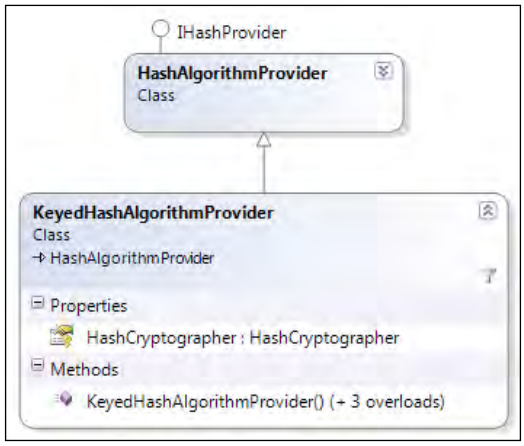
The following screenshot shows the definition of the `IHashProvider` interface:

The `HashAlgorithmProvider` class, which inherits from the `IHashProvider` interface, is a hash provider implementation for hash algorithms derived from the `System.Security.Cryptography.HashAlgorithm` class. This class internally utilizes the `HashCryptographer` class, which provides basic cryptographic services for a hash algorithm. It also has another hash provider named `KeyedHashAlgorithmProvider` apparently inheriting from the `HashAlgorithmProvider` class; as the name suggests this is a hash provider for hash algorithms deriving from the `System.Security.Cryptography.KeyedHashAlgorithm` class.

The following diagram shows the definition of the `HashAlgorithmProvider` class:



The following diagram shows the members and inheritance hierarchy of the `KeyedHashAlgorithmProvider` class.



The first step in leveraging the hash functionality is to configure a hash provider in the configuration file using the configuration editor. Once we have the configuration in place, we can access the required functionality either using the static facade, service locator, or using Unity container directly. Regardless of approach, the application block identifies the configured hash provider and loads it to be used while creating and comparing hash value.

Creating CryptographyManager and IHashProvider instances

We have several options at hand; while utilizing the static facade we can either use the static `Cryptographer` class or we can create `CryptographyManager` or `IHashProvider` implementation objects using Unity service locator or using Unity container directly. A few approaches such as configuring the container through a configuration file or code are not listed here but the recommended approach is either to use the Unity service locator for applications with few dependencies or create objects using Unity container by loading through configuration or programmatically to leverage the benefits of this approach. Use of the static factory class is not recommended.

Using the static facade

Static factory classes were the default approach to creating objects with versions prior to 5.0. This approach is no longer recommended and is still available for backwards compatibility. The Cryptography Application Block provides a static class called `Cryptographer` available in the `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography` namespace. This static facade class provides methods to generate and compare hashes, and also encrypt and decrypt data using the configured providers; this approach does not require creation of any object to perform the required actions.

Using Unity service locator

This approach is recommended for applications with few dependencies; the `EnterpriseLibraryContainer` class exposes a static property called `Current` of type **IServiceLocator**, which resolves and gets an instance of the specified type.

The following code snippet creates an instance of `CryptographyManager`:

```
CryptographyManager cryptoManager = EnterpriseLibraryContainer.
Current.GetInstance<CryptographyManager>();
```

The following is a code snippet to create a deafult `IHashProvider` implementation instance using Unity service locator:

```
IHashProvider defaultHashProvider = EnterpriseLibraryContainer.
Current.GetInstance<IHashProvider>();
```

The following is a code snippet to create a named `IHashProvider` implementation instance using Unity service locator:

```
IHashProvider hashProvider = EnterpriseLibraryContainer.Current.GetIns
tance<IHashProvider>("SHA256Managed");
```

Using Unity container directly

Larger complex applications demand looser coupling. This approach leverages the dependency injection mechanism to create objects instead of explicitly creating instances of concrete implementations. Unity container resolves objects using the type registrations and mappings; these can be configured programmatically or through a configuration file. Based on the configuration, it resolves the appropriate type whenever requested. The following example instantiates a new Unity container object and adds the Enterprise Library Core Extension. This loads the configuration and makes registrations and mappings of the Enterprise Library available.

The following is a code snippet to create a default `CryptographyManager` instance using `UnityContainer`:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
CryptographyManager cryptoManager = container.
Resolve<CryptographyManager>();
```

The following is a code snippet to create a deafult `IHashProvider` implementation instance using `UnityContainer`:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
IHashProvider defaultHashProvider = container.
Resolve<IHashProvider>();
```

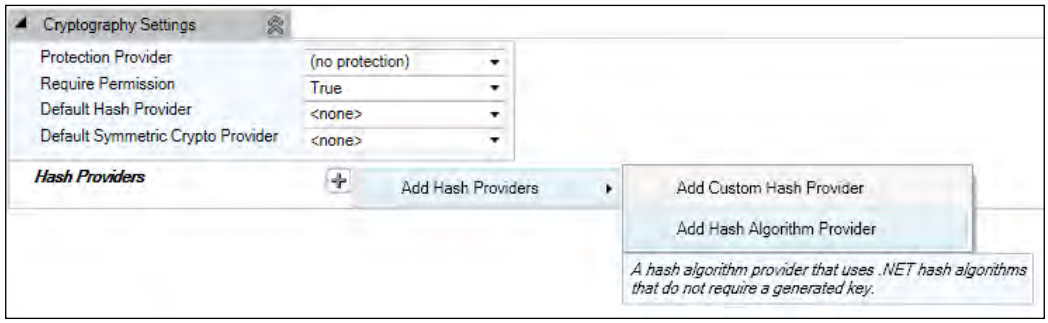
The following is a code snippet to create a named `IHashProvider` implementation instance using `UnityContainer`:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
IHashProvider hashProvider = container.Resolve<IHashProvider>("SHA256
Managed");
```

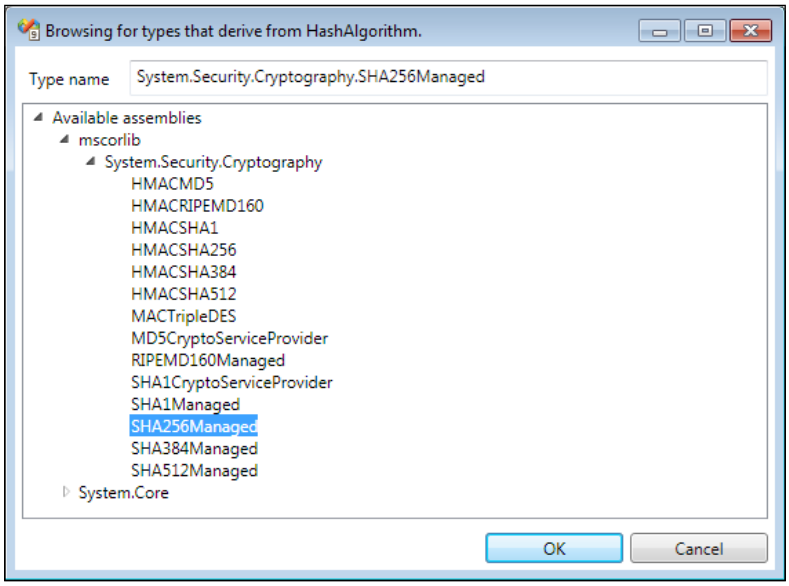
Configuring Hash Provider

We have already learned to add **Cryptography Settings** to the configuration file; click on the plus symbol provided on the top right-corner of the **Hash Providers** section, navigate, and click **Add Hash Providers** | **Add Hash Algorithm Provider**. This action will display the Hash Algorithm selection dialog box. For the purposes of this demonstration, we will select **SHA256Managed**, which is part of `System.Security.Cryptography` namespace, and hit the **OK** button.

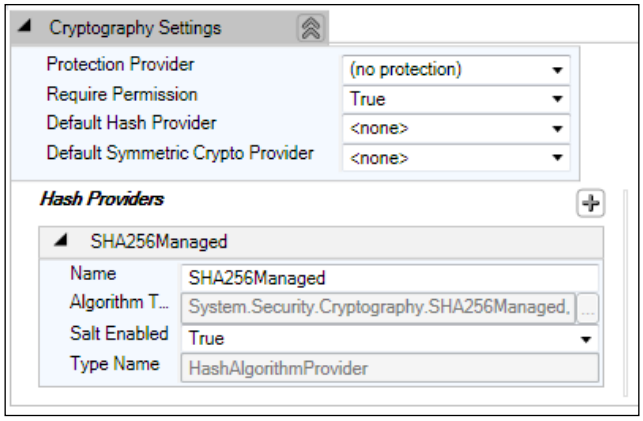
The following screenshot shows the menu option **Add Hash Algorithm Provider**:



Once we click on the menu option **Add Hash Algorithm Provider**, the following HashAlgorithm selection dialog is displayed:



The previous action will result in addition of **SHA256Managed** hash provider in the configuration as shown in the following screenshot.



By default the property **Salt Enabled** is set to **True**; this property determines whether a random string (salt value) is generated and pre-pended to the plain text before hashing. Salt values help in protecting against dictionary attacks by making it difficult to generate the hash.

Generating a hash value

Generating the hash for a given `string` or `byte[]` is pretty simple; we create an instance of either `CryptographyManager` or `IHashProvider` using the methods described in the section *Creating CryptographyManager and IHashProvider instances*. Once we have the respective object, we can invoke the `CreateHash` method. While creating a hash using `IHashProvider` the `CreateHash` method accepts and returns a `byte[]` whereas `CryptographyManager` provides an overloaded `CreateHash` method that accepts and returns `string` objects as well. `IHashProvider` allows us to leverage the **Default Hash Provider** configuration and additionally a named instance can also be constructed.

The following code snippet shows the creation of the hash for the given text using a `CryptographyManager` instance:

```
CryptographyManager cryptoManager = EnterpriseLibraryContainer.
Current.GetInstance<CryptographyManager>();

string hashValue = cryptoManager.CreateHash("SHA256Managed", "Some
text to be hashed");
```

In the above code snippet, we have created an instance of `CryptographyManager` or rather its real implementation the `CryptographyManagerImpl` class. Next, we invoke the `CreateHash` method, passing the configured hash provider name **SHA256Managed** and the plain text that must be hashed. Although the generated hash value will not make any sense, it will definitely give you an idea of its gibberish nature.

The following text is the generated hash value:

4yieYiwA9YXAbGiIme1GWtjKJtXUpbKOiQl6Q6VApL30zCXtuL1UfQgTTAA1TItq

Comparing hash values

Comparing hash values for a given `string` or `byte[]` is also a simple affair. We follow the same process of creating an instance of either `CryptographyManager` or `IHashProvider` using the methods described in the section *Creating CryptographyManager and IHashProvider instances*. Next, we invoke the `CompareHash` method; depending on the object, we may or may not pass the hash instance name as the first parameter; the other two parameters accepts plain text and the hashed value for comparison.

The following code snippet shows the comparison of a hash value with the given text:

```
CryptographyManager cryptoManager = EnterpriseLibraryContainer.
Current.GetInstance<CryptographyManager>();

string hashValue =
"4yieYiwA9YXAbGiIme1GWtjKJtXUpbKOiQl6Q6VApL30zCXtuL1UfQgTTAA1TItq";

bool result = cryptoManager.CompareHash("SHA256Managed", "Some text to
be hashed", hashValue);
```

The `CompareHash` method for the above given code snippet will result in a return value of `true`.

Implementing a custom Hash Provider

Although the .NET Framework provides implementation of several hash algorithms, there might be a scenario in which we will need to use a custom hash provider to meet certain proprietary or statutory requirements. The Cryptography block provides extensibility points that allow us to configure a custom hash provider without re-compiling the code. Apart from the assemblies listed in section *referencing required and optional assemblies*, we will have to add an additional reference of `System.Configuration.dll`. This assembly is used to indicate the configuration object type specified using the `ConfigurationElementType` attribute. As pointed out previously, the `IHashProvider` interface is the contract that every hash provider must implement. This interface provides two methods, `CreateHash` and `CompareHash`.

Adding the following namespaces will help in saving IDE real estate and improve readability of the code and so it is recommended to add these namespaces.

- `System.Collections.Specialized`
- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`
- `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography`
- `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography.Configuration`

Although the following code is self explanatory, we will attempt to make a quick walkthrough of the code snippet. We have a written a class named `CustomHashProvider`, which inherits from the `IHashProvider` interface and provides a stub implementation for demonstration purposes. Just in case you are wondering what the first line of code is all about, the attribute `ConfigurationElementType` attribute indicates the configuration object `CustomHashProviderData` used for `CustomHashProvider`. We also have to provide a constructor that accepts a parameter of type `NameValueCollection`.

The following code snippet demonstrates the implementation of a custom hash provider:

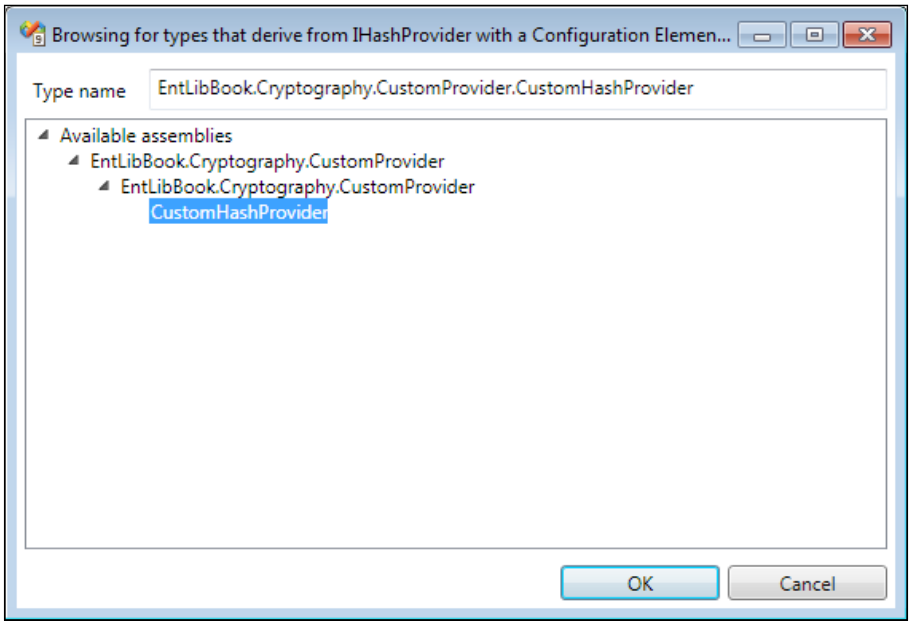
```
[ConfigurationElementType(typeof(CustomHashProviderData))]
public class CustomHashProvider : IHashProvider
{
    public CustomHashProvider(NameValueCollection attributes)
    {
    }

    public bool CompareHash(byte[] plaintext, byte[] hashedtext)
    {
        // Create hash of plain text and compare with hashed text
    }

    public byte[] CreateHash(byte[] plaintext)
    {
        // Implementing Custom Hashing Logic
    }
}
```

Configuring a Custom Hash Provider

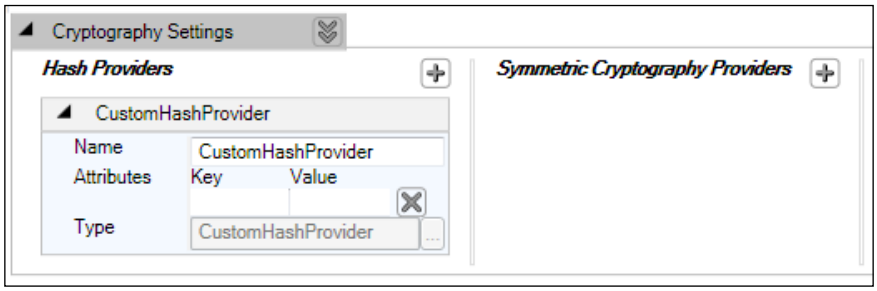
While leveraging the custom hash provider implementation, we have to follow a slightly different route for configuration. In the **Hash Providers** of **Cryptography Settings** click on the plus symbol; navigate and click **Add Hash Providers** | **Add Custom Hash Provider**. This action will display a types browsing dialog box listing the types derived from the `IHashProvider` interface. For the purposes of this demonstration, we will select the `CustomHashProvider` implementation. Post selection, we will have the custom hash provider configured; the configuration editor allows us to add custom key/value attributes, which will be passed on to the constructor of `CustomHashProvider`.

The following screenshot shows the menu option **Add Custom Hash Provider**:

Once we click on the **Add Custom Hash Provider** menu option, custom hash provider selection dialog is displayed as shown in the given screenshot:



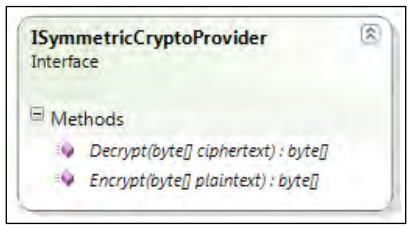
Once we select the custom hash provider type and click the **OK** button, the configuration editor adds the selected hash provider as shown in the following screenshot:



With these simple steps, the custom hash provider is configured and is ready to be used.

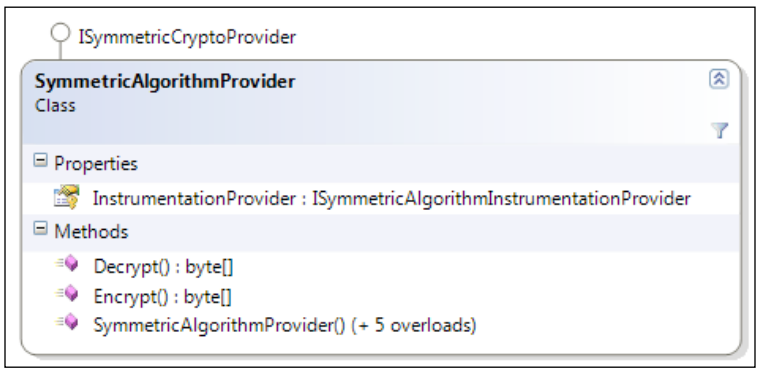
Working of symmetric cryptography providers

The symmetric cryptography provider works the same way as the hash provider except for the fact that the interface and class involved are different. The `ISymmetricCryptoProvider` interface defines the core contract for configurable symmetric cryptographic implementation; this interface is part of the `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography` namespace. This interface defines two methods, **Encrypt** and **Decrypt**, as shown in the following diagram:



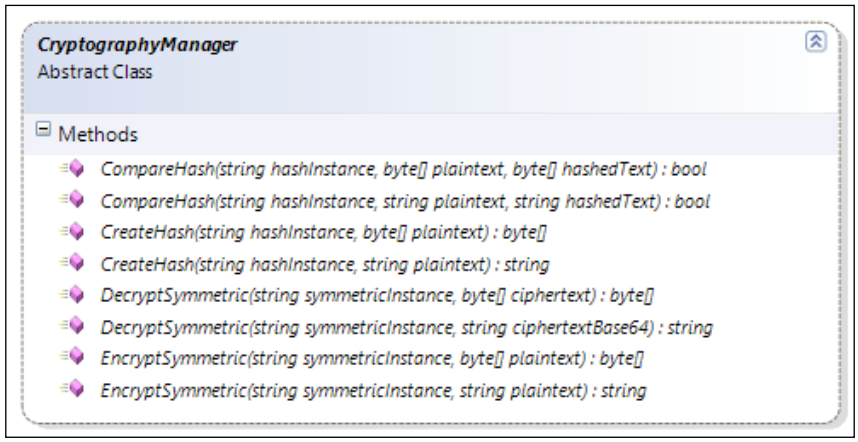
The `SymmetricAlgorithmProvider` class inherits the `ISymmetricCryptoProvider` interface and provides implementation for algorithms derived from the `System.Security.Cryptography.SymmetricAlgorithm` class.

The following diagram shows the members of the `SymmetricAlgorithmProvider` class:



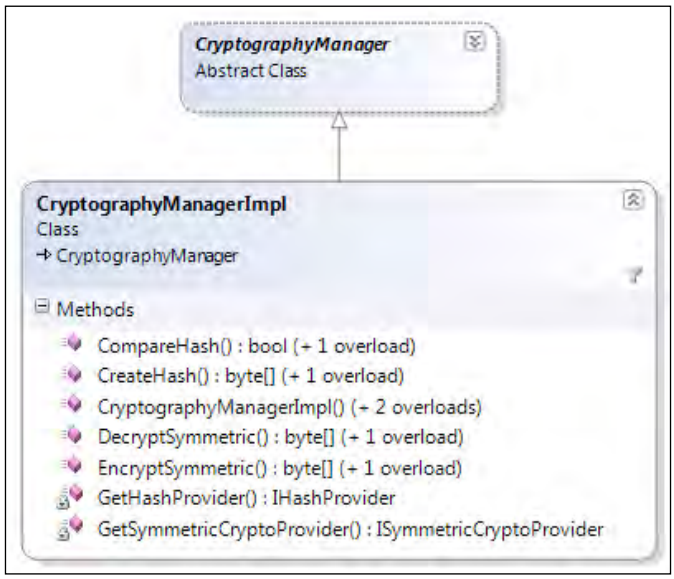
`CryptographyManager` is an abstract class, which wraps functionality from instances of both `IHashProvider` and `ISymmetricCryptoProvider`. The application block identifies the configured hash and symmetric cryptography provider and provides both hashing and encryption functionality by exposing the respective methods. It also provides a nifty method that accepts plain text and returns plain text while creating a hash.

The following diagram shows the methods exposed by the `CryptographyManager` abstract class:



The `CryptographyManagerImpl` class inherits from the abstract class `CryptographyManager`; this class provides the real implementation, which identifies both the providers and loads the configured providers. This class loads both the hash and cryptography providers from the configuration and leverages them to perform the respective actions.

The following diagram shows the inheritance hierarchy and methods exposed by the `CryptographyManagerImpl` class:



Creating CryptographyManager and ISymmetricCryptoProvider instances

We have already explored creating instances of CryptographyManager in the *Creating CryptographyManager and IHashProvider Instances* section, so in this section we will focus on creating instances of `ISymmetricCryptoProvider` using Unity service locator and using the Unity container directly.

Using the static facade

The `Cryptographer` static class provides `EncryptSymmetric` and `DecryptSymmetric` methods to perform encryption and decryption of data. Additionally, it accepts `string` as well as `byte[]` for both encryption and decryption. As discussed previously, being a static facade the methods can be invoked directly.

Using Unity service locator

Creating instances using Unity service locator has already been explored in the section *Creating CryptographyManager and IHashProvider Instances*.

We have already learned to create an instance of `CryptographyManager` while working with hashing functionality in the section *Creating CryptographyManager and IHashProvider Instances*; `CryptographyManager` also provides methods to perform encryption and decryption of data.

The following is a code snippet to create a deafult `ISymmetricCryptoProvider` instance using `UnityContainer`:

```
ISymmetricCryptoProvider defaultCryptoProvider =
EnterpriseLibraryContainer.Current.GetInstance<ISymmetricCryptoProvid
er>();
```

The following is a code snippet to create a `ISymmetricCryptoProvider` named instance using `UnityContainer`:

```
ISymmetricCryptoProvider cryptoProvider = EnterpriseLibraryContainer.
Current.GetInstance<ISymmetricCryptoProvider>("DPAPI Symmetric Crypto
Provider");
```

Using Unity container directly

While learning to leverage the hashing functionality in the section *Creating CryptographyManager and IHashProvider Instances* we explored creating instances using Unity container directly and the same applies to this section as well.

The following is a code snippet to create a deafult `ISymmetricCryptoProvider` instance using `UnityContainer`:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
ISymmetricCryptoProvider defaultCryptoProvider = container.Resolve<ISy
mmetricCryptoProvider>();
```

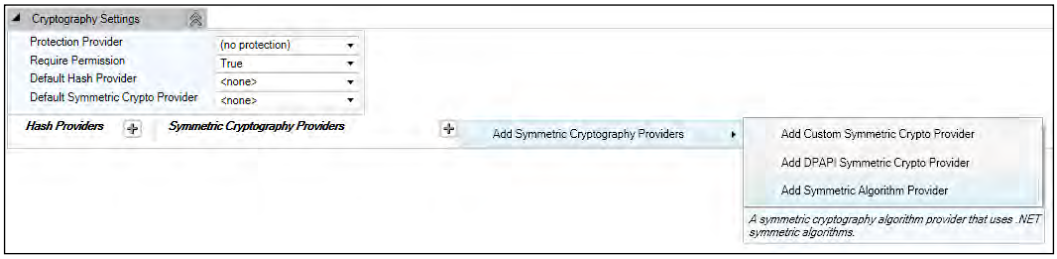
The following is a code snippet to create a `ISymmetricCryptoProvider` named instance using `UnityContainer`:

```
var container = new UnityContainer();
container.AddNewExtension<EnterpriseLibraryCoreExtension>();
ISymmetricCryptoProvider defaultCryptoProvider = container.Resolve<ISy
mmetricCryptoProvider>("DPAPI Symmetric Crypto Provider");
```

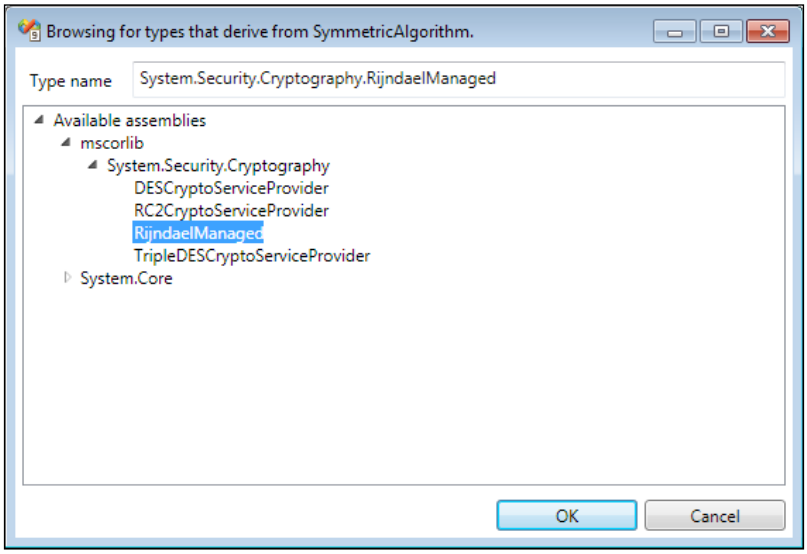
Configuring the symmetric cryptography provider

We have already learned to add **Cryptography Settings** to the configuration file; click on the plus symbol provided on the top-right corner of the **Symmetric Cryptography Providers** section, navigate, and click **Add Symmetric Cryptography Providers | Add Symmetric Algorithm Provider**. This action will display the Symmetric Algorithm selection dialog box. For the purposes of this demonstration, we will select the `RijndaelManaged` algorithm implementation, which is part of the `System.Security.Cryptography` namespace, and hit the **OK** button.

The following screenshot shows the menu option **Add Symmetric Algorithm Provider**:

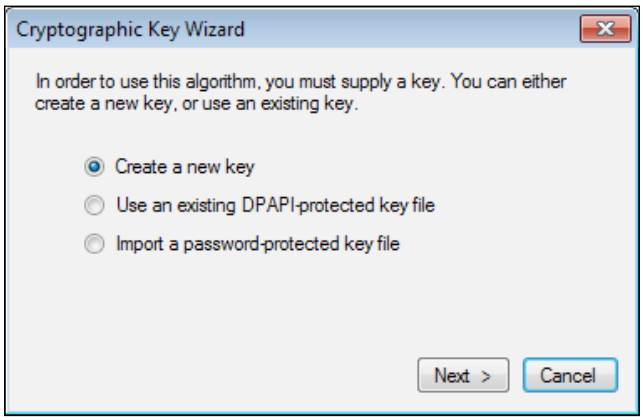


Once we click on the menu option **Add Symmetric Algorithm Provider**, the Symmetric algorithm selection dialog box is displayed as shown in the following screenshot:

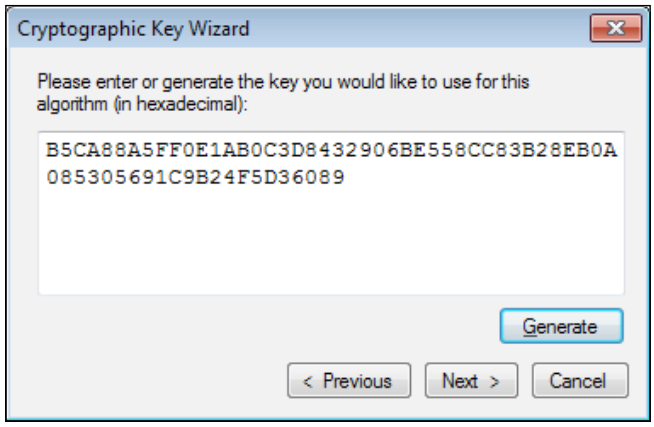


Selection of the Symmetric Algorithm will result in display of the **Cryptographic Key Wizard** dialog. Basically, the algorithm requires a key that can be used to encrypt and decrypt data. We can either create a new key, use an existing **Data Protection API** (**DPAPI**)-protected key file, or import a password-protected key file. For the purposes of this demonstration, we will opt to **Create a new key** and click **Next**.

The following screenshot shows the **Cryptographic Key Wizard** dialog box:



We are now prompted to either enter the key or generate the key using the **Generate** button. Click on **Generate** and a new key will be generated and displayed in the textbox as shown in the following screenshot. Click **Next** to move to the next step of the wizard.

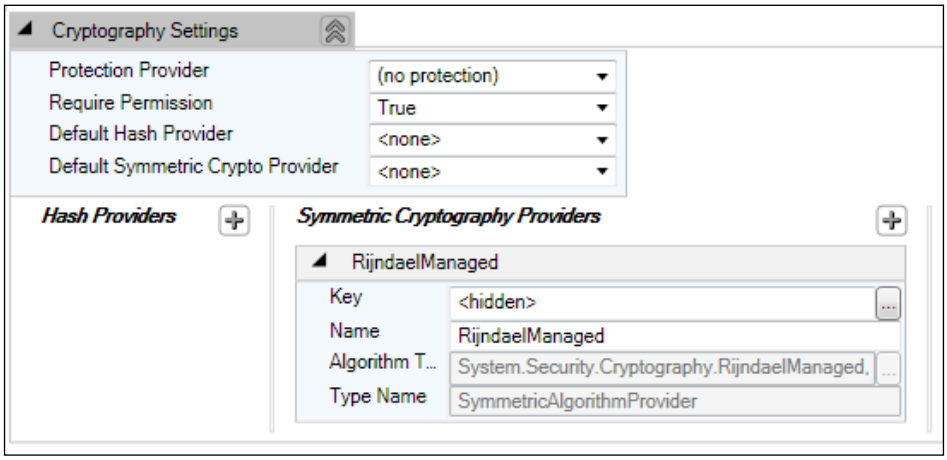


Once we click **Next**, we will be prompted to choose the file storage path. We can provide the appropriate path and key filename by clicking the ellipsis "**...**" button. Click **Next** to move to the next step of the wizard.

So far we have generated the key and specified the path and filename to store the key. But the key itself is not yet protected and vulnerable; this step prompts us to protect the key using the Data Protection API (**DPAPI**). We have to select the data protection mode; **User mode** encrypts the key using the credentials of the currently logged-in user while the **Machine mode** allows any users on this computer to encrypt or decrypt the key. For the purposes of this demonstration, we will select **User mode**; click the **Finish** button to close the wizard.

The following screenshot shows the data protection mode options:

Once the key wizard dialog is closed, we will end up with the **Cryptography Settings** configuration as shown in the following screenshot:

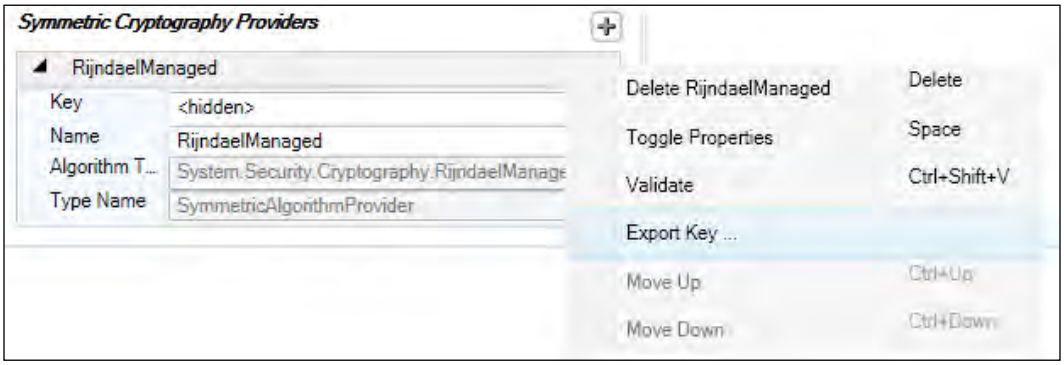


We are now done with the configuration of the Symmetric Cryptography Provider. We can now leverage the provider to encrypt/decrypt data but before we explore encryption/decryption, we will learn the important task of exporting the generated key in the next section.

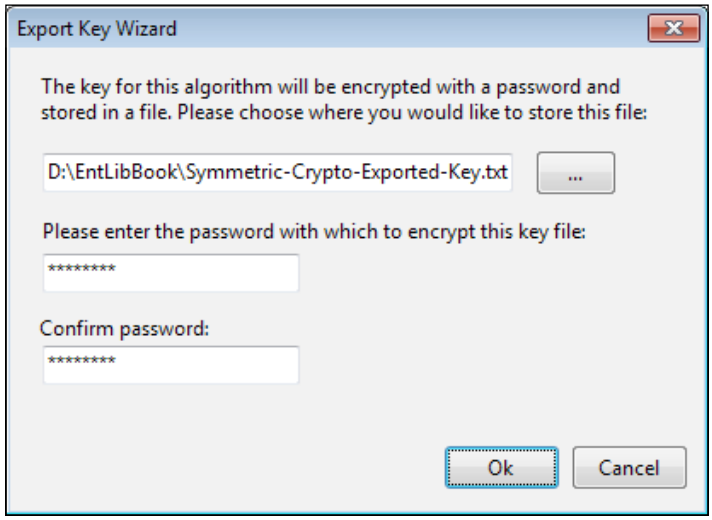
Exporting the key

We can export the generated key and save it in a file for backup purposes; the key is encrypted with the provided password. Since this file contains the key for encryption, it must be protected and only trusted users must be given access through ACL.

1. Right-click on the specific symmetric cryptography provider and click on the **Export Key ...** menu item as shown in the following screenshot.



2. The previous action will display an **Export Key Wizard** dialog; this dialog requires a key file path and a password, using which the key will be encrypted.



3. On clicking the **OK** button, the key file will be saved in the specified path with the encrypted key.

Encrypting data

The simplicity of the API makes it very easy to perform encryption operations. Encrypting data requires two/three input based on the approach (Factory vs. Service Locator vs. Unity) taken. For the purposes of this demonstration, we are using the service locator to get an instance of `CryptographyManager` and perform encryption using the configured symmetric algorithm provider.

The following code snippet gets an instance of `CryptographyManager` and encrypts the given data using the named provider:

```
CryptographyManager cryptoManager = EnterpriseLibraryContainer.
Current.GetInstance<CryptographyManager>();

//Encrypt Data Using Configured Symmetric Cryptography Provider Named
'RijndaelManaged'
//Returns encrypted data
string encryptedData = cryptoManager.EncryptSymmetric("RijndaelManag
ed", "Data to be encrypted");
```

The given code snippet returns the following encrypted data:

```
5B0oeiIoQNO5A2C/LE+L6Ax7ecPxU4jDQJ8I+j0Z8+VadOaqVcj7HdWMWHyDpfYblqxFgB
qNvfijMONAxNYCBQ==
```

Decrypting data

For the purposes of this demonstration we are using the service locator to get an instance of `CryptographyManager` and perform decryption using the configured symmetric algorithm. Note that the input for encrypted data is the result of the encryption action performed in the previous section; the same algorithm is used to decrypt the data.

The following code snippet demonstrates data decryption using the `DecryptSymmetric` method:

```
CryptographyManager cryptoManager = EnterpriseLibraryContainer.
Current.GetInstance<CryptographyManager>();

string encryptedData = "5B0oeiIoQNO5A2C/LE+L6Ax7ecPxU4jDQJ8I+j0Z8+VadO
aqVcj7HdWMWHyDpfYblqxFgBqNvfijMONAxNYCBQ==";

//Decrypt Data Using Configured Symmetric Cryptography Provider Named
'RijndaelManaged'
//Returns encrypted data
string decryptedData = cryptoManager.DecryptSymmetric("RijndaelManag
ed", encryptedData);
```

The given code snippet returns the following decrypted data:

```
Data to be encrypted
```

Implementing a custom symmetric provider

Although the .NET Framework provides implementation of several symmetric cryptography algorithms there might be a scenario in which we will need to use a custom symmetric encryption to meet certain proprietary or statutory requirements. The Cryptography block provides extensibility points that allow us to configure a custom symmetric encryption provider without re-compiling the code. Apart from the assemblies listed in the section *Referencing required and optional assemblies*, we will have to add an additional reference of `System.Configuration.dll`. This assembly is used to indicate the configuration object type specified using the `ConfigurationElementType` attribute. We can implement a custom symmetric provider by inheriting the `ISymmetricCryptoProvider` interface and providing logic to `Encrypt` and `Decrypt` data.

Adding the following given namespaces will help in saving IDE real estate and improve readability of the code and so it is recommended to add these namespaces.

- `System.Collections.Specialized`
- `Microsoft.Practices.EnterpriseLibrary.Common.Configuration`
- `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography`
- `Microsoft.Practices.EnterpriseLibrary.Security.Cryptography.Configuration`

Although the given code is self explanatory, we will attempt to make a quick walkthrough of the code snippet. We have written a class named `CustomEncryptionProvider`, which inherits from the `ISymmetricCryptoProvider` interface and provides a stub implementation for demonstration purposes.

The following code snippet provides a skeleton implementation of the custom encryption provider:

```
[ConfigurationElementType(typeof(CustomSymmetricCryptoProviderData))]
public class CustomEncryptionProvider : ISymmetricCryptoProvider
{
    public byte[] Decrypt(byte[] ciphertext)
    {
        // Implement Decryption Logic
    }

    public byte[] Encrypt(byte[] plaintext)
```

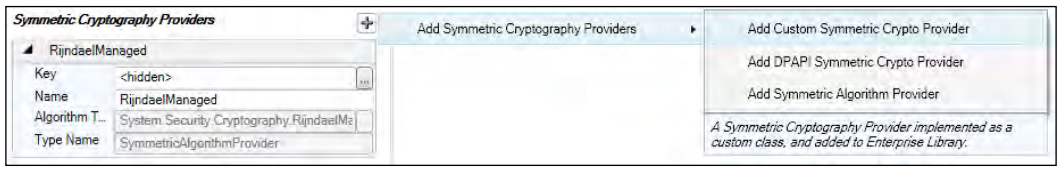
```
        {
            // Implement Encryption Logic
        }
    }
```

Just in case if you are wondering what the first line of code is all about, the attribute `ConfigurationElementType` indicates the configuration object `CustomSymmetricCryptoProviderData` used for `CustomEncryptionProvider`. We also have to provide a constructor that accepts a parameter of type `NameValueCollection`. Implementation of a custom encryption provider is quite straightforward as seen in the above code snippet; we just need to provide our custom encryption/decryption logic in the `Encrypt` and `Decrypt` methods respectively.

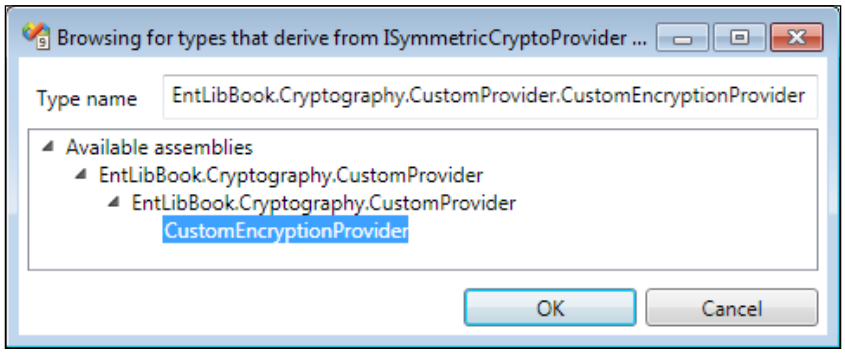
Configuring the custom symmetric provider

Configuration of a custom symmetric provider takes a slightly different approach. In the **Symmetric Cryptography Providers** section of **Cryptography Settings** click on the plus symbol; navigate and click on the menu option **Add Symmetric Cryptography Providers | Add Custom Symmetric Crypto Provider**.

The following screenshot shows the menu option **Add Custom Symmetric Crypto Provider**:

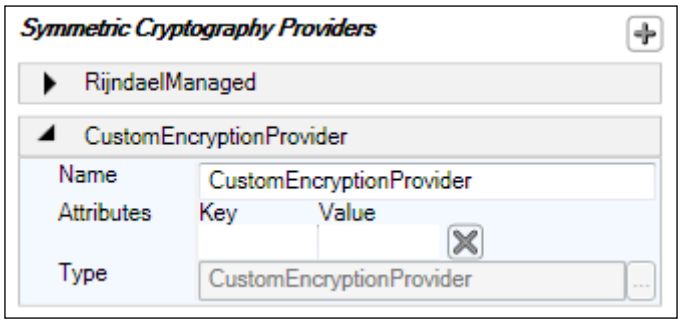


This action will display a types browsing dialog box listing the types derived from the `ISymmetricCryptoProvider` interface. The following is a screenshot of the custom symmetric cryptography provider selection dialog.



For the purposes of this demonstration, we will select the `CustomEncryptionProvider` implementation as shown in the given screenshot. Once we select the provider and click on the **OK** button, the custom encryption provider is added to the configuration file as shown in the following screenshot. The configuration editor allows us to add custom key/value attributes, which will be passed on to the constructor of `CustomEncryptionProvider`.

The following screenshot shows the selected custom **Symmetric Cryptography Provider**:



After configuring the cryptography provider, we will be able to leverage the custom encryption and decryption functionality without changing or impacting the application code.

Summary

In this chapter, we have learned the fundamental elements of the Cryptography Application Block such as `IHashProvider`, `ISymmetricCryptoProvider`, `CryptographyManager`, and so on. We have explored the various required and optional assemblies, the initial infrastructure configuration, and the individual feature-level configuration. We have also learned to initialize the `IHashProvider`, `ISymmetricCryptoProvider`, and `CryptographyManager` classes. We further learned to generate hashes, compare hashes, and implement custom hash providers. We also explored encryption and decryption of data and understood the implementation basics of a custom symmetric cryptography provider.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Thank you for buying

Microsoft Enterprise Library 5.0

About Packt Publishing

Packt, pronounced 'packed', published its first book "Mastering phpMyAdmin for Effective MySQL Management" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Enterprise

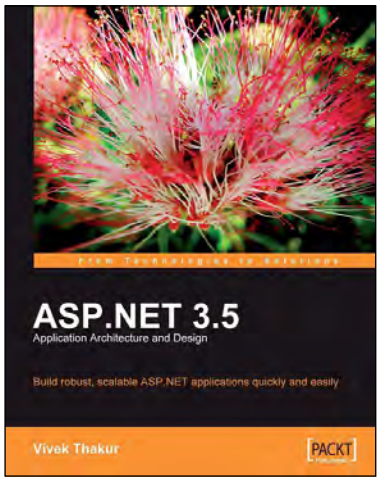
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING



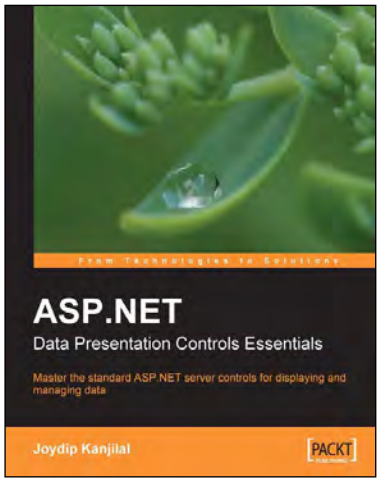
ASP.NET 3.5 Application Architecture and Design

ISBN: 978-1-847195-50-0 Paperback: 260 pages

Build robust, scalable ASP.NET applications quickly and easily

1. Master the architectural options in ASP.NET to enhance your applications
2. Develop and implement n-tier architecture to allow you to modify a component without disturbing the next one
3. Design scalable and maintainable web applications rapidly
4. Implement ASP.NET MVC framework to manage various components independently

---



ASP.NET Data Presentation Controls Essentials

ISBN: 978-1-847193-95-7 Paperback: 256 pages

Master the standard ASP.NET server controls for displaying and managing data

1. Systematic coverage of major ASP.NET data presentation controls
2. Packed with re-usable examples of common data control tasks
3. Covers LINQ and binding data to ASP.NET 3.5 (Orcas) controls