Exception Handling Application Block Hands-On Labs for Enterprise Library

patterns & practices
proven practices for predictable results

This walkthrough should act as your guide to learn about the Exception Handling Application Block and practice how to leverage its capabilities in various application contexts.

After completing this lab, you will be able to do the following:

- You will be able to add Exception Logging to an application.
- You will be able to use a Replace Handler to hide sensitive information.

This hands-on lab includes the following two labs:

The estimated completion for this lab is **30 minutes**.

Authors

These Hands-On Labs were produced by the following individuals:

- Program Management: Grigori Melnik and Madalyn Parker (Microsoft Corporation)
- Development/Testing: Fernando Simonazzi (Clarius Consulting), Chris Tavares (Microsoft Corporation), Mariano Grande (Digit Factory), and Naveen Pitipornvivat (Adecco)
- Documentation: Fernando Simonazzi (Clarius Consulting), Grigori Melnik, Alex Homer, Nelly Delgado and RoAnn Corbisier (Microsoft Corporation)

In this release of Enterprise Library, the recommended approach to creating instances of Enterprise Library objects is to use the programmatic approach and instantiate the objects directly. You may decide to store some configuration information externally (such as in a custom configuration section or in the Windows Azure service settings) and use this information when you create the Enterprise Library objects programmatically. In this way, you can expose just those settings that you want to make available, rather than all the settings, which is the case when you use declarative configuration. Each block is self-contained and does not have dependencies on other blocks for basic operations. Typically, creating an instance is a simple operation that takes only a few lines of code.

You could use Unity, or another dependency injection container, to manage your Enterprise Library objects and their dependencies with the corresponding lifecycles. Unlike in the previous release, it is now your responsibility to register and resolve the types you plan to use. Unity 3 now supports the registration by convention to make it easier to do so. See the Developer's Guide to Dependency Injection Using Unity for more info.

However, to simplify the examples and make it easier to see the code that uses the features of each of the Enterprise Library Application Blocks, the Hands-On Labs examples use the simpler approach for resolving Enterprise Library objects from the container by using the **GetInstance** method of the container service locator. You will see this demonstrated in each of the examples.

To learn more about using dependency injection to create instances of Enterprise Library objects, see the documentation available on MSDN at http://msdn.microsoft.com/entlib/.

Lab 1: Logging Exceptions

In this lab, you will take an application without exception handling, and add local and global exception handlers that log the exceptions to the Event Log using the Exception Handling Application Block.

To begin this exercise, open the Puzzler.sln file located in the ex01\begin folder.

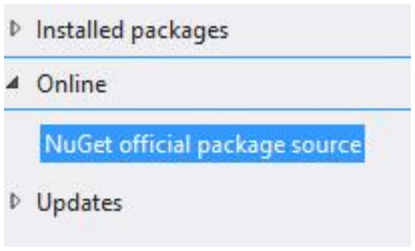
To review the application

1. This application performs two functions: it checks the spelling of words against a dictionary (unix dict for size) and it uses the dictionary to generate a list of words that can be constructed from a character list.
2. Select the **Debug | Start Debugging** menu command to run the application.

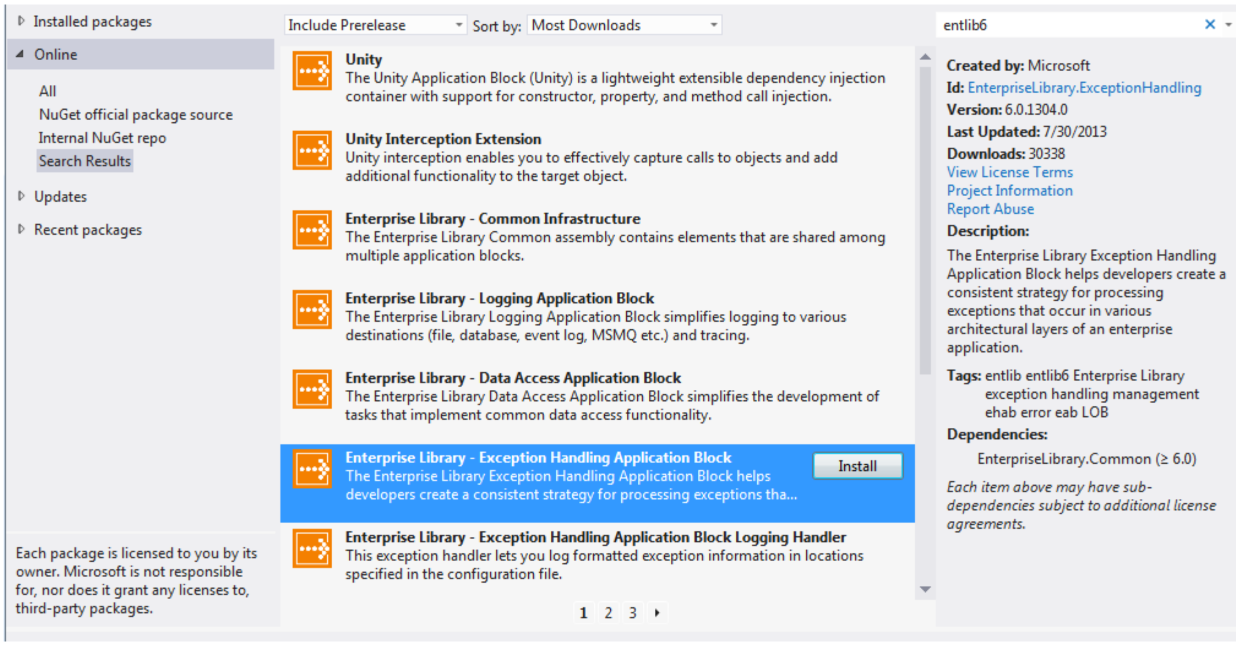
   There is currently no exception handling in the application. Attempt to add a word that contains numbers to the dictionary (type "ab123" in the **word to check** text box, and then click **Add Word**). An unhandled exception will occur, which will break into the debugger.
3. Select the **Debug | Stop Debugging** menu command to exit the application and return to Visual Studio.

To add try/catch exception handling

1. Select the **PuzzlerUI** project. Select the **Project | Manage NuGet Packages** menu command.
2. Select the “Online” option to view NuGet packages available online.



3. Search for **EntLib6** in the search bar. Select **Enterprise Library – Exception Handling Application Block** and click install.



Each package is licensed to you by its owner. Microsoft is not responsible for, nor does it grant any licenses to, third-party packages.

Unity
The Unity Application Block (Unity) is a lightweight extensible dependency injection container with support for constructor, property, and method call injection.

Unity Interception Extension
Unity interception enables you to effectively capture calls to objects and add additional functionality to the target object.

Enterprise Library - Common Infrastructure
The Enterprise Library Common assembly contains elements that are shared among multiple application blocks.

Enterprise Library - Logging Application Block
The Enterprise Library Logging Application Block simplifies logging to various destinations (file, database, event log, MSMQ etc.) and tracing.

Enterprise Library - Data Access Application Block
The Enterprise Library Data Access Application Block simplifies the development of tasks that implement common data access functionality.

Enterprise Library - Exception Handling Application Block
The Enterprise Library Exception Handling Application Block helps developers create a consistent strategy for processing exceptions tha...

Enterprise Library - Exception Handling Application Block Logging Handler
This exception handler lets you log formatted exception information in locations specified in the configuration file.

Created by: Microsoft
Id: EnterpriseLibrary.ExceptionHandling
Version: 6.0.1304.0
Last Updated: 7/30/2013
Downloads: 30338
View License Terms
Project Information
Report Abuse
Description:
The Enterprise Library Exception Handling Application Block helps developers create a consistent strategy for processing exceptions that occur in various architectural layers of an enterprise application.
Tags: entlib entlib6 Enterprise Library exception handling management ehab error eab LOB
Dependencies:
EnterpriseLibrary.Common (≥ 6.0)
Each item above may have sub-dependencies subject to additional license agreements.

4. Also install the **Enterprise Library – Logging Application Block** and **Enterprise Library – Exception Handling Application Block Logging Handler** NuGet packages.
5. Select the **Puzzler.cs** file in the Solution Explorer. Select the **View | Code** menu command.
6. Add the following namespace inclusion at the top of the file:

```
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
```

7. Find the **btnAddWord_Click** method, and add a try/catch section around the AddWord and SetError calls. (Inserted code in bold):

```
private void btnAddWord_Click(object sender, System.EventArgs e)
{
  try
  {
    // TODO: Handle exceptions
    PuzzlerService.Dictionary.AddWord(txtWordToCheck.Text);
    errorProvider1.SetError(txtWordToCheck, "");
  }
  catch (Exception ex)
  {
    bool rethrow = ExceptionPolicy.HandleException(ex, "UI Policy");
    if (rethrow)
      throw;

    MessageBox.Show(string.Format(
                 "Failed to add word {0}, please contact support.",
                 txtWordToCheck.Text));
  }
}
```

Note: It is very important to just use the throw statement, rather than throw ex. If you have "throw ex," then the stack trace of the exception will be replaced with a stack trace starting at the re-throw point, which is usually not the desired effect.

To configure your application programmatically

Declarative configuration with the Enterprise Library Configuration tool is still supported. If you select to use this method, please refer to the Hands on Labs in Enterprise Library Version 5.0.

1. Open the **Startup.cs** file.
2. Add the following namespace inclusions at the top of the file:

```
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling;
using Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging;
using Microsoft.Practices.EnterpriseLibrary.Logging.Formatters;
using Microsoft.Practices.EnterpriseLibrary.Logging.TraceListeners;
```

3. Add the following method to the **Startup** class**.** This method builds a configuration for exception logging that enables you to write to the event log. It also sets the formatting of the log messages so they are easier to read.

```
private static LoggingConfiguration BuildLoggingConfig()
{
        // Formatters
        TextFormatter formatter =
                new TextFormatter(
                        @"Timestamp: {timestamp}{newline}
Message: {message}{newline}Category: {category}{newline}
Priority: {priority}{newline}EventId: {eventid}{newline}
Severity: {severity}{newline}Title:{title}{newline}
Machine: {localMachine}{newline}
App Domain: {localAppDomain}{newline}
ProcessId: {localProcessId}{newline}
Process Name: {localProcessName}{newline}
Thread Name: {threadName}{newline}
Win32 ThreadId: {win32ThreadId}{newline}
Extended Properties: {dictionary({key} - {value}{newline})}");

       // Listeners
       var flatFileTraceListener =
              new FlatFileTraceListener(
                     @"C:\Temp\Puzzler.log",
                     "----------------------------------------",
                     "----------------------------------------",
                     formatter);
       var eventLog =
              new EventLog("Application", ".",
                     "Enterprise Library Logging");
       var eventLogTraceListener = new
         FormattedEventLogTraceListener(eventLog);

       // Build Configuration
       var config = new LoggingConfiguration();
       config.AddLogSource("General", SourceLevels.All, true)
              .AddTraceListener(eventLogTraceListener);
       config.LogSources["General"]
              .AddTraceListener(flatFileTraceListener);

       // Special Sources Configuration
       config.SpecialSources.LoggingErrorsAndWarnings
              .AddTraceListener(eventLogTraceListener);

       return config;
}
```

4. Add the following method to the **Startup** class. This method bootstrap the Logging and Exception Handling blocks programmatically.

```
private static void BootstrapEnterpriseLibrary()
{
    Logger.SetLogWriter(new LogWriter(BuildLoggingConfig()));

    var uiPolicies = new List<ExceptionPolicyEntry>
    {
        new ExceptionPolicyEntry(
            typeof(Exception),
            PostHandlingAction.None,
            new IExceptionHandler[]
            {
                new LoggingExceptionHandler(
                    "General", 100,
                    TraceEventType.Error,
                    "Enterprise Library Exception Handling", 0,
                    typeof(TextExceptionFormatter), Logger.Writer)
            })
    };
    var policies = new List<ExceptionPolicyDefinition>
        {
            new ExceptionPolicyDefinition("UI Policy",
                uiPolicies),
        };

    ExceptionPolicy.SetExceptionManager(
        new ExceptionManager(policies));
}
```

This creates a new exception policy for UI Exceptions and sets the exception logging to use the configuration created in the **BuildLoggingConfig** method you just added. Each exception handling policy requires a type (this one is an Exception), Post Handling Action (None), and a logging exception handler. The logging exception handler for this UI policy is of category "General," will be logged under event ID 100, and it is at the "Error" level of severity. Its title is "Enterprise Library Exception Handling," it is of priority 0, it will use the TextExceptionFormatter, and use the logWriter you created with your logging configuration.

5. In the **Main** method, invoke the **BootstrapEnterpriseLibrary** method when the application starts.

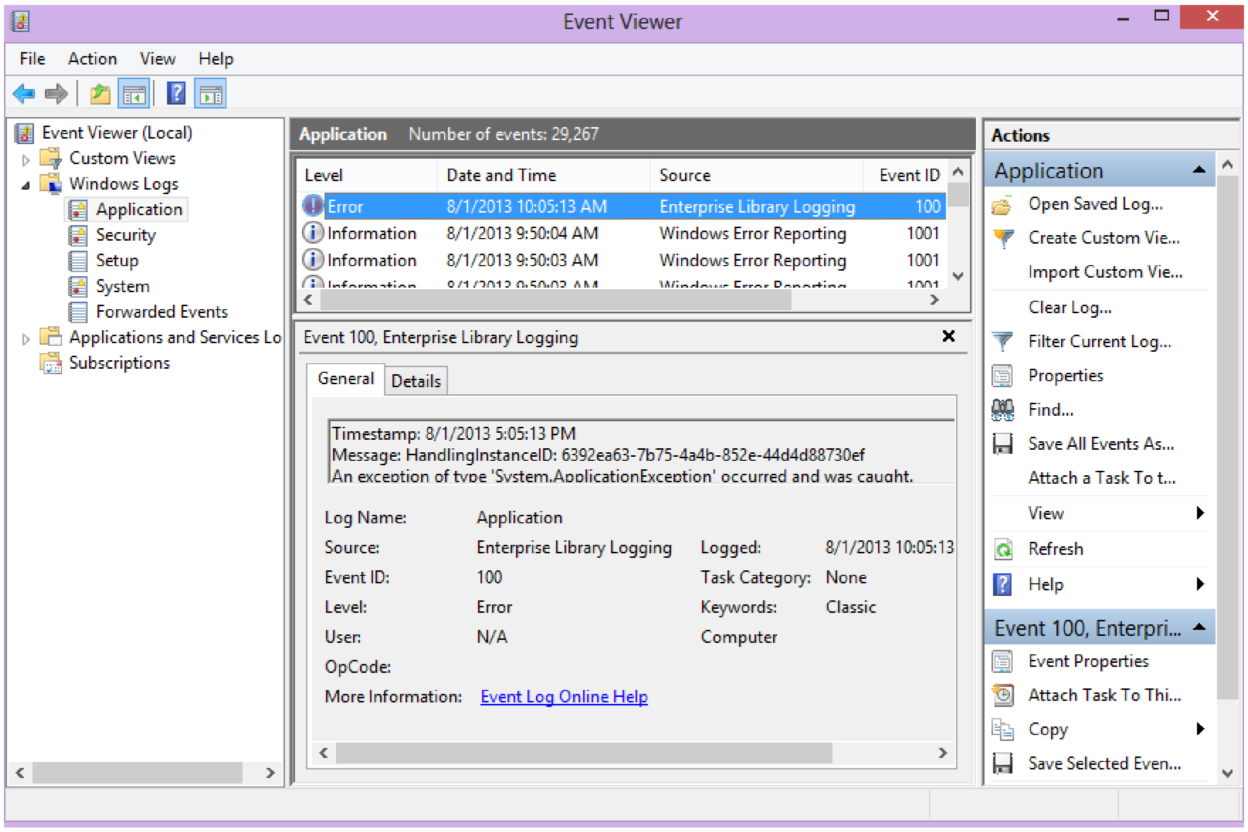
```
static void Main()
{
    BootstrapEnterpriseLibrary();

    // TODO: Handle unhandled exceptions
    Puzzler f = new Puzzler();
    Application.Run(f);
}
```

To run the application

1. Select the **Debug | Start Without Debugging** menu command to run the application.

   Try adding a number (type a number in the **Word to check** text box, and click **Add Word**)— a message box is displayed with an error—"Failed to add word …, please contact support".

2. Check the Event Log by using the Event Viewer (**Control Panel | Administrative Tools | Event Viewer**). Look at the top of the Application Log. The exception will be logged.



Level	Date and Time	Source	Event ID
Error	8/1/2013 10:05:13 AM	Enterprise Library Logging	100
Information	8/1/2013 9:50:04 AM	Windows Error Reporting	1001
Information	8/1/2013 9:50:03 AM	Windows Error Reporting	1001
Information	8/1/2013 9:50:03 AM	Windows Error Reporting	1001

```
Timestamp: 8/1/2013 5:05:13 PM
Message: HandlingInstanceID: 6392ea63-7b75-4a4b-852e-44d4d88730ef
An exception of type 'System.ApplicationException' occurred and was caught.
```

Log Name:	Application		
Source:	Enterprise Library Logging	Logged:	8/1/2013 10:05:13
Event ID:	100	Task Category:	None
Level:	Error	Keywords:	Classic
User:	N/A	Computer	
OpCode:			
More Information:	Event Log Online Help		

After the previous exception, the Exception Management Application Block will have written an entry in the Application Event Log using the Logging Application Block.

> You must run the application at least once using an account that has administrative permissions on the machine to allow entries to be logged, or create the "Enterprise Library Logging" Event Log source in advance.

3. Close the application.

While it is possible to put try/catch sections around all the event handlers in an application, in many cases, you want to do the same thing for every exception (like logging or saving current state) regardless of where the exception occurs. To do this, you add a global exception handler. There are two events you can use to listen for unhandled exceptions:

The **Application.ThreadException** event is raised when an unhandled exception occurs on the thread that is executing the **Application.Run** method.

If an exception is raised during that handler, or occurs on a different thread to the UI, then the **AppDomain.UnhandledException** event will fire.

> In general, you should try and handle exceptions at the point that they occur. This allows you to take the correct action and preserve the exception information more easily. You can pass exceptions to the Exception Handing Application Block using the **HandleException** method. However, for the

purposes of demonstrating how you can use the block to perform a range of tasks, this example uses a global exception handler.

To add global exception handling

1. Select the **Puzzler.cs** file in the Solution Explorer. Select the **View | Code** menu command. Locate the **btnAddWord_Click** method, and remove the exception handling code and **using** statements added earlier.

```
private void btnAddWord_Click(object sender, System.EventArgs e)
{
  PuzzlerService.Dictionary.AddWord(txtWordToCheck.Text);
  errorProvider1.SetError(txtWordToCheck, "");
}
```

2. Select the **Startup.cs** file in the Solution Explorer. Select the **View | Code** menu command.

3. Add the following method to handle exceptions in the **Startup** class.

```
public static void HandleException(Exception ex, string policy)
{
    bool rethrow = false;
    try
    {
        rethrow = ExceptionPolicy.HandleException(ex, policy);
    }
    catch (Exception innerEx)
    {
        string errorMsg = "An unexpected exception occurred while " +
          "calling HandleException with policy'" + policy + "'. ";
        errorMsg += Environment.NewLine + innerEx.ToString();

        MessageBox.Show(errorMsg, "Application Error",
           MessageBoxButtons.OK, MessageBoxIcon.Stop);
        throw ex;
    }

    if (rethrow)
    {
        //WARNING: this will truncate the stack of the exception
        throw ex;
    }
    else
    {
        MessageBox.Show("An unhandled exception occurred"
            + " and has been logged. Please contact support.");
    }
}
```

This method will use the Exception Handling Application Block and will also display valid information if there is a problem with the Exception Handling Application Block itself (for example, missing configuration).

It will also display a message to the user if the exception is swallowed (i.e. not re-thrown).

4. Add the following method as the event handler to the Application **ThreadException** event.

```
static void Application_ThreadException(object sender,
                                        ThreadExceptionEventArgs e)
{
  HandleException(e.Exception, "UI Policy");
}
```

This event handler will use the policy (**UI Policy**) that you defined before for the UI layer. In the next exercise, you will customize this policy to allow certain types of exception to "escape" and shut the application down.

5. Add an event handler for the AppDomain **UnhandledException** event.

```
static void CurrentDomain_UnhandledException(object sender,
                                     UnhandledExceptionEventArgs e)
{
    if (e.ExceptionObject is Exception)
    {
        HandleException(
            (Exception)e.ExceptionObject,
            "Unhandled Policy");
    }
}
```

This handler will use a new policy, named **Unhandled Policy**, which will be set up in the next exercise. The Unhandled Policy should almost always just log the exception, and not re-throw.

6. Connect the event handlers to the events at the beginning of the application by including the following code in the **Main** method (Inserted code in bold).

```
static void Main()
{
      BootstrapEnterpriseLibrary();

      Application.ThreadException +=
             Application_ThreadException;
      AppDomain.CurrentDomain.UnhandledException +=
             CurrentDomain_UnhandledException;

      Puzzler f = new Puzzler();
      Application.Run(f);
}
```

7. Select the **Debug | Start Without Debugging** menu command to run the application.

Try adding a number (type a number in the **Word to check** text box and click **Add Word**)—a message box is displayed—"An unhandled exception occurred and has been logged. Please contact support." Look in the event log for the logged exception.

8. Close the application and Visual Studio.

To verify you have completed the exercise correctly, you can use the solution provided in the ex01\end folder.

Lab 2: Exception Handling Strategies

In this lab, you will secure part of your application service with code access security and then use a Replace handler with the Exception Handling Application Block to hide sensitive information from clients. You will also see how you can filter which exceptions escape the top application layer.

To begin this exercise, open the Puzzler2.sln file located in the ex02\begin folder.

To view the service modifications

1. Select the **PuzzlerService** project **DictionaryService.cs** file in the Solution Explorer. Select the **View | Code** menu command.

This class acts as a **Service Interface** on top of the **Dictionary** class. Within this class you provide exception filtering and transformation before sending the results back to the client.

To protect the service's Add Word functionality with code access security

1. Select the **PuzzlerService** project **Dictionary.cs** file in the Solution Explorer. Select the **View | Code** menu command. Locate the **AddWord** method and decorate it with a security attribute as follows:

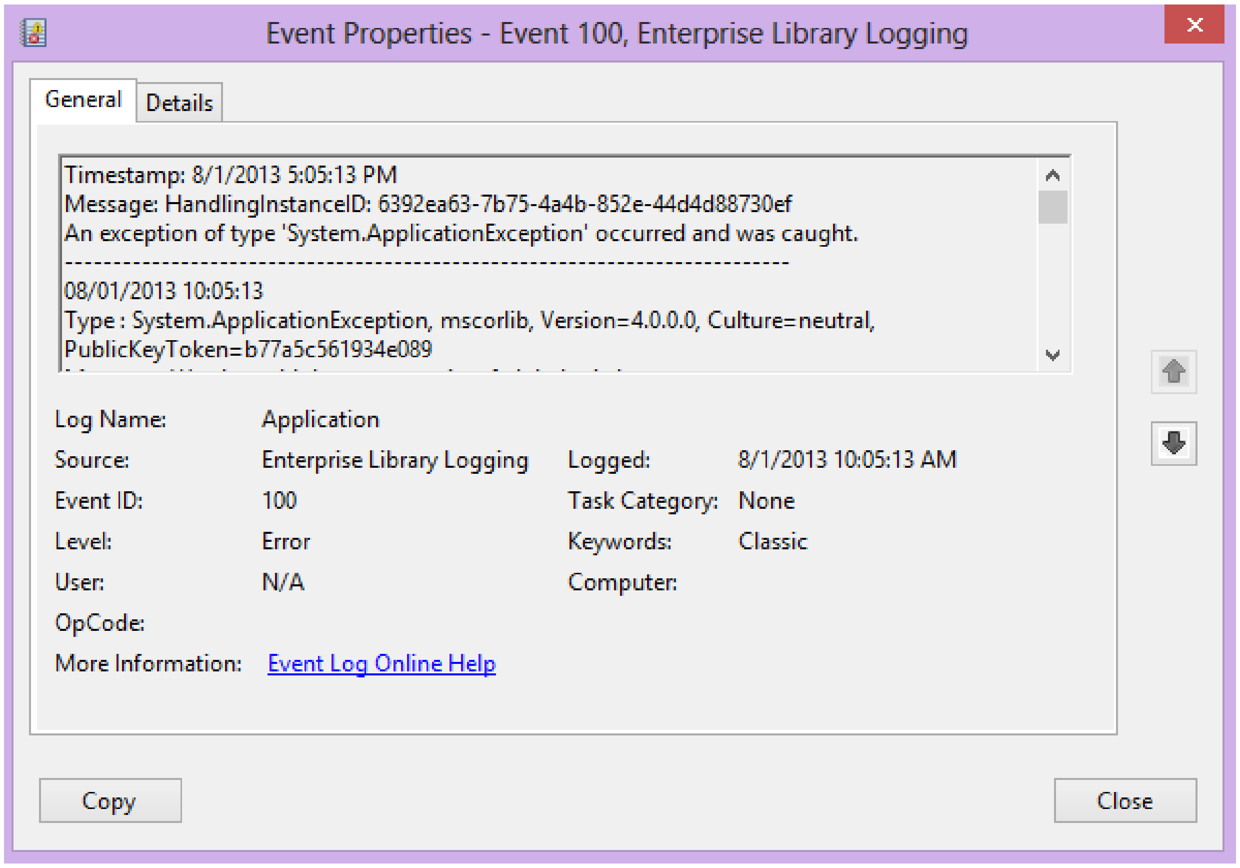
```
// TODO: Add security attribute
[PrincipalPermission(SecurityAction.Demand, Role = "Grand PoohBah")]
public static Boolean AddWord(string wordToAdd)
{
    if (!IsWord(wordToAdd))
    {
        // It is not alphabetic! Throw an exception
        throw new ApplicationException(
            "Word to add does not consist of alphabetic letters");
    }
    if (Dict[wordToAdd] == null)
    {
        Dict.Add(wordToAdd, wordToAdd);
    }
    return true;
}
```

   This method can now only be executed by a member of the role **Grand PoohBah**, an unlikely situation.

   Note: Decorate the **AddWord** method in Dictionary.cs not DictionaryService.cs.

2. Select the **Debug | Start Without Debugging** menu command to run the application.

   Type a nonsense word (alphabetic— no numbers!) into the **Word To Check** text box (ensure that you have an error flashing), and then click the **Add Word** button. This will call the service's **AddWord** function and throw a **SecurityException**, which you can check in the event viewer.



```
Timestamp: 8/1/2013 5:05:13 PM
Message: HandlingInstanceID: 6392ea63-7b75-4a4b-852e-44d4d88730ef
An exception of type 'System.ApplicationException' occurred and was caught.
--------------------------------------------------------------------------
08/01/2013 10:05:13
Type : System.ApplicationException, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089
```

Log Name:	Application		
Source:	Enterprise Library Logging	Logged:	8/1/2013 10:05:13 AM
Event ID:	100	Task Category:	None
Level:	Error	Keywords:	Classic
User:	N/A	Computer:	
OpCode:			
More Information:	Event Log Online Help		

The **SecurityException** may be serialized from a server to a client (over Web Services) and contains information that may help an attacker break your security. You would prefer to catch and log the security exception on the server, then pass an exception containing less information to the client.

3. Close the application.

To configure the application to replace security exceptions

1. Add the following policies to the **BootstrapEnterpriseLibrary** method in the **Startup** class:

```
private static void BootstrapEnterpriseLibrary()
{
    Logger.SetLogWriter(new LogWriter(BuildLoggingConfig()));

    var uiPolicies = new List<ExceptionPolicyEntry>
    {
        new ExceptionPolicyEntry(
            typeof(Exception),
            PostHandlingAction.None,
            new IExceptionHandler[]
            {
                new LoggingExceptionHandler(
                    "General", 100,
}
```

```
                    TraceEventType.Error,
                    "Enterprise Library Exception Handling", 0,
                    typeof(TextExceptionFormatter), Logger.Writer)
            })
    };

    var unhandledPolicies = new List<ExceptionPolicyEntry>
    {
        new ExceptionPolicyEntry(
            typeof (Exception),
            PostHandlingAction.None,
            new IExceptionHandler[]
            {
                new LoggingExceptionHandler("General", 100,
                    TraceEventType.Error,
                    "Unhandled Policy Exception Handling", 0,
                    typeof(TextExceptionFormatter), Logger.Writer),
            })
    };

    var securityPolicies = new List<ExceptionPolicyEntry>
    {
        new ExceptionPolicyEntry(
            typeof (System.Security.SecurityException),
            PostHandlingAction.ThrowNewException,
            new IExceptionHandler[]
            {
                new LoggingExceptionHandler("General", 100,
                    TraceEventType.Error,
                    "Security Exception in Service Layer", 0,
                    typeof(TextExceptionFormatter), Logger.Writer),
                new ReplaceHandler("Unauthorized Access",
                    typeof(System.Security.SecurityException))
            })
    };

    var policies = new List<ExceptionPolicyDefinition>
        {
            new ExceptionPolicyDefinition("UI Policy",
                uiPolicies),
            new ExceptionPolicyDefinition("Unhandled Policy",
                unhandledPolicies),
            new ExceptionPolicyDefinition("Service Policy",
                securityPolicies)
        };

    ExceptionPolicy.SetExceptionManager(
        new ExceptionManager(policies));
}
```

These Exception Handling policies will handle Unhandled Exceptions and Service Exceptions. The Service Exception Policy has different parameters than the UI and Unhandled Policies since we want to handle it differently. Its exception type is System.Security.SecurityException and its Post Handling Action is Throw New Exception. Also, in addition to the Logging Handler, it requires a Replace Handler. This will throw a different exception than the caught Security Exception. Although you have kept the type the same, the new Security Exception will not provide the client with any of the stack information or internal security information that could compromise security.

To change the application to exit on a security exception

1. Add the following policies to the **BootstrapEnterpriseLibrary** method in the **Startup** class:

```
var uiPolicies = new List<ExceptionPolicyEntry>
{
    new ExceptionPolicyEntry(
        typeof(Exception),
        PostHandlingAction.None,
        new IExceptionHandler[]
        {
            new LoggingExceptionHandler(
                "General", 100,
                TraceEventType.Error,
                "Enterprise Library Exception Handling", 0,
                typeof(TextExceptionFormatter), Logger.Writer)
        }),
    new ExceptionPolicyEntry(
        typeof (System.Security.SecurityException),
        PostHandlingAction.NotifyRethrow,
        new IExceptionHandler[]
        {
            new LoggingExceptionHandler(
                "General", 100,
                TraceEventType.Error,
                "Security Exception in UI Layer", 0,
                typeof(TextExceptionFormatter), Logger.Writer),
        })
};
```

This policy will handle Security Exceptions thrown on the UI layer. The value of the **Post handling action** property is **NotifyRethrow**, which will cause the handler for the **Application.ThreadException** event to re-throw the exception. The re-thrown exception will be an unhandled exception, and the .NET Framework common language runtime (CLR) will be forced to shut down the application.

To test the Replace Handler

1. Select the **Debug | Start Without Debugging** menu command to run the application.
2. Type a nonsense (alphabetic) word into the **Word To Check** textbox (ensure that you have an error flashing), then click on the **Add Word** button.
3. Click **OK** when the "Unhandled exception" message appears.
4. Next, an application exception dialog appears. Click **Close the program**.
5. Open the event log to look at the events created. You can use the debugger to observe exactly what is happening and relate this to the Exception Handling Policies. This time there are three errors shown in the Event Log:
    - First a **SecurityException** is thrown in Dictionary.cs. This is caught by the service layer (DictionaryService.cs) which applies **Service Policy.** This will cause the exception to be written to the event log on the server (with all available information included) and then will capture a new replacement **SecurityException** (without specific stack information).
    - Second, the replacing **SecurityException** is caught by the Application **ThreadException** handler in Startup.cs. This applies **UI Policy**, which will write the exception to the event log on the client (the same machine in this case) and set the re-throw result to **true**, which allows your code to decide to re-throw the second **SecurityException**.
    - Third, the re-thrown **SecurityException** is caught by our AppDomain **UnhandledException** handler (it was thrown from outside the Application.Run) which applies **Unhandled Policy**. This will log the exception and display a message box informing us that there was an error in the application.

    The AppDomain **UnhandledException** handler does not consume exceptions, so the exception continues to pass to the runtime or debugger exception handler. This will cause a standard unhandled exception dialog box to be displayed.

To verify you have completed the exercise correctly, you can use the solution provided in the ex02\end folder.

More Information

For more information about the Exception Handling Application Block, see the documentation in the Enterprise Library 6 Developer's Guide and the Enterprise Library 6 Reference documentation.

patterns & practices
proven practices for predictable results

Copyright