Exception Handling Application Block Hands-On Labs for Enterprise Library

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:

- Lab 1: Logging Exceptions
- Lab 2: Exception Handling Strategies

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

Authors

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

- Product/Program Management: Grigori Melnik (Microsoft Corporation)
- Development: Chris Tavares (Microsoft Corporation), Nicolas Botto (Digit Factory), Olaf Conijn (Olaf Conijn BV), Fernando Simonazzi (Clarius Consulting), Erik Renaud (nVentive Inc.)
- Testing: Rick Carr (DCB Software Testing, Inc) plus everyone above
- Documentation: Alex Homer and RoAnn Corbisier (Microsoft Corporation) and Dennis DeWitt (Linda Werner & Associates Inc)

All of the Hands-On Labs use a simplified approach to object generation through Unity and the Enterprise Library container. The recommended approach when developing applications is to generate instances of Enterprise Library objects using dependency injection to inject instances of the required objects into your application classes, and thereby obtain all of the advantages that this technique offers.

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 installed with the Enterprise Library, or 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 | Add Reference** menu command. Select the **Browse** tab and select the following assembly located in the Enterprise Library **bin** folder (typically C:\Program Files\Microsoft Enterprise Library 5.0\Bin).
   - Microsoft.Practices.EnterpriseLibrary.Common.dll
   - Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.dll
   - Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging.dll
   - Microsoft.Practices.EnterpriseLibrary.ServiceLocation.dll

   While this assembly is the only assembly required for the ExceptionHandling API, other assemblies may need to be available in the bin\debug directory to provide specific exception handling functionality, which you will add later.
2. Select the **Puzzler.cs** file in the Solution Explorer. Select the **View | Code** menu command.
3. Add the following namespace inclusion at the top of the file:

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

4. 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)
{
```

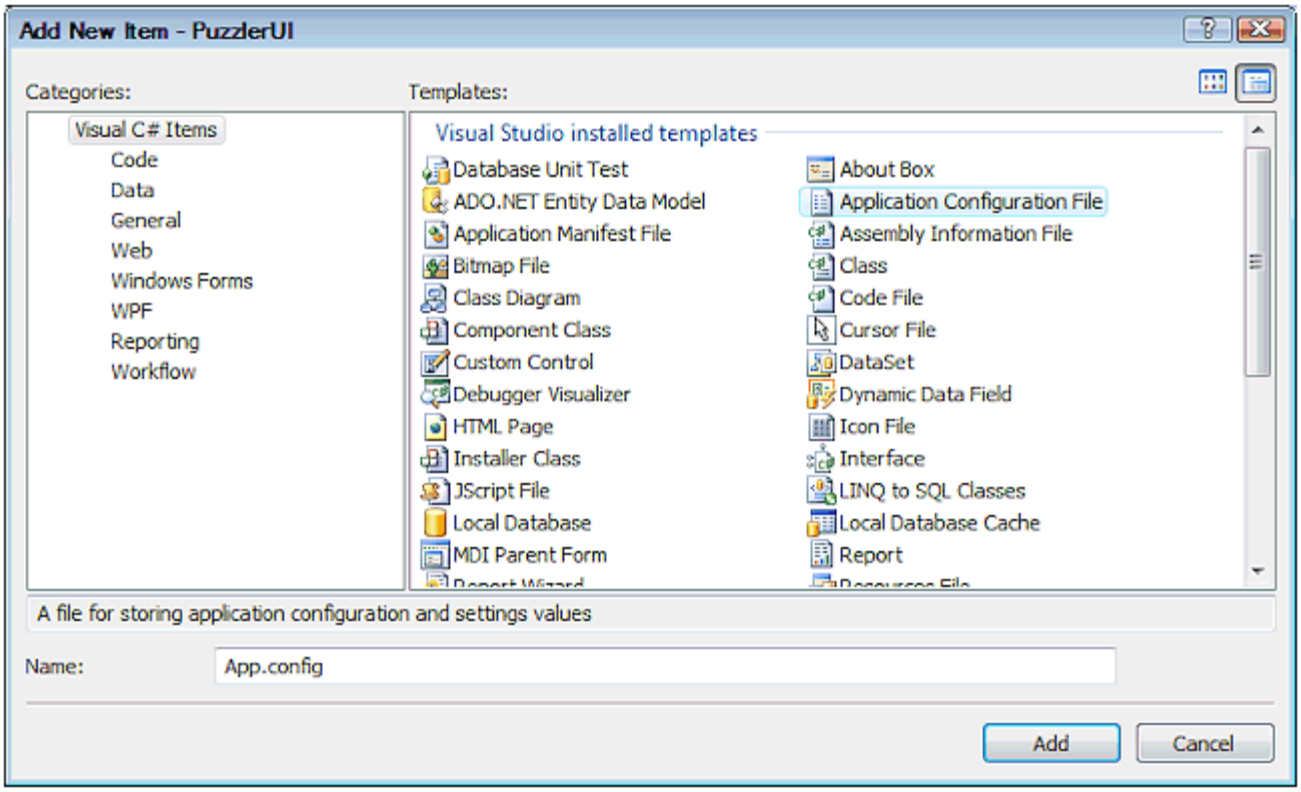
```
  try
  {
    // TODO: Handle exceptions
    PuzzlerService.Dictionary.AddWord(txtWordToCheck.Text);
    errorProvider1.SetError(txtWordToCheck, "");
  }
  catch (Exception ex)
  {
    bool rethrow = ExceptionPolicy.HandleException(ex, "UI Policy");
    if (rethrow)
      throw;

    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 using the Enterprise Library Configuration tool

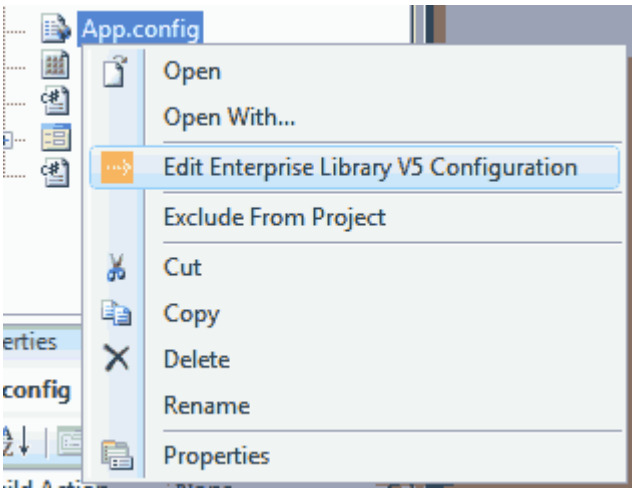
1. Add a new application configuration file (App.config) to the **PuzzlerUI** project.

   Click the **PuzzlerUI** project. Select the **Project| Add New Item** menu command. Select **Application configuration file** template. Leave the Name as **App.config**.



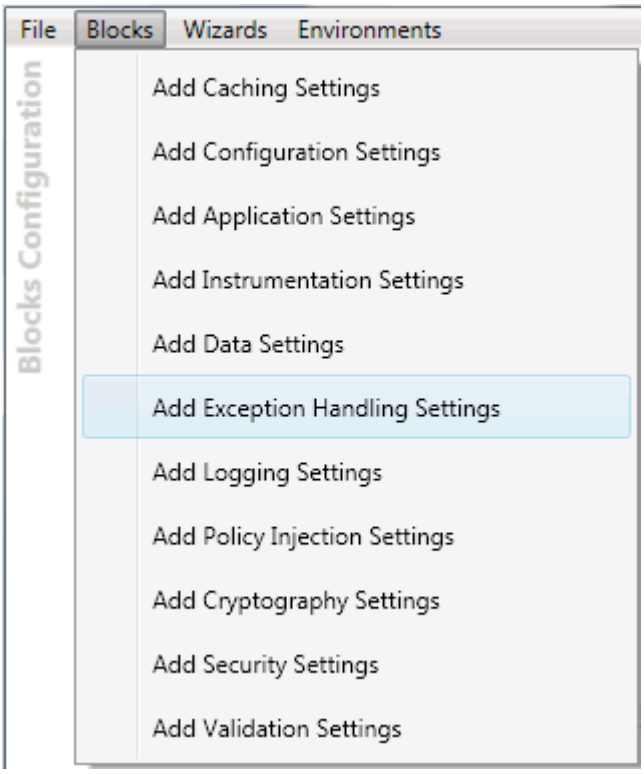
2. In Visual Studio, close the new App.config file.

3. You will use the Enterprise Library Configuration tool to configure your application. You may either start the stand-alone version of the tool or use the version integrated in Visual Studio. To use the stand-alone version, select **Start |All Programs | Microsoft patterns and practices | Enterprise Library 5.0 | Enterprise Library Configuration**, select the appropriate version of the configuration tool, and open the App.config file. To use the Visual Studio integrated configuration editor, right-click the App.config file in Solution Explorer and select **Edit Enterprise Library V5 Configuration** from the context menu.

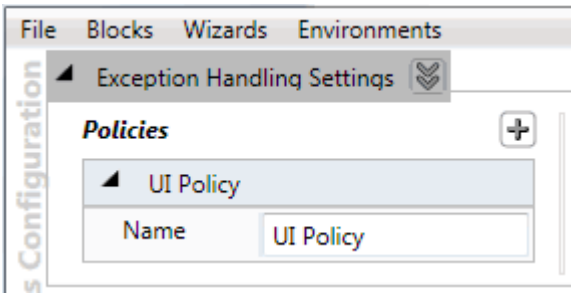


To configure the application to use exception management

1. Open the Blocks menu and select **Add Exception Handling Settings**.

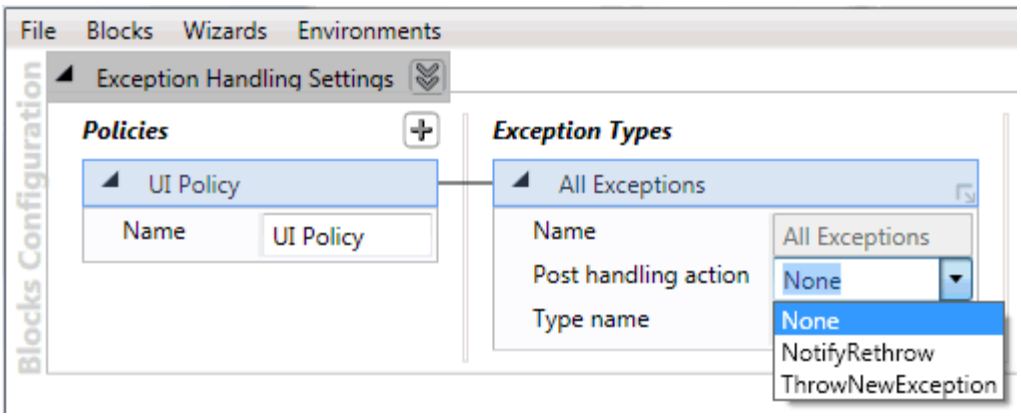


2. Click the arrow to the left of the title bar for the default **Policy** that the configuration tool automatically creates to show the properties of the policy. Change the value of the **Name** property to **UI Policy**.



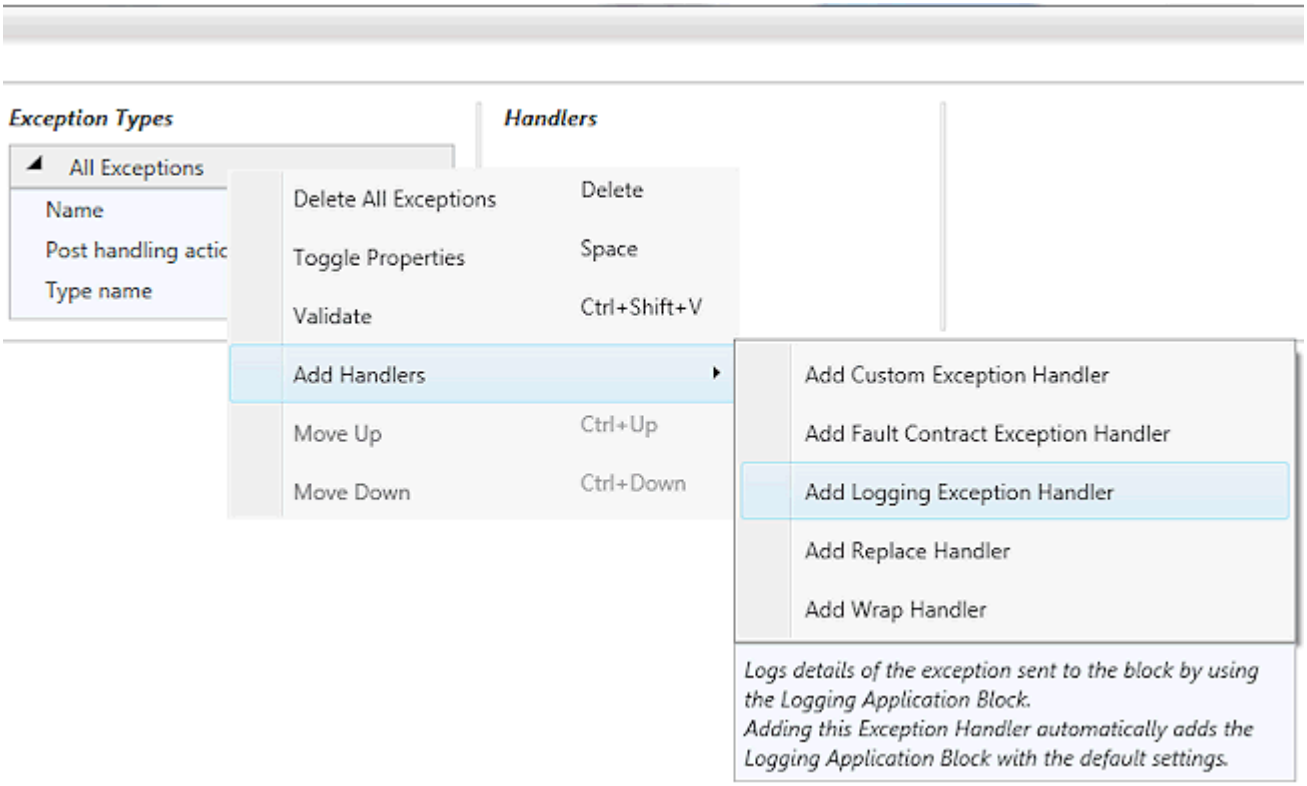
The policy name you enter here must match that specified in the code. Typically, you would use constants to help prevent typographical errors, especially since the exception handling typically is not tested as thoroughly as normal code paths.

3. The **Exception Types** column in the configuration tool contains the list of exception types that the policy will handle. The default policy defines a single exception type named All Exceptions that specifies the **System.Exception** type (and will therefore match any exception types that inherit **System.Exception**). Click the arrow to the left of the **All Exceptions** title to show the properties, and change the value of the **Post handling action** property to **None**.



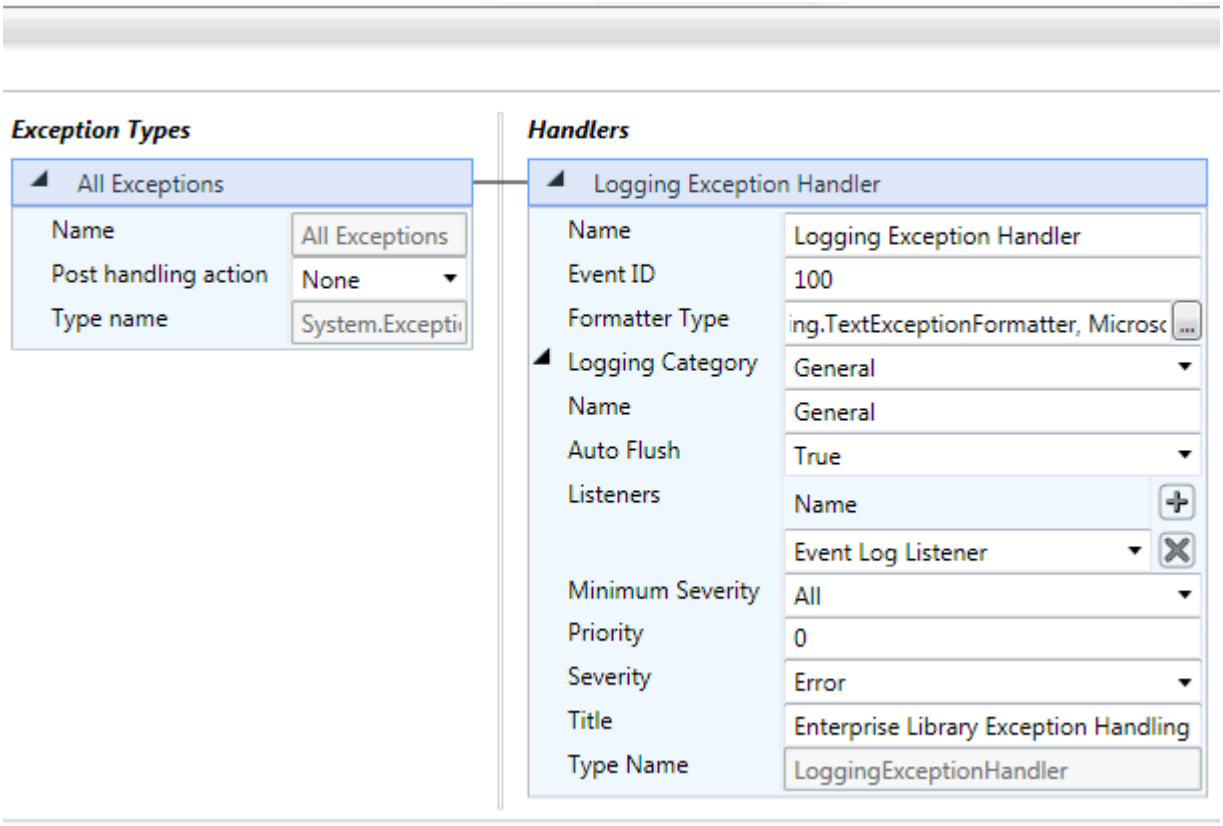
This will cause all exceptions to be handled internally within the exception handling code, and the caller will not be requested to re-throw the exception with its catch block.

4. Now you will add an exception handler to the policy. Right-click the title bar of the **All Exceptions** item, point to **Add Handlers**, and click **Add Logging Exception Handler** in the context menu.



5. This automatically adds a Logging Exception Handler to the **Exception Handling Settings** section to your configuration, and adds the **Logging Settings** section with the default configuration for the Logging Application Block. Click the arrow to the left of the **Logging Exception Handler** title to show the properties for the handler. By default, the handler will use a Text Exception Formatter, and write log messages to the **General** category.

6. Click the arrow to the left of the **Logging Category** property to show the settings for this category in the configuration of the Logging Application Block.



   Alternatively, you can examine the settings in the **Logging Settings** section of the configuration tool.

7. Save and close the application configuration.

To change the application to include the exception logging assembly

1. Select the **PuzzlerUI** project. Select the **Project | Add Reference** menu command. Select the **Browse** tab and select the following assembly located in the Enterprise Library **bin** folder (typically C:\Program Files\Microsoft Enterprise Library 5.0\Bin).
   - Microsoft.Practices.EnterpriseLibrary.ExceptionHandling.Logging.dll

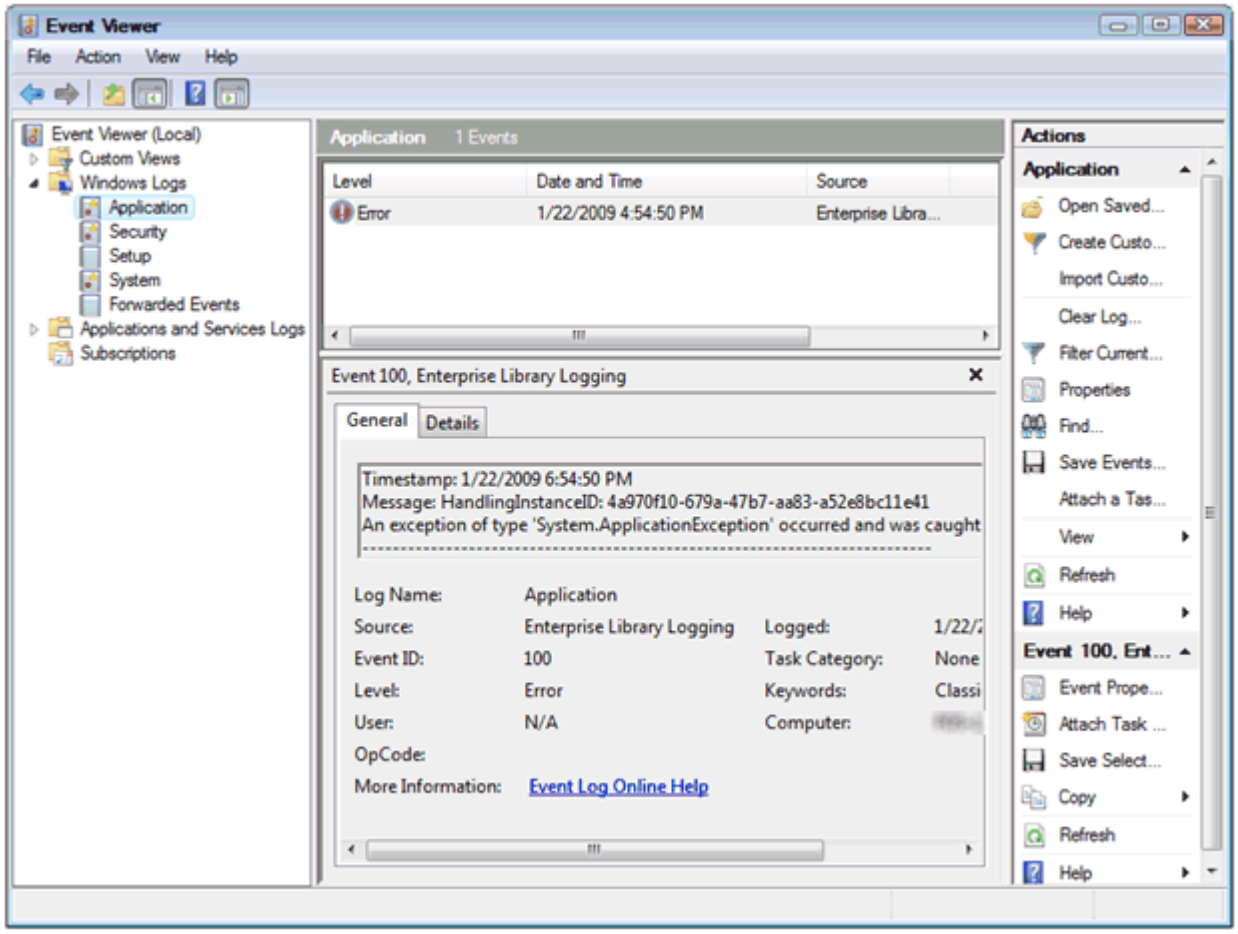
   Because one of the goals of the Enterprise Library is to keep the application blocks decoupled, it is possible to use the Exception Handling Application Block without needing the Logging Application Block (e.g. by creating your own exception handler). If you want to use the two application blocks together, then you need to use this assembly, which contains an Exception Handler that logs via the Logging Application Block.

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.



   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.

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 **Process** 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 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. Add the following namespace inclusions at the top of the file:

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

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

```
public static void HandleException(Exception ex, string policy)
{

  Boolean rethrow = false;

  // Resolve an Exception Manager instance
  var exManager
    = EnterpriseLibraryContainer.Current.GetInstance<ExceptionManager>();

  try
  {
    rethrow = exManager.HandleException(ex, policy);
  }
  catch (Exception innerEx)
  {
    string errorMsg = "An unexpected exception occured 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 (e.g. 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 System.Exception)
  {
    HandleException((System.Exception)e.ExceptionObject, "Unhandled Policy");
  }
}
```

This handler will use a new policy, named **Unhandled Policy**, which will 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()
{
  // TODO: Handle unhandled exceptions
  Application.ThreadException +=
        new ThreadExceptionEventHandler(Application_ThreadException);

  AppDomain.CurrentDomain.UnhandledException +=
        new UnhandledExceptionEventHandler(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 our 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

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:

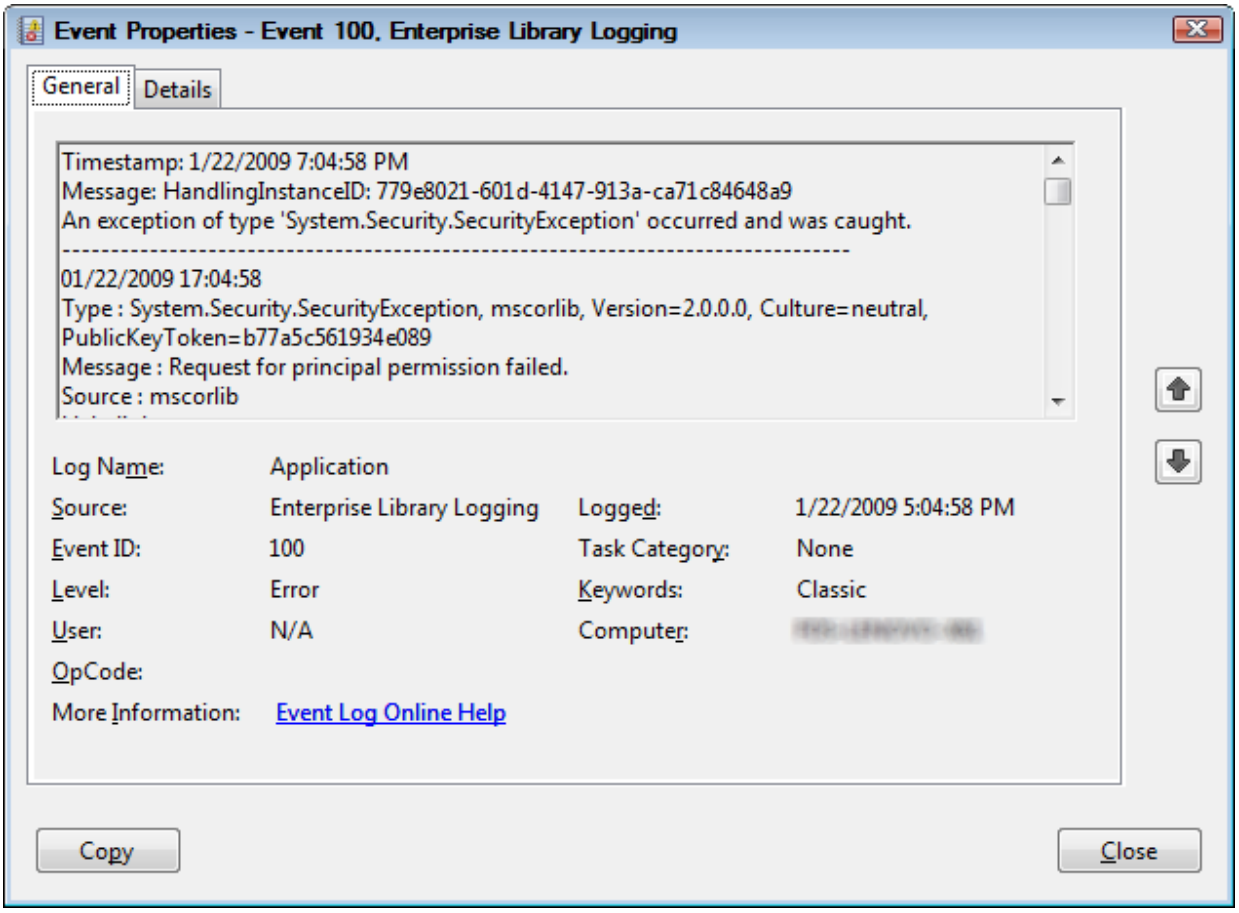
```
// TODO: Add security attribute
[PrincipalPermission(SecurityAction.Demand, Role = "Grand PoohBah")]
public static Boolean AddWord(string wordToAdd)
{
    if (!IsWord(wordToAdd))
    {
        // It is not alphabetic! Throw an exception
        throw new ApplicationException(
            "Word to 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.



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

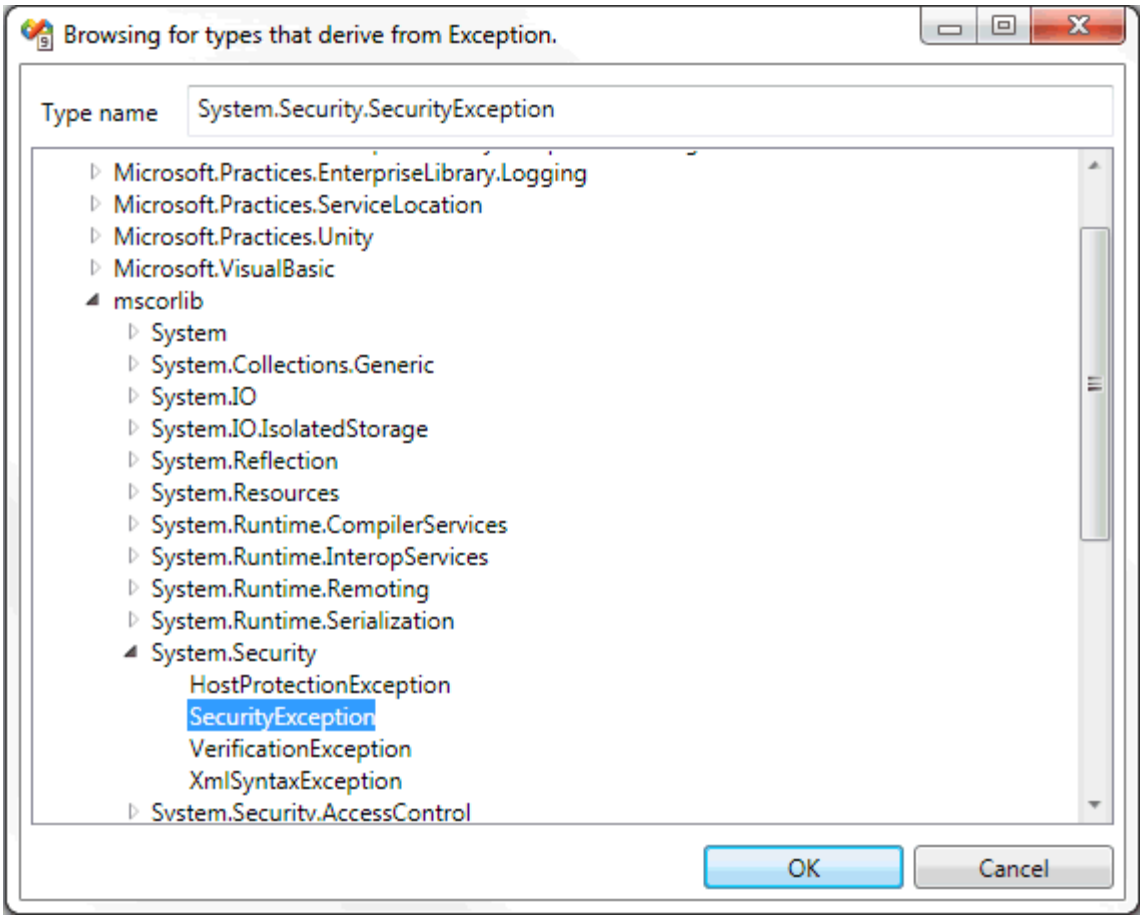
3. Close the application.

To configure the application to replace security exceptions

1. Select the **PuzzlerUI** project **App.config** file in the Solution Explorer. Select the **Edit Enterprise Library V5 Configuration** menu command from the context menu. Then click the arrow to the left of the **Exception Handling Settings** title bar to expand this section to show the exception handling policies it contains.

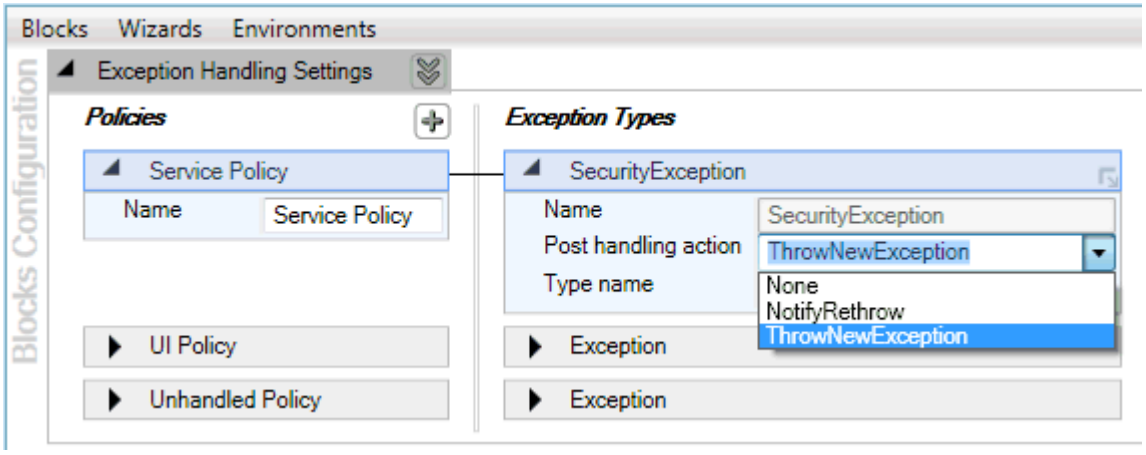
   The App.config file already has an empty policy, named **Service Policy**. By default, if a policy is empty, then exceptions will just be re-thrown inside the catch block, so in effect the policy will do nothing.

2. Right-click the title of the **Service Policy** item and click **Add Exception Type**. In the type selector dialog, expand the **mscorlib** node, expand the **System.Security** node, and select **SecurityException**. Then click **OK**.

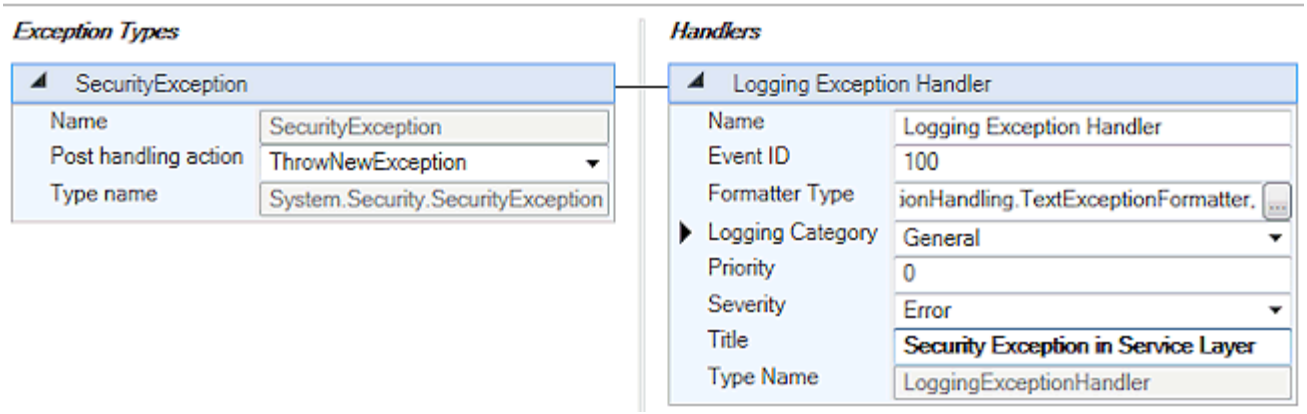


Note: You can use the Type name text box to filter the list and easily find the required type.

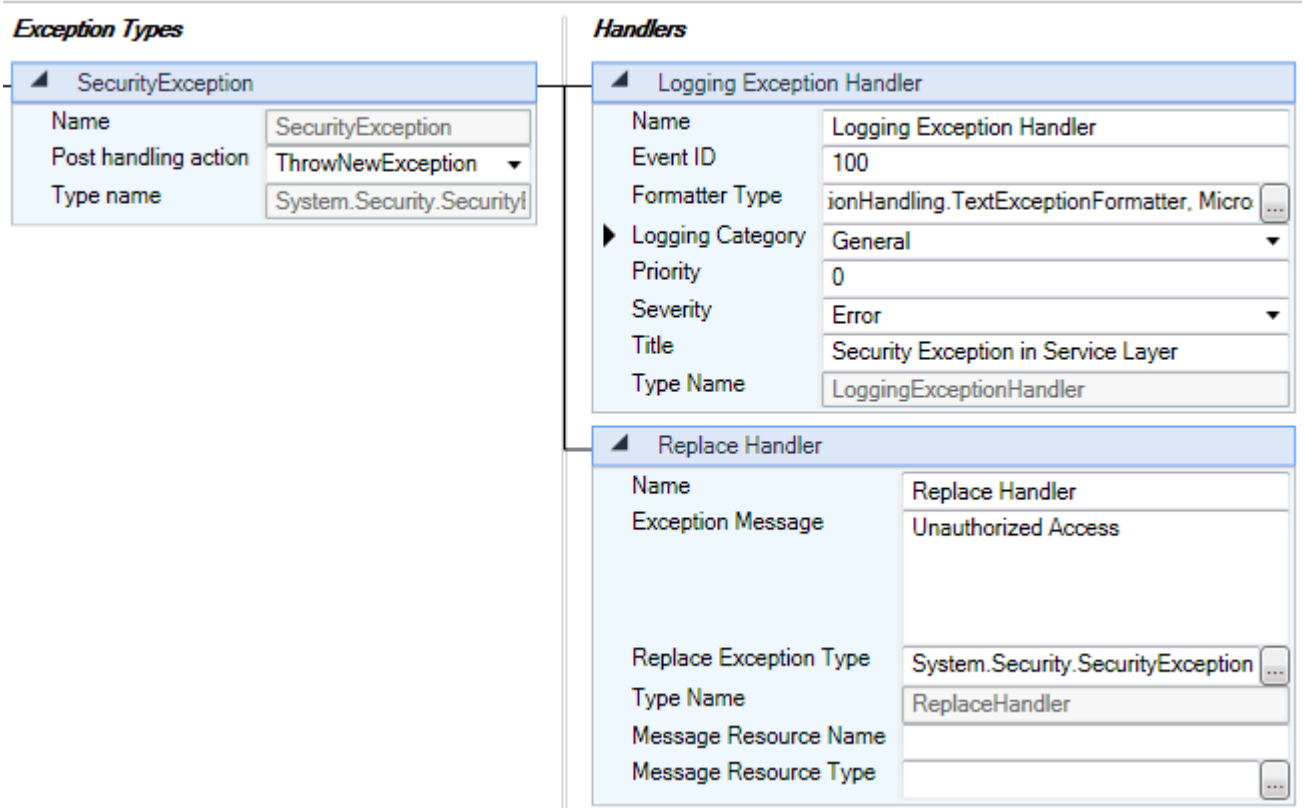
3. Change the value of the **Post handling action** property for the exception type to **ThrowNewException**.



4. Add a new **Logging Handler**. Right-click the title of the **SecurityException** item, point **to Add Handlers**, and click **Add Logging Exception Handler**. Change the value of the **Title** property to **Security Exception in Service Layer**.



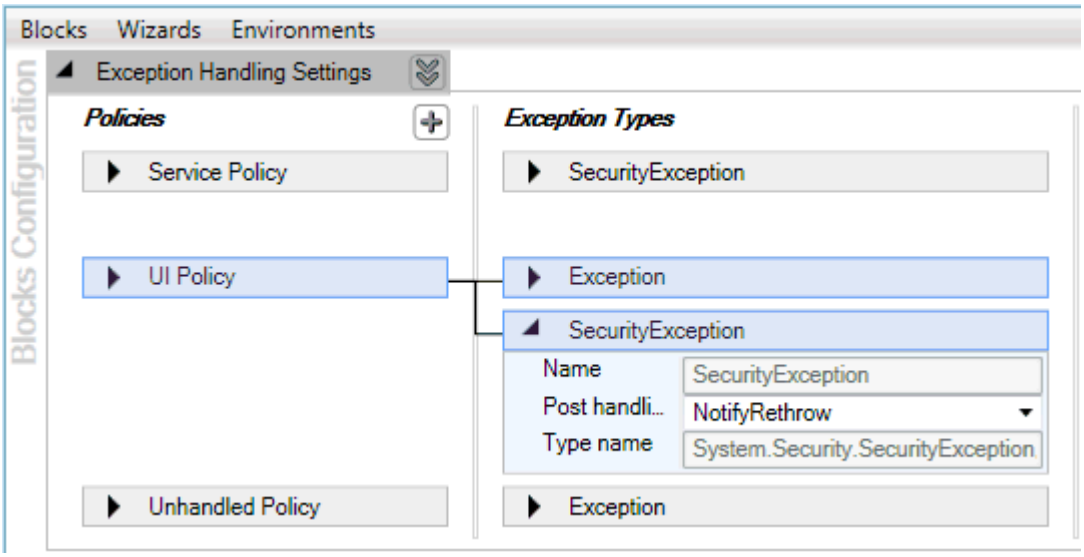
5. Add a new **Replace Handler**. Right-click the title of the **SecurityException** item, point **to Add Handlers**, and click **Add Replace Handler**. Set the value of the **Exception Message** property to **Unauthorized Access**. Then click the ellipses (**...**) button for the **Replace Exception Type**. In the type selector dialog, expand the **mscorlib** node, expand the **System.Security** node, and select **SecurityException**. Then click **OK**.



Although you have kept the type the same, the new SecurityException will not provide the client with any of the stack information or internal security exception information; which could compromise security.

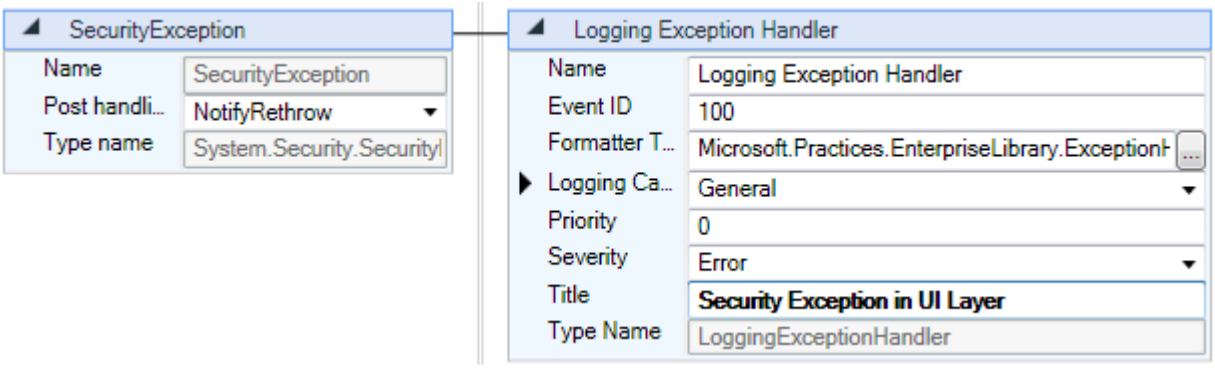
To change the application to exit on a security exception

1. Right-click the title of the **UI Policy** item and click **Add Exception Type**. In the type selector dialog, expand the **mscorlib** node, expand the **System.Security** node, and select **SecurityException**. Then click **OK**.



The default 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.

2. Add a new **Logging Handler** to the **SecurityException** exception type for the **UI Policy**. Right-click the title of the **SecurityException** item, point **to Add Handlers**, and click **Add Logging Exception Handler**. Change the value of the **Title** property to **Security Exception in UI Layer**.



3. Save and close the application configuration.

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.

Copyright