A robust and well-planned exception handling strategy is a vital feature of your application design and implementation that will help you avoid risks such as exposing error messages containing sensitive information, or leaving your application and systems in an inconsistent state and open to attack when an error occurs.

An exception handling strategy consists of a series of policies that define how you will present clear and appropriate messages to users and how you can provide assistance for operators, administrators, and support staff. A comprehensive exception-handling strategy will usually:

* Notify the user with a friendly message
* Store details of the exception in a production log or other repository
* Alert the customer service team to the error
* Assist support staff in cross-referencing the exception and tracing the cause

The Exception Handling Application Block can help you to define and create consistent exception management strategies by implementing three well-known design patterns:

* **Exception Shielding**. This pattern ensures that your application does not leak sensitive information, no matter what run-time or system event may occur to interrupt normal operation. And on a more granular level, it can prevent your assets from being revealed across layer, tier, process, or service boundaries.
* **Exception Logging**. This pattern can help you to diagnose and troubleshoot errors, audit user actions, and track malicious activity and security issues by logging details of exceptions and errors that occur.
* **Exception Translation**. This pattern describes how you can wrap exceptions within other exceptions specific to a layer to ensure that they actually reflect user or code actions within the layer at that time, and not some miscellaneous details that may not be useful.

The Exception Handling Application Block lets you associate exception types with named policies. You do this by using the configuration tools. Policies specify the exception handlers that execute when the block processes a particular exception type. Exception handlers are .NET classes that encapsulate exception handling logic and implement the Exception Handling Application Block interface named **IExceptionHandler**.

You can chain these handlers together so that a series of them execute when the associated exception type is handled. The following are some examples of named policies and descriptions of what they might provide:

* **Base policy**. This policy logs the exception and re-throws the original exception.
* **Secure policy**. This policy logs the exception, replaces the original exception with a custom exception, and throws the new exception.
* **Expressive policy**. This policy wraps the original exception inside another exception and throws the new exception.

The following schematic illustrates examples of cross-layer and single-application component exception handling.

Diagram

Description automatically generated

**Figure 1**

Examples of exception handling policies

In this example, exceptions that occur in the data access layer are logged and then wrapped inside another exception that provides more meaningful information to the calling layer. Within the business component layer, the exceptions are logged before they are propagated. Any exceptions that occur in the business component layer and that contain sensitive information are replaced with exceptions that no longer contain this information. These are sent to the user interface (UI) layer and displayed to the user.

For information about how to develop exception management strategies, see [Determining Appropriate Exception Policies and Actions](https://documentation.help/MS-Enterprise-Library-5.0/EntLib50_9c728a58-1aad-475e-a3ca-85c4618128c4.html).

**Example Application Code**

Without the Exception Handling Application Block, the typical exception handling code for a method of a data access component might look like the following. The code below does not include implementations of the custom **DataAccessException** exception type, or the **RunQuery**, **FormatException**, and the **Logging.Log** methods. These methods represent typical ways to retrieve a DataSet and to log information.

|  |  |
| --- | --- |
| **C#** | **Copy Code** |
| public DataSet GetDataSet(string queryName)  {  try  {  return RunQuery(queryName);  }  catch(Exception ex)  {  string formattedInfo = FormatException(ex);  Logging.Log(formattedInfo);  throw new DataAccessException("Database access failure for query" + queryName, e);  }  } | |
| **Visual Basic** | **Copy Code** |
| Public Function GetDataSet(ByVal queryName As String) As DataSet  Try  Return RunQuery(queryName)  Catch ex As Exception  Dim formattedInfo As String = FormatException(ex)  Logging.Log(formattedInfo)  Throw New DataAccessException("Database access failure for query " & queryName, e)  End Try  End Function | |

Code similar to that shown in the **catch** section would be repeated in all the routines that perform different data access queries. To change the behavior of the exception handling code, you must update each routine that contains this code.

However, with the Exception Handling Application Block the same application could use the following code to define the method and then execute it. It assumes you have resolved the class through the container to obtain a reference to an **ExceptionManager** instance and stored it in a variable named **exManager**.