# Exception Logging using **`clsErrorHandling`** class of XForce Common Library by SSM Infotech Solutions Pvt Ltd

Proper exception handling is critical for any application. A key component to that is logging the exceptions to a logging library so that you can record that the exceptions occurred. Every exception in your app should be logged. They are critical to finding problems in your code! It is also important to log more contextual details that can be useful for troubleshooting an exception. Things like what customer was it, key variables being used, etc.

## **`clsErrorHandling`** class of **`SSM.XForce.CommonLibrary`**

`clsErrorHandling` **class** is developed specially to create logs for exceptions occurred in our application. This class is available under `SSM.XForce.CommonLibrary` **namespace** of SSM.XForce.CommonLibrary.dll **reference dll file** which is designed and developed by SSM Infotech Solutions Pvt Ltd. By using this class, we can **generate exception log files** for our application.

### Properties of **`clsErrorHandling`** class

1. `ApplicationPath (string)`
   Used to store/retrieve directory path from where application (.exe) is running.
2. `ApplicationPriority (int)`
   Used to store/retrieve the priority of application. Exceptions with `Priority` lower than `ApplicationPriority` will be logged into log files. This property is used to define the detailing level of log file of your application. If `ApplicationPriority` is Nine then it will create log for all the messages from most critical error messages to general information type messages.
3. `AppName (string)`
   Used to store/retrieve the name of your application.
4. `ErrDesc (string)`
   Used to store/retrieve the custom description of the error (exception).
5. `ErrorTraceDetails (string)`
   Used to store/retrieve the StackTrace of the error (exception).
6. `IsDebugMode (bool)`
   Used to store/retrieve whether application is running in debug mode or not.
7. `LogFilePath (string)`
   Used to store/retrieve log file path.
8. `MethodName (string)`
   Used to store/retrieve the name of the method (from StackTrace) which causes the exception.
9. `ModName (string)`
   Used to store/retrieve the name of the module (from StackTrace) which causes the exception. Generally module is the name of the class file.
10. `MsgType (string)`
    Used to store/retrieve the type of the error message. There are four message types defined in enum `clsGeneral.MessageType`: Errors, Info, Trace and Warning
11. `Priority (int)`

Used to store/retrieve the priority of the exception. There are 10 priority levels defined in enum `clsGeneral.Priority`: Zero, One… Nine. Priority Nine is the lowest priority and priority zero is the highest priority. Priority is used to show the criticality of the error. Error message with priority Nine can be consider as general information type messages and Error message with priority Zero can be considered as most critical error message.

12. `UserName (string)`
    Used to store/retrieve the windows' current user name. We can use `Name` property from method `GetCurrent()` of the class `WindowsIdentity` which is defined in the namespace `System.Security.Principle`.

## Methods of `clsErrorHandling class`

1. `public void ErrorHandle()`
   This method will call appropriate method to handle exception and create exception log files. These methods are defined in `clsGeneral`.

## Error handling methods of `clsGeneral` class

The class `clsGeneral` contains 2 overloads of `ErrorHandle()` method as listed below:

1. `public void ErrorHandle(SSM.XForce.CommonLibrary.clsGeneral.MessageType pMsgType, SSM.XForce.CommonLibrary.clsGeneral.Priority pPriority, string pstrMessage)`
   This method has 3 parameters and returning nothing:
   a. `pMsgType:` Message type of the error.
   b. `pPriority:` Priority of the error.
   c. `pstrMessage:` Custom error message to print in the log file.

   This method will create a text log file **hourly** under **Log directory** in the directory path specified by `ApplicationPath` property of the `clsErrorHandling` if it is not exist. If a text log file is already been created on the same path then this method will append the log text to existing log file. The name of the log file is in the format **YYMMDDHH.txt** i.e. this method will create new log file for every hour of the current date. Example: **21010809.txt.**

   This method will create log text as per following format:

   `DateTime.Now, AppName, AssemblyVersion, CultureInfo, PublicKeyToken, ModName, MethodName, pMsgType, pPriority, pstrMessage, UserName, FileLineNumber`

   Example of log text:

   08/01/2021 08:27:16 AM, ConsoleApp1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null, Program, Main, ERROR, 3, Attempted to divide by zero., SSMINFOTECH\dhaval.rangrej, 0

   If error priority `pPriority` is greater than application priority `ApplicationPriority` then this method will ignore the message i.e. it will not entered into log file.

2. `public void ErrorHandle(SSM.XForce.CommonLibrary.clsGeneral.MessageType pMsgType, SSM.XForce.CommonLibrary.clsGeneral.Priority pPriority, System.Exception pExError, [string pstrMessage = ])`
   This method is overloaded version of previous method. It has 4 parameters:
   a. `pMsgType`: Message type of the error.
   b. `pPriority`: Priority of the error.
   c. `pExError`: Object of any exception for which we are making log.
   d. `pstrMessage`: Optional parameter and used for custom error message. It's empty string by default.

   This method will also write error log in text log file with name as YYMMDDHH.txt format in same manner as previous method. But this method is used to create a more detailed error logs in separate file with the name as Log_YYMMDDHH.txt format. Example: **Log_21010809.txt.** This file will be created in hourly manner on the same path as described in previous method i.e. `ApplicationPath`\Log\. If the file doesn't exist then this method will create a new file and write the error log in the same. If file exists then it will append the error log in that file.

   When exception occurs in our source code and in catch block we pass object of that exception in ErrorHandle() method with other required arguments then this version of the ErrorHandle() method will be called. This method will print the **`ErrorTraceDetails`** of the exception in Log_YYMMDDHH.txt file. This method will retrieve all required information from the exception object for exception logging.

   Example of log details:

   at ConsoleApp1.Program.Main(String[] args) in C:\Users\Dhaval.Rangrej\source\repos\ConsoleApp1\Program.cs:line 176

Class `clsErrorHandling` is designed for multithreaded applications. That means when a process is writing the a log file then it will be locked and at the same time if other process tries to write the same file then it must wait while file is being unlocked. So, we can call this methods on different threads for exception logging.

*Example:*

```
using System;
using SSM.XForce.CommonLibrary;
using System.IO;
using System.Diagnostics;
using System.Reflection;
namespace ExceptionLoggerDemo
{
      // Create a class for error logging
      public class ErrorLogger
      {
         // Create object of clsErrorHandling to set its properties and call the
         // method ErrorHandle()
         clsErrorHandling errorHandler = new clsErrorHandling();
         // Create a method to overload first version of ErrorHandle() method
```

```
   public void Log(clsGeneral.MessageType pMsgType, clsGeneral.Priority
pPriority, string pstrMessage)
   {
      StackTrace stTrace = new StackTrace();
      StackFrame stFrame = stTrace.GetFrame(1);
      MethodBase methodBase = stFrame.GetMethod();

      errorHandler.ApplicationPath = Path.GetDirectoryName(
Assembly.GetExecutingAssembly().Location);
      errorHandler.AppName = Assembly.GetExecutingAssembly().FullName;
      errorHandler.ModName = methodBase.ReflectedType.Name;
      errorHandler.MsgType = pMsgType.ToString();
      errorHandler.Priority = int.Parse(pPriority);
      errorHandler.ApplicationPriority = 9;
      errorHandler.ErrorDesc = pstrMessage;
      errorHandler.ErrorTraceDetails = "";
      errorHandler.IsDebugMode = false;
      errorHandler.MethodName = methodBase.Name;
      errorHandler.UserName = System.Security.Principle.WindowsIdentity
.GetCurrent().Name;
      errorHandler.ErrorHandle();
   }
   // Create a method to overload second version of ErrorHandle() method
   public void Log(clsGeneral.MessageType pMsgType, clsGeneral.Priority
pPriority, Exception pExError, string pstrMessage="")
   {
      StackTrace stTrace = new StackTrace();
      StackFrame stFrame = stTrace.GetFrame(1);
      MethodBase methodBase = stFrame.GetMethod();

      errorHandler.ApplicationPath = Path.GetDirectoryName(
Assembly.GetExecutingAssembly().Location);
      errorHandler.AppName = Assembly.GetExecutingAssembly().FullName;
      errorHandler.ModName = methodBase.ReflectedType.Name;
      errorHandler.MsgType = pMsgType.ToString();
      errorHandler.Priority = int.Parse(pPriority);
      errorHandler.ApplicationPriority = 9;
      errorHandler.ErrorDesc = pExError.Message;
      errorHandler.ErrorTraceDetails = pExError.StackTrace;
      errorHandler.IsDebugMode = false;
      errorHandler.MethodName = methodBase.Name;
      errorHandler.UserName = System.Security.Principle.WindowsIdentity
.GetCurrent().Name;
      errorHandler.ErrorHandle();
   }
}
class Program
{
    //Create an object of ErrorLogger
    //This object can be used among all methods in this class
    ErrorLogger logger = new ErrorLogger();
    static void Main()
```

```
        {
            //Call Log method of logger object to create an information type
            //of log with low priority
            logger.Log(clsGeneral.MessageType.Info, clsGeneral.Priority.Six,
    "Program Started");
            try
            {
                Console.WriteLine("ENTER ANY TWO NUBERS");
                int a = int.Parse(Console.ReadLine());
                int b = int.Parse(Console.ReadLine());
                int c = a / b;
                Console.WriteLine("C VALUE = " + c);
            }
            catch(Exception ex)
            {
          //Call Log method of logger object to create an error type
          //of log with highest priority
                logger.Log(clsGeneral.MessageType.Errors,
    clsGeneral.Priority.Zero, ex.Message);
                logger.Log(clsGeneral.MessageType.Errors,
    clsGeneral.Priority.Zero, ex, ex.Message);
            }
            logger.Log(clsGeneral.MessageType.Info, clsGeneral.Priority.Six,
    "Program Ended");
        }
    }
}
```