

CodePlex.Diagnostics

Version 1.0.0.95

Introduction

Welcome to the first release of the **CodePlex.Diagnostics** framework, designed to enable exception publication and logging through components based upon the ASP.NET 2.0 provider design pattern.

Using this framework you can write your applications and defer the decision of which exception providers will be used to publish exceptions until deployment time. In this release, the **SqlExceptionProvider** publishes the exceptions to a SQL Server 2005 database. The database T-SQL creation script is included within the **CodePlex.Diagnostics** Visual Studio 2005 solution.

Use of SQL Server 2005, as opposed to previous releases, is important because it allows the exceptions to be serialized using the **SoapFormatter** class. The Soap encoded exceptions are then stored within the database using the new **Xml** column type. Columns are included within the **Exception** table for most properties within the base **System.Exception** type but the serialized exceptions are there to allow a developer to browse through any additional information provided by exceptions that are either directly or indirectly derived from the **System.Exception** base type. In most cases the properties within **System.Exception** have corresponding columns within the **Exception** table; the **TargetSite** property however has a corresponding **TargetSite** table.

Central to the **CodePlex.Diagnostics** framework is the static **ExceptionProvider** class shown in the following Visual Studio 2005 class diagram. As you can see, the **ExceptionProvider** contains a single method which requires two arguments. The first argument is the exception to be published and the second argument is the **IIdentity** representing the user on whose behalf the code was executing. Within a Windows Forms application this is usually the **WindowsIdentity** of the current user, however within an ASP.NET application an **IIdentity** representing the user who is authenticated to the Web site should be provided.



Figure 1. **ExceptionProvider** class.

Another central type within the **CodePlex.Diagnostics** framework is the static **LoggingProvider** class, shown in the next class diagram, which provides logging capabilities. As with the publication of exceptions, the **LoggingProvider** class is designed to allow the decision of where log entries should be persisted to be deferred until deployment time. The default **SqlLoggingProvider** class persists log entries within the same SQL Server 2005 database.



Figure 2. **LoggingProvider** class.

When using the static **LoggingProvider** class, the log entry is simply a **System.String**, however you must also determine the priority of the entry and the type of information the entry provides. The **LoggingPriority** enumeration allows you to indicate that the entry is of **Low**, **Medium**, **High**, or **Critical** priority. Additionally, using the **LoggingType** enumeration, the log entry can be described as **Information**, **Warning**'s or an **Error**. The fourth argument that the **Log** method expects is the **IIdentity** which, as with the **ExceptionHandler**, represents the user on whose behalf the code was executing.



Figure 3. **LoggingPriority** and **LoggingType** enumerations.

Default Providers

The **CodePlex.Diagnostics** framework contains default exception and logging provider classes although you are free to construct other exception and logging providers, and then configure the framework accordingly. The default exception and logging providers are the **SqlExceptionHandler** and **SqlLoggingProvider** classes, these providers, as mentioned previously, store the exceptions and log entries within a SQL Server 2005 database. Figure 4 provides a Visual Studio 2005 class diagram depicting these two classes.

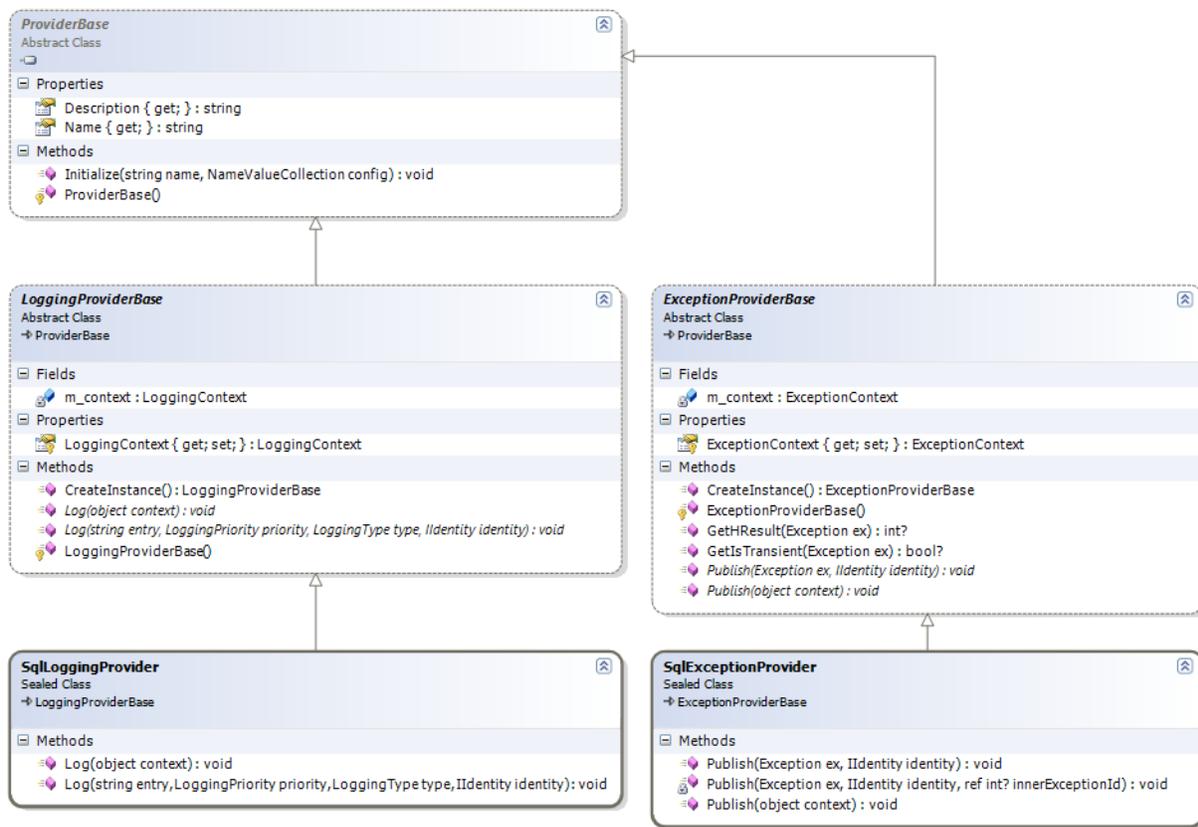


Figure 4. **SqlExceptionHandler** and **SqlLoggingProvider** types.

Within the **System.Configuration** assembly resides the **System.Configuration.Provider** namespace which is where you'll find the **ProviderBase** abstract base class shown in figure 2. The **ProviderBase** class forms the foundation for the ASP.NET 2.0 Provider design pattern, upon which this frameworks design is based. Subsequently, the **LoggingProviderBase** and **ExceptionProviderBase** abstract classes allow you to define specific providers such as the aforementioned default providers.

Configuration

Configuration of the **CodePlex.Diagnostics** framework is achieved using the application domain's configuration file (e.g. **App.config** or **Web.config**). Some additional sections, within which, are required to allow the framework to determine the correct exception and logging providers at runtime. For an example of the Xml configuration sections required by the framework you should examine the **App.Config** from the unit test project **CodePlex.Diagnostics.External.UnitTests**.

Although these additional Xml sections are required within the **App.Config** or **Web.Config** for the framework to determine the correct providers, it is also permissible to omit them entirely. Essentially, the framework is designed to operate in a *'fire and forget'* mode which implies that if the publication of an exception or log entry fails then the calling code itself will be unaware of this failure. This is by design, because at no time should the use of this library inhibit the calling code itself.

Assuming you have access to the Visual Studio 2005 Team Developer or Visual Studio 2005 Team Suite editions then it is suggested that you use the various unit test projects included within the **CodePlex.Diagnostics** solution. Once Visual Studio 2008 is released the Professional edition will also contain support for unit testing.

These unit test projects show the intended use of the various types defined within the library and the **App.config** from the unit test project is a good starting point for using the framework within your own projects. If you have other versions of Visual Studio 2005 (including the express editions), you'll still be able to use the framework although some of the projects within the solution might not function due to the limitations within the particular edition you are using.

Once you've installed the **CodePlex.Diagnostics** library, using the setup program that is included with the solution, the **CodePlex.Diagnostics** library will appear as an option within the Visual Studio 2005 / Visual Studio 2008 "Add Reference" dialog (see figure 5).

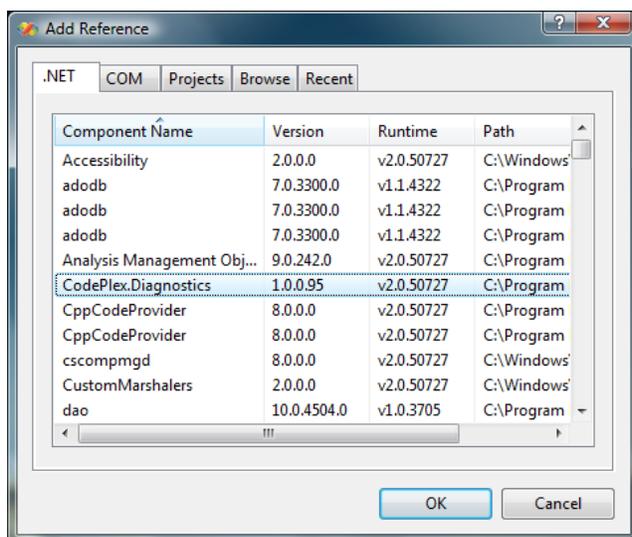


Figure 5. Visual Studio 2008 *Add Reference* dialog showing the **CodePlex.Diagnostics** library.

Assuming you create a C# Console Application and add a reference to the **CodePlex.Diagnostics** library, then you can then use the following code to test the publication of exceptions and logging events to the supplied **CodePlex.Diagnostics** database. If you haven't done so already, run the database creation script that is included within the source code for the library within SQL Server 2005 Management Studio.

```
using System;

using System.Security;
using System.Security.Principal;

using CodePlex.Diagnostics;
using CodePlex.Diagnostics.Providers;

namespace CodePlex.Diagnostics.Samples
{
    public static class Program
    {
        #region private static void Main(string[] args)

        /// <summary>
        /// Program entry point.
        /// </summary>
        /// <param name="args">
        /// An array of <see cref="T:System.String"/> containing the command line arguments.
        /// </param>
        [STAThread]
        private static void Main(string[] args)
        {
            try
            {
                for (int index = 0; index < 10; index++)
                {
                    index /= index;
                }
            }
            catch (DivideByZeroException ex)
            {
                IIdentity identity = WindowsIdentity.GetCurrent() as IIdentity;
                ExceptionProvider.Publish(ex, identity);
            }

            Console.ReadLine();
        }

        #endregion
    }
}
```

Listing 1. Sample C# Console Application to publish an exception using the static **ExceptionProvider** class.

You'll also need to add the appropriate Xml nodes to the **App.config** which can be copied from the **App.config** within the **CodePlex.Diagnostics.External.UnitTests** project, remembering to change the SQL connection string to reference the server upon which you installed the **CodePlex.Diagnostics** database.

Using SQL Server 2005 Management Studio (or other similar tools), you'll observe the data that was collected when the **DivideByZeroException** was caught and subsequently published to the **CodePlex.Diagnostics** database using the **ExceptionProvider** class.

```

USE [CodePlex.Diagnostics]

SELECT * FROM Exception
SELECT * FROM LogEntry
SELECT * FROM AppDomain           -- new in build 1.0.0.8
SELECT * FROM Assembly           -- new in build 1.0.0.8
SELECT * FROM GraphicsProcessor  -- new in build 1.0.0.15
SELECT * FROM Machine            -- new in build 1.0.0.15
SELECT * FROM MachineGraphicsProcessor -- new in build 1.0.0.15
SELECT * FROM MachineProcessor  -- new in build 1.0.0.15
SELECT * FROM Processor          -- new in build 1.0.0.15
SELECT * FROM TargetSite
SELECT * FROM Thread            -- new in build 1.0.0.8
SELECT * FROM WorkItem          -- new in build 1.0.0.60
SELECT * FROM TeamFoundationServer -- new in build 1.0.0.60

```

Listing 2. Simple T-SQL select statements to show the content of the various tables used by the framework.

If all went according to plan then you should see something such as the screenshot of SQL Server 2005 Management Studio found in figure 7 which is on the last page of this document, although possibly containing only a single exception. In build 1.0.0.60, new **WorkItem** and **TeamFoundationServer** tables were added to the database to enable exceptions and log entries to be promoted to Team Foundation Server work items. These tables are merely place holders for future functionality which will be enabled within the forthcoming **Diagnostics Studio** smart-client application.

PublishedException

The **CodePlex.Diagnostics** framework includes three new **Exception** types that can be used to either control the behavior of the framework or add additional meta-data regarding the exceptions that were caught and subsequently published. The first of these new **Exception** types is the **PublishedException** class which is shown along with the **ThreadException** and **UnhandledException** in figure 6.

The code fragment in listing 3 shows how the **DivideByZeroException** can be thrown again as the inner-exception of a **PublishedException**, ensuring that the original exception is published only once by the **ExceptionProvider**. Subsequent calls to the class **ExceptionProvider**, further up the call stack, will ignore an exception that has already been published.

```

catch (DivideByZeroException ex)
{
    IIdentity identity = WindowsIdentity.GetCurrent() as IIdentity;

    Guid publicationId = ExceptionProvider.Publish(ex, identity);

    throw new PublishedException(publicationId, ex);
}

```

Listing 3. Using the **PublishedException** class to indicate an exception has already been published.

ThreadException

In addition to the aforementioned **PublishedException**, the framework also contains the new **ThreadException** class, which indicates that the exception was caught within an event handler for the **ThreadException** event. The **ThreadException** event is located within the **Application** class from the **System.Windows.Forms** namespace.

UnhandledException

Finally, the framework contains the new **UnhandledException** class, which indicates that an exception was caught within an event handler for the **UnhandledException** event. The **ThreadException** event is located within the **AppDomain** class.

```
AppDomain.CurrentDomain.UnhandledException += delegate(object sender,
                                                    UnhandledExceptionEventArgs e)
{
    Exception ex = e.ExceptionObject as Exception;
    IIdentity identity = WindowsIdentity.GetCurrent() as IIdentity;
    ExceptionProvider.Publish(new UnhandledException(ex), identity);
};
```

Listing 4. Using the **UnhandledException** class to indicate an exception was unhandled.

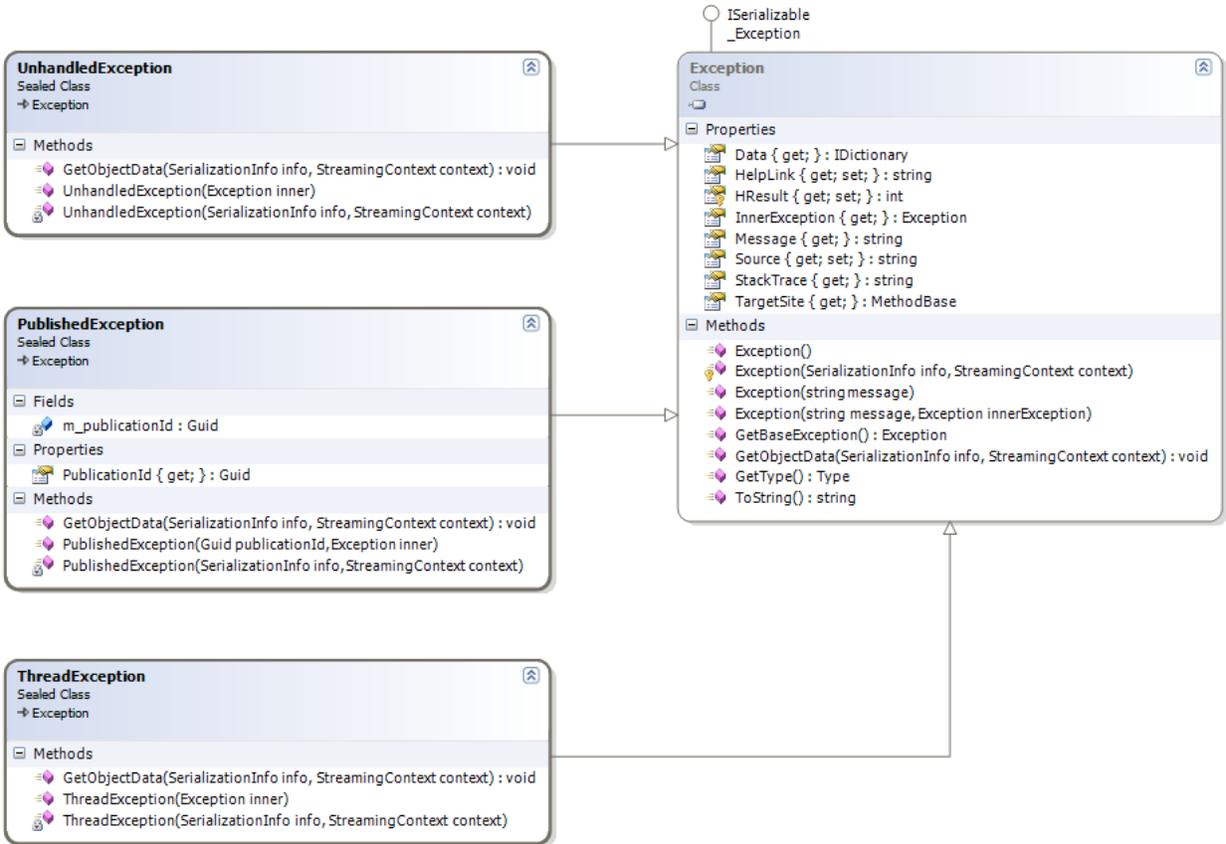


Figure 6. New Exception types contained within the framework.

Limitations

In this release, the internal **ExceptionSerializer** and **IIdentitySerializer** classes, which are responsible for creating the Soap encoded Xml which is stored within the **Exception** and **LogEntry** tables, are unable to serialize objects if either of the following conditions is true:

1. Supplied **Exception** | **IIdentity** is not serializable which is determined by checking the **ISerializable** property of the .NET Framework **Type** class.
2. Supplied **Exception** | **IIdentity** is generic, which determined by checking the **IsGeneric** property of the .NET Framework **Type** class.

In both of these instances, the appropriate column within either the **Exception** or **LogEntry** tables will contain NULL within the **SerializedIIdentityXml** column. In previous releases, any generic **Exception** type would simply not be published and this was fixed in build 1.0.0.60.

The above limitations are in actual fact limitations of the serialization capabilities of the **SoapFormatter** class which is located within the **System.Runtime.Serialization.Formatters.Soap** namespace.

Other serialization alternatives are presently being explored although the **XmlSerializer** class is unable to handle serialization of objects implementing the **IDictionary** interface. When serializing types inheriting from the **System.Exception** base class this is a major problem because the **Data** property is itself an **IDictionary**.

Alternatively, the new serialization capabilities within the .NET Framework 3.0, specifically within the Windows Communication Foundation, require you to specify the known types using the **KnownTypeAttribute**. This also is not appropriate given we cannot at compile time know all of the known types for which serialization may be required.

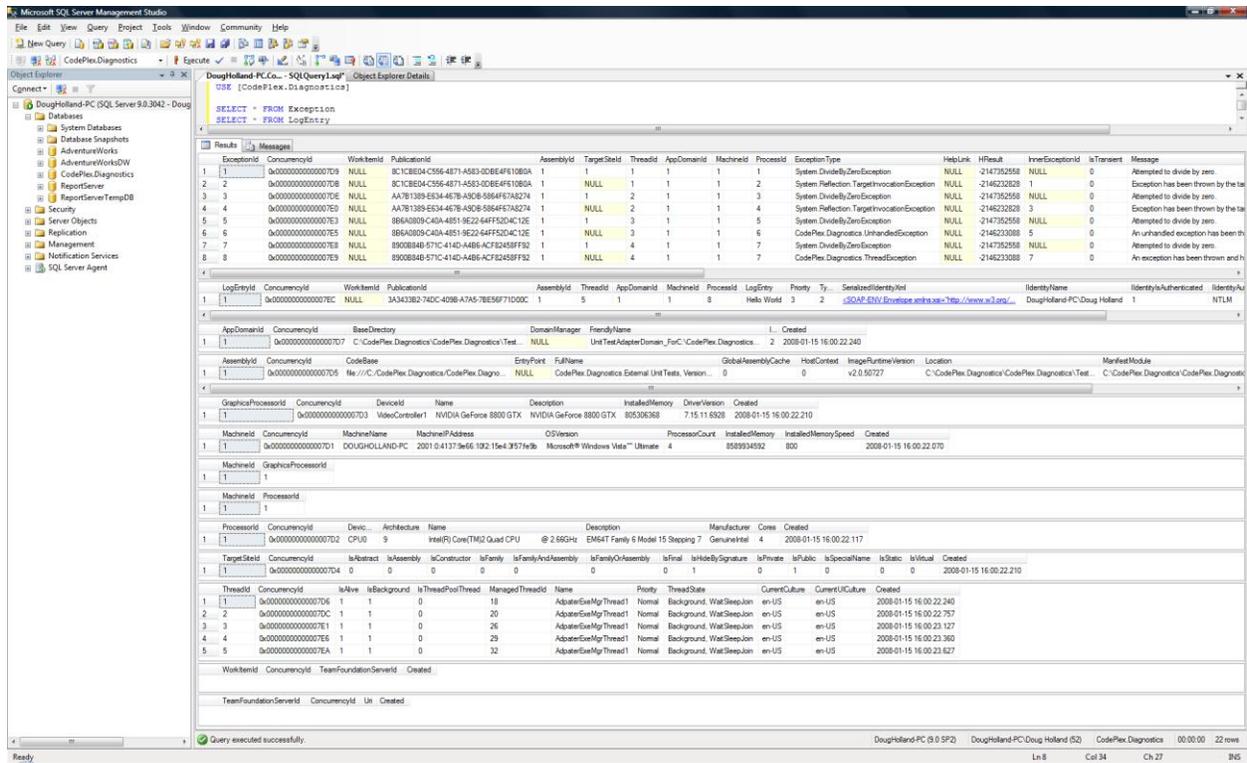


Figure 7. SQL Server 2005 Management Studio showing the result of running the T-SQL in listing 2.