CodePlex.Diagnostics

Version 4.0.0.0

Introduction

Welcome to the fourth major release of the **CodePlex.Diagnostics** framework, designed to provide extensive exception management and logging capabilities for the .net developer.

Using this framework, exceptions and log entries can be written to SQL Server 2008 databases along with extensive contextual information that is often critical to the resolution of exceptions. In this release, the connectivity to the database has been refactored to use the Entity Framework instead of direct ADO.NET. Another major difference is that this release enables true multi-tier architectures by using the Windows Communication Foundation (WCF). With WCF comes additional flexibility in that the specific deployment configuration can be determined on a case by case basis and is easily configured within standard application domain configuration files of the various components within the SOA architecture. (e.g. **App.config**, **Web.config**, and **ServiceReferences.ClientConfig**)

Given the impact of Microsoft's Silverlight technology, the **CodePlex.Diagnostics** framework enables exceptions and log entries, originating from within code executing upon the Silverlight runtime, to be written to the SQL Server 2008 database. Silverlight versions of the **CodePlex.Diagnostics** assemblies are provided for Silverlight developers and these assemblies communicate with the same WCF service layer as the counterparts, targeting the standard Common Language Runtime (CLR), do.

It is important to be aware of some differences between **CodePlex.Diagnostics** assemblies with respect to the Silverlight runtime. The Silverlight runtime imposes security constraints that preclude the gathering of many of the certain system details that are otherwise gathered when exceptions and log entries are written to the database. In this document you will find these differences discussed at some length although generally the usage model of the framework is consistent across the CLR and Silverlight versions.

SQL Server 2005 is no longer supported and therefore either SQL Server 2008 or SQL Server 2008 R2 is required, due to the fact that the database now uses the **datetime2** data type which was not supported in previous releases.

Within the database, columns are included in the **[dbo].[Exception]** table for all of the properties within the .net base **System.Exception** type, aside from the **TargetSite** property which is mapped to the corresponding **[dbo].[TargetSite]** table.

Soap-encoded versions of exceptions are also stored within the database to allow developers access to information residing in fields and properties defined within exception types that derive, directly or indirectly, from the .net base **System.Exception** type. Serialization of exceptions and identities (e.g. types implementing the .net interface **System.Security.Principal.IIdentity**), is achieved using the .net framework's **SoapFormatter** class.

While the **SoapFormatter** class has some well-known limitations, which otherwise would make its use somewhat controversial, the resultant Soap format has the unique characteristic of being largely human readable. The resulting Soap-encoded exceptions and identities are stored within the database using the SQL Server **Xml** column type.

Central to this release of the **CodePlex.Diagnostics** framework is the **ExceptionExtensions** class, shown in the following two Visual Studio 2008 class diagrams, which provide the **Publish** extension method, of which there are four overloads for the .net and Silverlight versions.

Previous releases of the **CodePlex.Diagnostics** framework used the static **ExceptionProvider** class for exception publication; this class still resides within the framework for backwards compatibility purposes. The **[Obsolete]** attribute has been applied to the **ExceptionProvider** class and compiler warnings will result from usage of obsolete types, therefore usage of the **ExceptionProvider** class should be refactored as soon as possible.

Figure 1. **`ExceptionExtensions`** class.

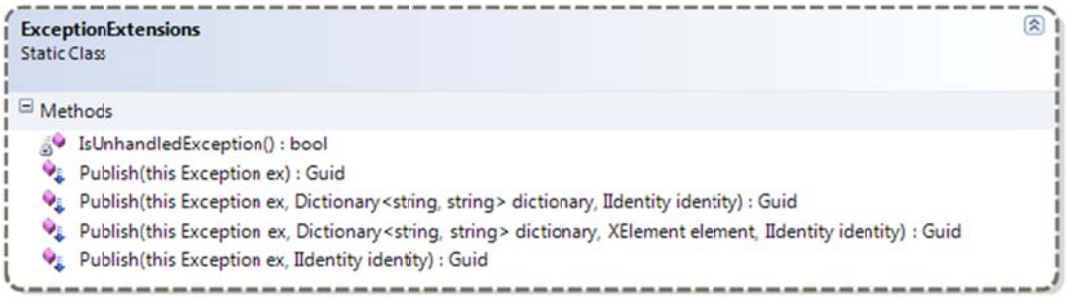

Figure 2. Silverlight version of the **`ExceptionExtensions`** class.

When an instance of a type implementing the **`IIdentity`** interface is not specified, the identity associated with the exception, or log entry, within the database is derived from the current **`WindowsIdentity`**. Given that the Silverlight runtime can be hosted upon operating systems other than Microsoft Windows, the **`WindowsIdentity`** class does not exist within Silverlight's class library; as such an identity must be specified when using the Silverlight version of the framework. Silverlight applications typically use the **`System.Windows.Application`** class to encapsulate the Silverlight application. Using the CodePlex.Diagnostics framework, Silverlight applications should instead use the generic class **`CodePlex.Diagnostics.Silverlight.Application<T>`** shown in figure 3, the type parameter **`T`** allows an **`IPrincipal`** interface to be specified.

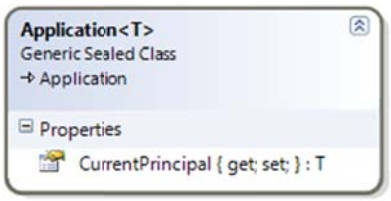

Figure 3. Silverlight **`Application<T>`** class.

The **`Publish`** extension methods return a **`System.Guid`**, or globally unique identifier, called the publication Id and this publication Id is intended to uniquely represent an exception (and any inner-exceptions contained within). An ASP.NET application, for example, could use the publication Id within the URL of pages designed to inform an administrator of the exception that occurred. The publication Id is not used internally within the framework and is merely provided for scenarios such as that described and it can be safely ignored if you have no use for it.

In addition to publishing the exception itself, it is also possible to provide additional contextual information in the form of key value pairs that are contained within an instance of the generic **`Dictionary<string, string>`** class.
As an alternative, or in addition to, providing contextual information within the generic dictionary, contextual information can also be specified within Xml using the **`XElement`** class. The **`XElement`** class is defined within the

System.Xml.Linq namespace and the type was chosen over several alternatives because it is also supported within the Silverlight class libraries.

Another central type within the **CodePlex.Diagnostics** framework is the **StringExtensions** class, shown in the next two Visual Studio 2008 class diagrams, which provides the **Log** extension method, of which there are five overloads for the .net version and four for the Silverlight version.

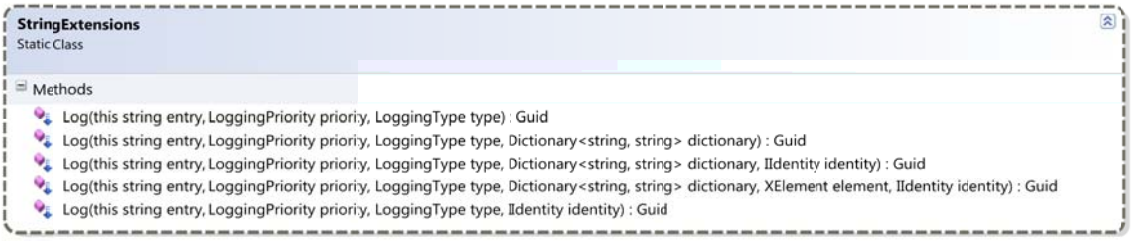

Figure 4. **StringExtensions** class.

Figure 5. Silverlight version of the **StringExtensions** class.

When using any of the overloaded **Log** extension methods, it is necessary to determine the priority of the log entry and the type of information the entry coveys. In order to specify the priority of log entries, the **LoggingPriority** enumeration provides **Low**, **Medium**, **High**, or **Critical** levels of priority or severity.

The **LoggingType** enumeration then enables log entries to be described as **Information**, **Warning** or **Error**.

Figure 6. **LoggingPriority** and **LoggingType** enumerations.

CodePlex.Diagnostics Visual Studio Solution

The **`CodePlex.Diagnostics`** solution contains 13 projects and is shown in figure 7 below.

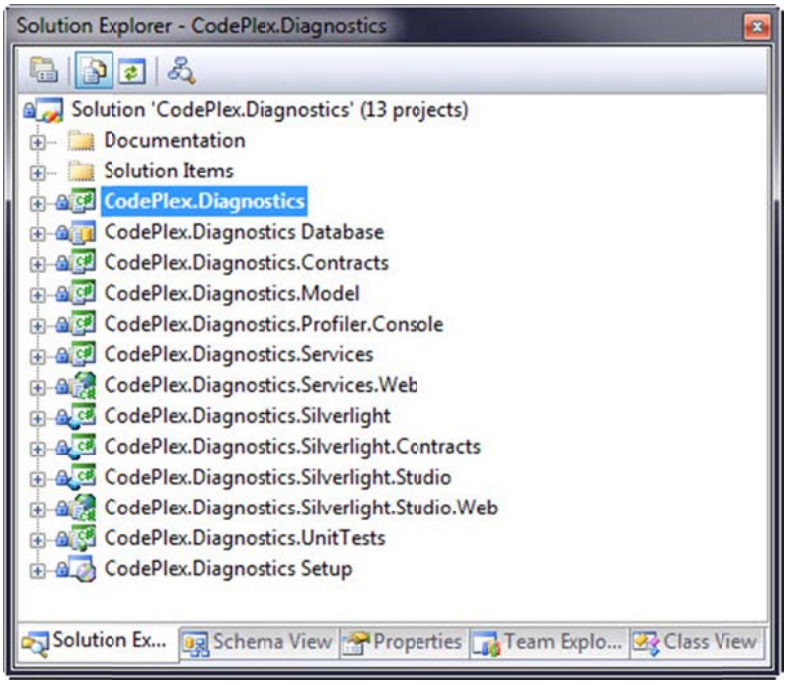

Figure 7. Visual Studio 2008 SP1 Solution

Central to the **`CodePlex.Diagnostics`** framework is the CodePlex.Diagnostics library project that contains the core types of the framework, including the **`ExceptionExtensions`** and **`StringExtentions`** classes.

As discussed earlier within this document, one of the major changes within this version of the framework is the adoption of the Entity Framework which enables some degree of loose coupling between the framework and the underlying database schema design. The Entity Framework model is found within the CodePlex.Diagnostics.Model library project along with several partial declarations that extend the entity classes generated from the model.

Another major change in this version of the framework, also discussed earlier within this document, is the use of the Windows Communication Foundation to enable SOA based architectures to use the framework. Several projects within the solution support this goal.

Data contracts are used to transfer data from the client to the diagnostics service and these contracts are defined within the CodePlex.Diagnostics.Contracts and CodePlex.Diagnostics.Silverlight.Contracts library projects. While the data contracts are structurally very similar between the two libraries there are some differences although these differences are due to limitations in the Silverlight class library. As an example, the **`IExtensibleDataObject`** interface is unavailable within the Silverlight 3 class library although its use within data contracts is regarded as a best practice upon the standard .net CLR.

Service contracts are used to define the diagnostics service and variants of the **`IDiagnosticsService`** interface are found within the CodePlex.Diagnostics, CodePlex.Diagnostics.Silverlight, and CodePlex.Diagnostics.Servics library

projects. Within the client libraries the service is described in terms of asynchronous operation contracts while the service is described purely in terms of synchronous operation contracts within the service library[1].

The CodePlex.Diagnostics.Services.Web project provides an ASP.NET Web application to host the diagnostics service, although the diagnostics service can be hosted based upon the specific needs of your environment. Hosting the services within Internet Information Services (IIS) will limit the communication protocols to those based upon Http and therefore you may choose to use the Windows Activation Service (WAS) upon Windows Server 2008 or Windows Vista and later. Using WAS for service hosting enables the use of Tcp based communication which will also be supported in the forthcoming release of Silverlight 4[2].

Visual Studio Team System 2008 Database Edition, also known as "Data Dude", is used for defining the database and the schema objects of the CodePlex.Diagnostics database can be found within the CodePlex.Diagnostics Database project. See the build requirements section at the end of this document for details about the versions of "Data Dude" to use to build the framework database. SQL scripts will also be provided with the framework release for those who do not have access to an appropriate version of either Visual Studio 2008 or Visual Studio 2010.

The CodePlex.Diagnostics framework includes extensive unit testing based upon the Visual Studio Unit Testing framework and the unit tests for the framework can be found within the CodePlex.Diagnostics.UnitTests project.

Several of the unit tests make use of the commercial mocking framework, Typemock Isolator 2010, and upon release there will also be code available where this dependency has been removed. That said however, it is recommended that the version of the framework that uses Typemock Isolator 2010 be reviewed along with the Typemock Isolator 2010 product itself. Microsoft Research, at the time of writing, is working on an alternative mocking framework, called Moles, and this framework may provide an alternative to Typemock Isolator 2010. At the time of writing no analysis between the two mocking frameworks has been completed although such analysis will determine which mocking framework is used for CodePlex.Diagnostics beyond this release.

Silverlight continues to make a significant impact on the software development world, initially seen as a platform primarily targeted at media applications, in recent months the Silverlight platform has demonstrated potential for business applications.

As discussed earlier, the CodePlex.Diagnostics.Silverlight and CodePlex.Diagnostics.Silverlight.Contracts library projects form the foundation of the framework for the Silverlight developer. In this release of the framework users will be able to use a Silverlight application, CodePlex.Diagnostics.Silverlight.Studio, to examine the exceptions and log entries within one or more instances of the CodePlex.Diagnostics database[3].

Finally, it would not be possible for developers to use the CodePlex.Diagnostics framework without an ability to install the framework. The CodePlex.Diagnostics.Setup project installs the core components, for both the standard CLR and Silverlight, and makes the necessary changes to the Windows registry such that the framework becomes accessible from within Visual Studio 2008 and Visual Studio 2010[4].

Configuration

Configuration of the **`CodePlex.Diagnostics`** framework is achieved using the application domain's configuration file (e.g. **`App.config`** or **`Web.config`**) or the **`ServiceReferences.ClientConfig`** within Silverlight applications. An

---

[1] See *Requirements for an Asynchronous Mechanism in Programming WCF Services, Third Edition by* Juval Löwy.
[2] Silverlight 4 will provide support for Tcp based communication while Silverlight 3 only supports Http communication.
[3] CodePlex.Diagnostics.Silverlight.Studio and CodePlex.Diagnostics.Silverlight.Studio.Web are place holder projects and development of the Diagnostics Studio application will begin after the RC build of the CodePlex.Diagnostics framework has been released.
[4] Registry entries within "HKLM\Software\Wow6432Node\Microsoft\.NET Framework\AssemblyFolders" are made upon 64-bit versions of Windows, upon 32-bit versions of Windows the registry keys are made within "HKLM\Software\Microsoft\.NET Framework\AssemblyFolders".

example of the Xml configuration sections required by the framework is found within the **App.Config** of the framework unit test project **CodePlex.Diagnostics.UnitTests**.

Although the configuration sections are required within the **App.Config** or **Web.Config** for the correct operation of the framework, 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 were to fail the calling code itself will continue unaware of the 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 at least the Professional edition of either Visual Studio 2008 or Visual Studio 2010, it is suggested that you use the unit test project included within the **CodePlex.Diagnostics** solution for verification of any configuration changes that you intend to make. If you have other versions of Visual Studio 2008 or Visual Studio 2010 (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.

Setup

Once you've installed the **CodePlex.Diagnostics** library, using the setup program that is included with the solution, the **CodePlex.Diagnostics** and **CodePlex.Diagnostics.Contracts** assemblies will appear as an option within the Visual Studio 2008 or Visual Studio 2010 "Add Reference" dialog (see figure 8). Silverlight versions of the assemblies will be displayed if the project, for which the reference is required, is targeting the Silverlight runtime.

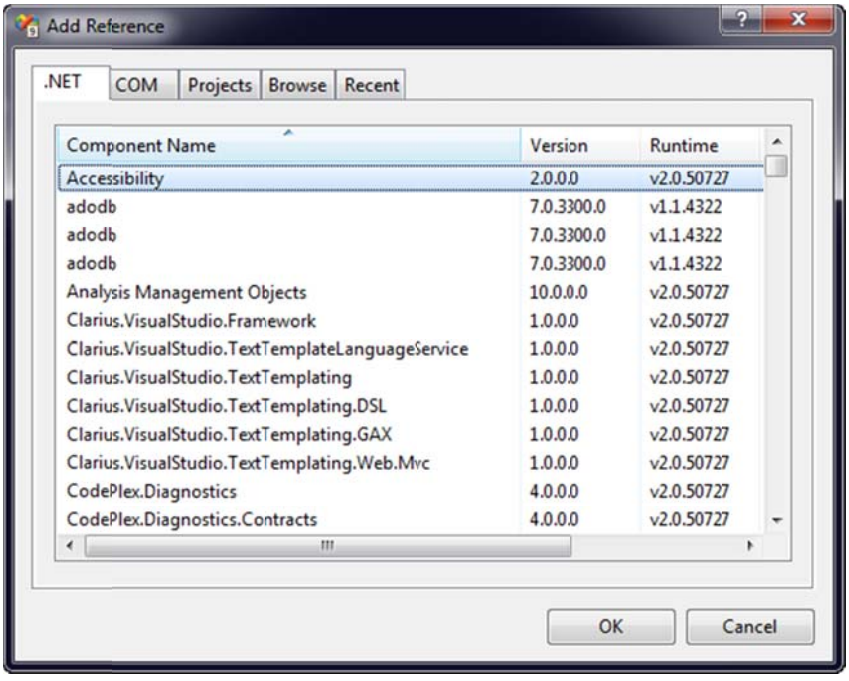

Figure 8. Visual Studio 2008 *Add Reference* dialog showing the **CodePlex.Diagnostics** library.

Create a C# console application and replace the code within the default Program class with that shown below in listing 1, adding references to the **CodePlex.Diagnostics** and **CodePlex.Diagnostics.Contracts** libraries. 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 2008 or SQL Server 2008 R2 Management Studio. Alternatively the "Data Dude" project can be deployed to the local SQL Server 2008 or SQL Server 2008 R2 instance.

```
using System;

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

using CodePlex.Diagnostics;

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)
            {
                ex.Publish();
            }

        }

        #endregion
    }
}
```

Listing 1. Sample C# Console Application to publish an exception using the **`Publish`** extension method defined within the **`ExceptionExtensions`** class.

You'll also need to include an appropriate application domain configuration file, **`App.config`**, which can be based upon that found within the unit testing project **`CodePlex.Diagnostics.UnitTests`**.

```
USE [CodePlex.Diagnostics]

SELECT * FROM [dbo].[AppDomain]
SELECT * FROM [dbo].[Assembly
SELECT * FROM [dbo].[Exception
SELECT * FROM [dbo].[ExceptionInnerException]
SELECT * FROM [dbo].[LogEntry]
SELECT * FROM [dbo].[Machine]
SELECT * FROM [dbo].[Process]
SELECT * FROM [dbo].[SqlError]
SELECT * FROM [dbo].[SqlLog]
SELECT * FROM [dbo].[TargetSite]
SELECT * FROM [dbo].[TeamFoundationServer]
SELECT * FROM [dbo].[Thread]
SELECT * FROM [dbo].[WorkItem]
```

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

Executing the SQL within listing 2 within SQL Server Management Studio will result in results such as those shown within figures 13 and 14. 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`** Silverlight application.

PublishedException

The **`CodePlex.Diagnostics`** framework includes four custom **`Exception`** types that can be used to either control the behavior of the framework or add additional meta-data regarding the inner-exceptions that are contained within them. The first of these custom **`Exception`** types is the **`PublishedException`** class which is shown along with the **`IPublishedException`** interface in figure 9. Silverlight versions of the **`PublishedException`** class and the **`IPublishedException`** interface are shown in figure 10.

The code fragment in listing 3 shows how the **`DivideByZeroException`** can be re-thrown as the inner-exception of the **`PublishedException`** class, ensuring that the original **`DivideByZeroException`** exception is published only once by the **`ExceptionExtensions`** class. Subsequent calls to the **`ExceptionExtensions`** class, further up the call stack, will ignore exceptions that have already been published because the **`PublishedException`** class, among others, implements the **`IPublishedException`** interface.

```
catch (DivideByZeroException ex)
{
    Guid publicationId = ex.Publish();

    throw new PublishedException(publicationId, ex);
}
```

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

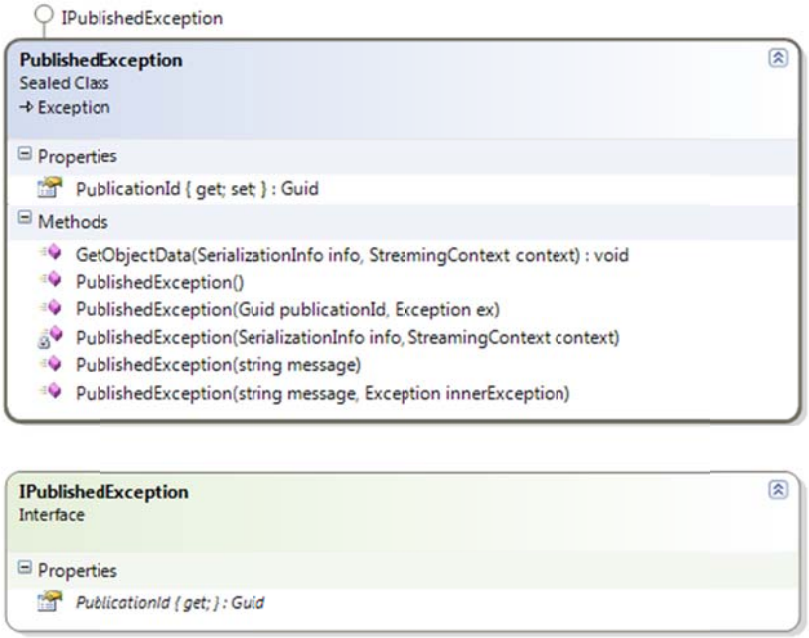

Figure 9. IPublishedException interface and PublishedException class

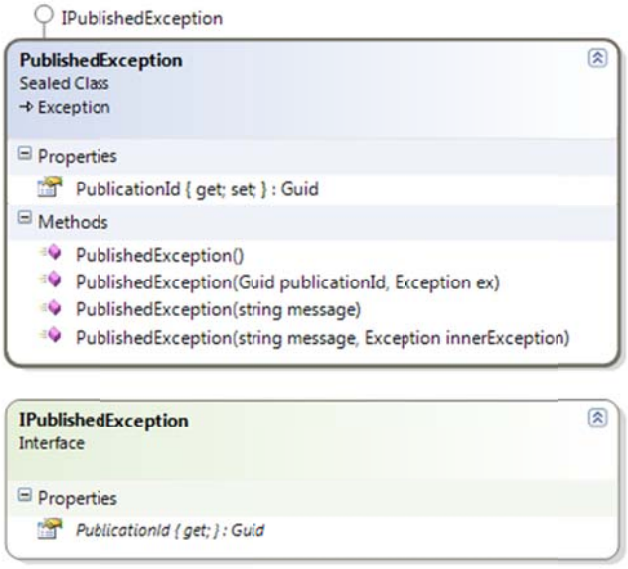

Figure 10. Silverlight version of the IPublishedException interface and PublishedException class

PublishedFaultException

When Microsoft released version 3.0 of the .NET Framework one of the major new additions to the framework was the Windows Communication Foundation (WCF) which provides architects and developers with a framework for building next generation Web services. Within the **`System.ServiceModel`** namespace is the **`FaultException`** type which is the type of exception that clients receive from WCF services. For more information on WCF and exception handling within WCF see Programming WCF Services 3rd Edition by Juval Löwy.

Within version 2.0.0.4 of **`CodePlex.Diagnostics`** the **`PublishedFaultException`** was added to the framework to allow exceptions to be thrown from the service layer indicating to code within the client that the original exception has been successfully published. Also beginning with version 2.0.0.4 the **`IPublishedException`** interface has been added to the framework and can be seen within figure 11 below.

PublishedFaultException<T>

Within the **`System.ServiceModel`** namespace there is also a generic **`FaultException<T>`** which derives from the aforementioned **`FaultException`** type. As such there is also a generic **`PublishedFaultException<T>`** within the CodePlex.Diagnostics framework which derives from the WCF **`FaultException<T>`** class.

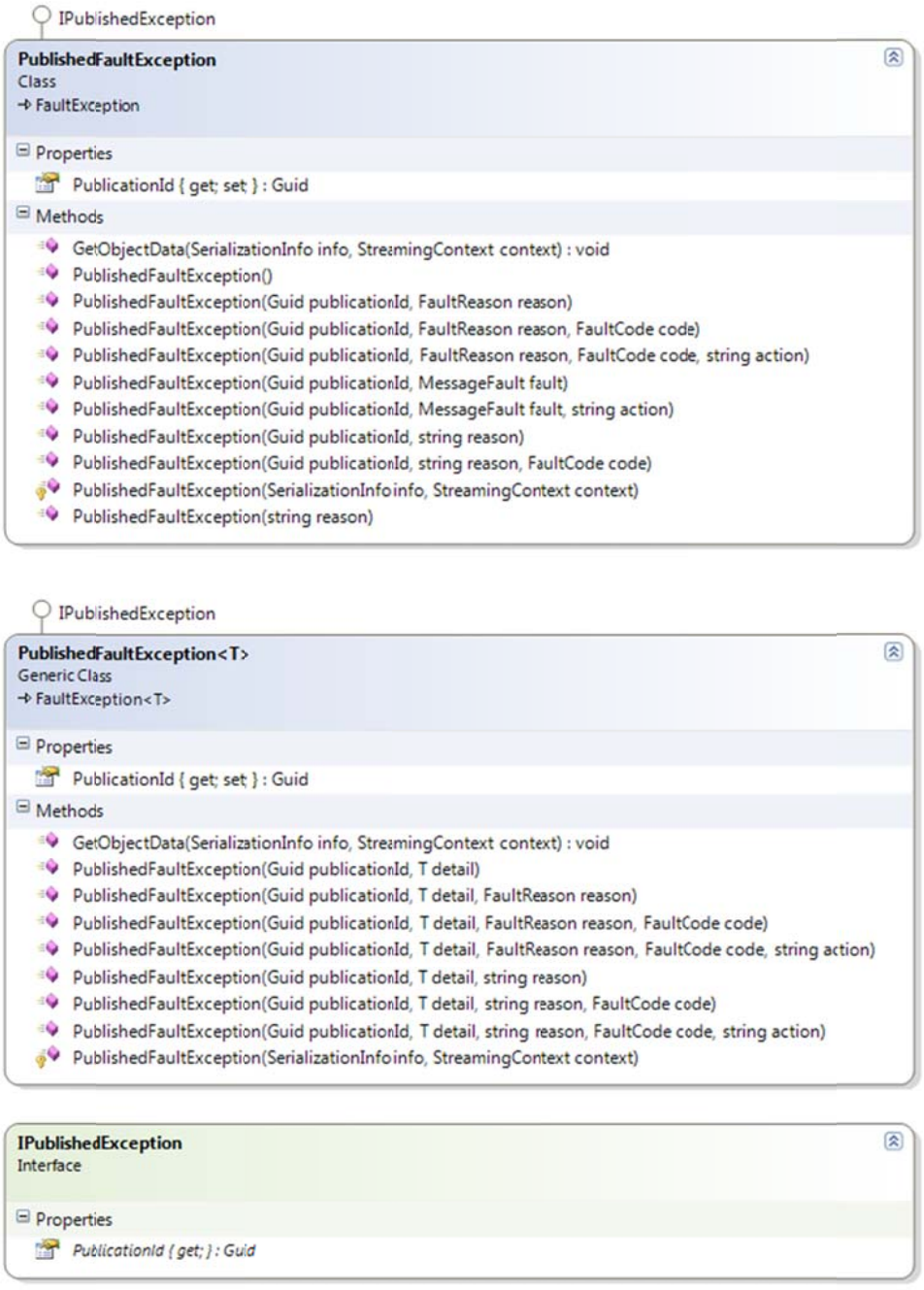

Figure 11. PublishedFaultException and PublishedFaultException<T> classes.

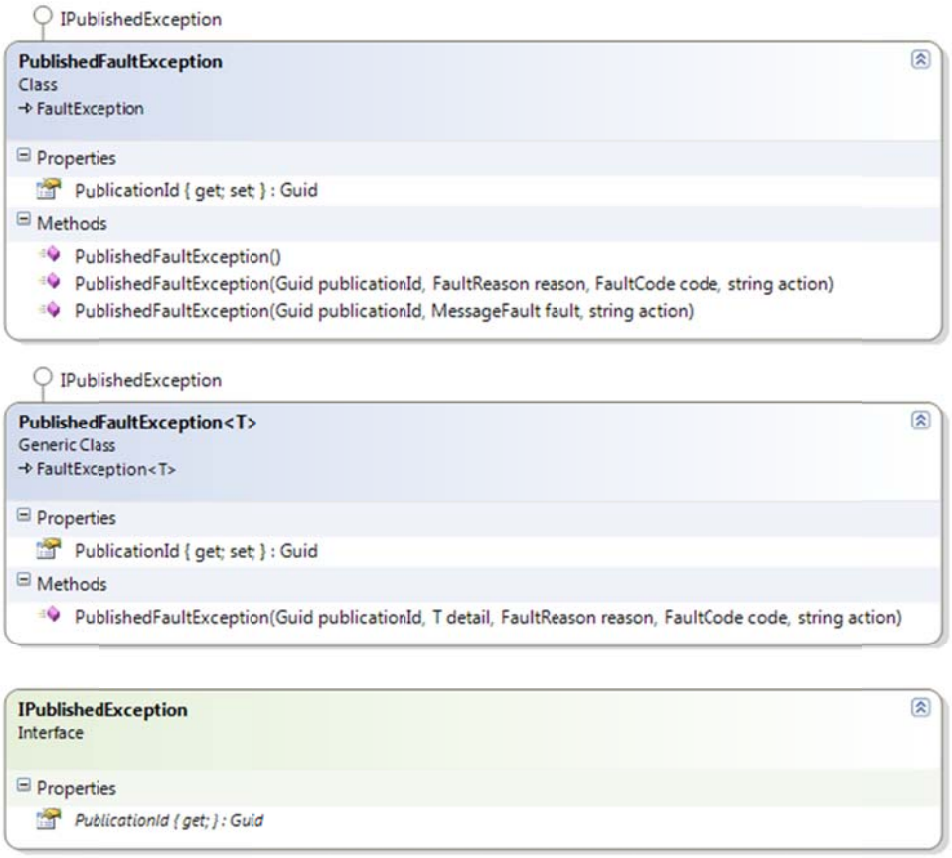

Figure 12. Silverlight versions of the PublishedFaultException and PublishedFaultException<T> classes.

UnhandledException

Earlier versions of the CodePlex.Diagnostics framework contained the **UnhandledException** class which was designed to indicate that the inner-exception was caught within an event handler for the **UnhandledException** event upon the **AppDomain** class.

Listing 4 shows how the **UnhandledException** class was used previously, although in this version of the framework it is now obsolete. It is obsolete due to the fact that the **[dbo].[Exception]** table includes the **Unhandled** column, using the bit data type, to indicate whether an exception was unhandled. Within the **ExceptionExtensions** class there is a private static method that determines whether or not an exception was unhandled and the **Unhandled** column is set accordingly.

```
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.

It may be useful to then explore unhandled exceptions within the database and determine if the exception should have been caught further up the call stack and subsequently published there.

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 **`IsSerializable`** 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.

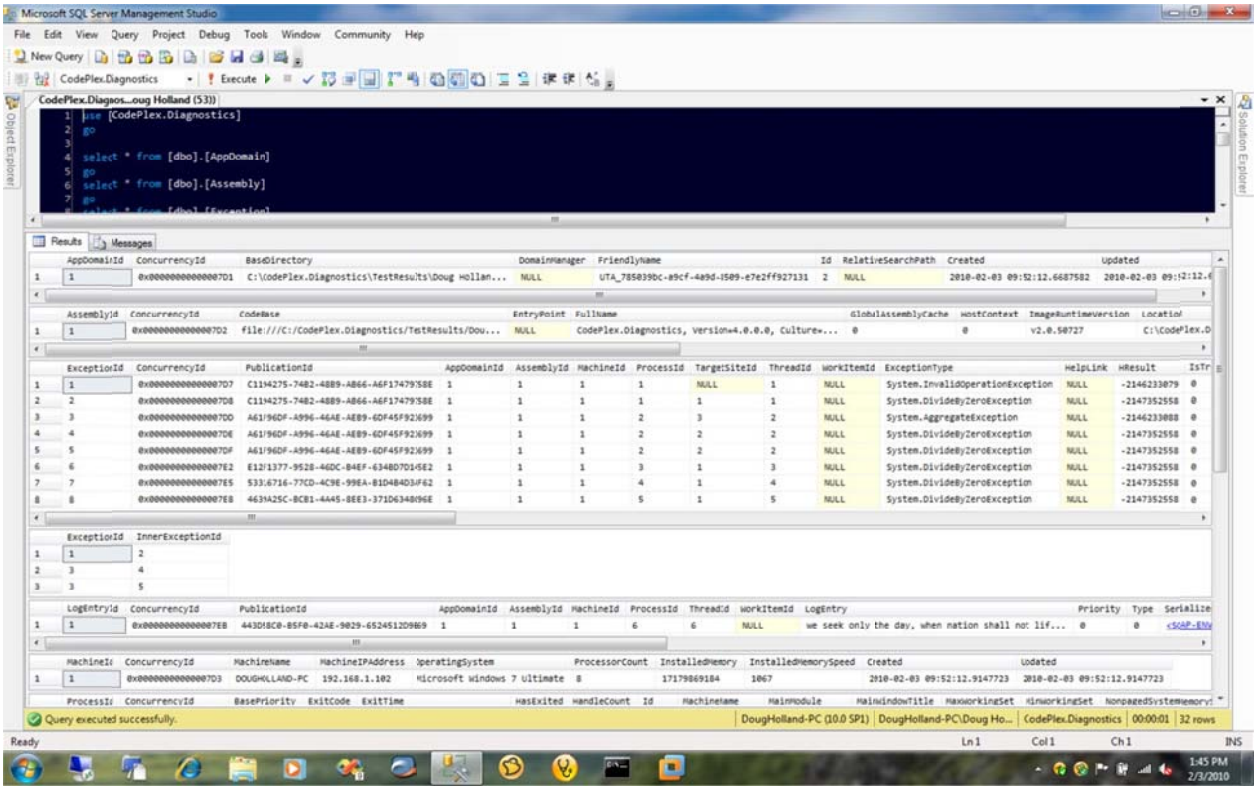

Figure 13. SQL Server 2008 Management Studio showing the result of running the T-SQL in listing 2.

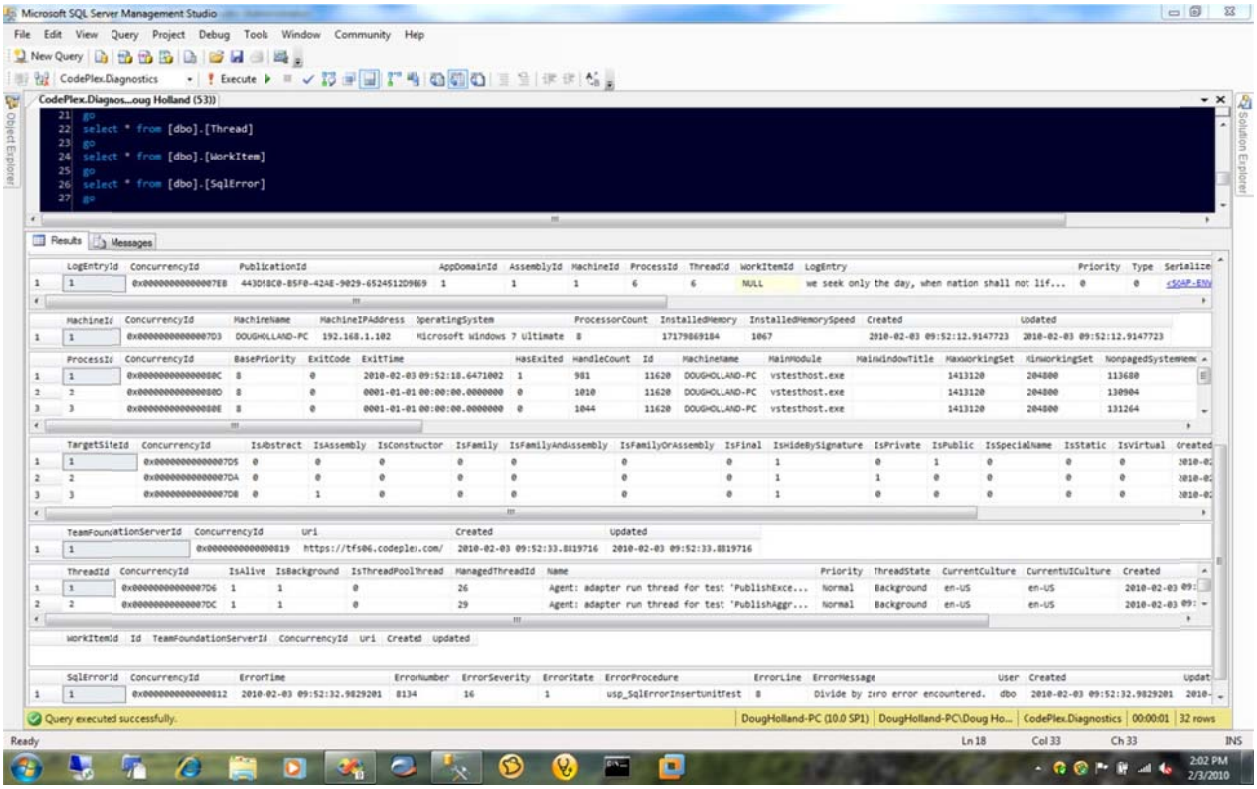

Figure 14. SQL Server 2008 Management Studio showing the result of running the T-SQL in listing 2.

Build Requirements

1. Visual Studio 2010 Ultimate or Visual Studio 2008 Team Suite.
2. Visual Studio Tools for Silverlight 3.0
3. Reactive Extensions for .NET[5]
4. Typemock Isolator 2010[6]

Changes since CodePlex.Diagnostics 2.0.0.4

1. **`ExceptionProvider`** class is now obsolete and has been replaced by the extension methods defined within the **`ExceptionExtensions`** class.
2. **`LoggingProvider`** class is now obsolete and has been replaced by the extension methods defined within the **`StringExtensions`** class.
3. **`SqlExceptionProvider`** class has been replaced by the Entity Framework model contained within the CodePlex.Diagnostics.Model project.

[5] Reactive Extensions are only required when building the CodePlex.Diagnostics framework using Visual Studio 2008 as the framework uses the **`AggregateException`** and **`Parallel`** classes.

[6] Typemock Isolator 2010 is used as a mocking framework within the CodePlex.Diagnostics.UnitTests project although an alternative build of the framework will be available using the Moles framework from Microsoft Research.