**Project Development Phase**

### Debugging & Traceability

|  |  |
| --- | --- |
| **NM ID** | **8073B6FD2C609D2A026419D64F1FD32** |
| **PROJECT TITTLE** | **BUILD IN EVENT MANAGEMENT USING SALESFORCE** |

Tracing is a way for you to monitor the execution of your application while it is running. You can add tracing and debugging instrumentation to your .NET Framework application when you develop it, and you can use that instrumentation both while you are developing the application and after you have deployed it. You can use the [System.Diagnostics.Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace), [System.Diagnostics.Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug), and [System.Diagnostics.TraceSource](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.tracesource) classes to record information about errors and application execution in logs, text files, or other devices for later analysis.

The term instrumentation refers to an ability to monitor or measure the level of a product's performance and to diagnose errors. In programming, this means the ability of an application to incorporate:

* **Code tracing** - Receiving informative messages about the execution of an application at run time.
* **Debugging** - Tracking down and fixing programming errors in an application under development. For more information, see [Debugging](https://learn.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour).
* **Performance counters** - Components that allow you to track the performance of your application. For more information, see [Performance Counters](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/performance-counters).
* **Event logs** - Components that allow you to receive and track major events in the execution of your application. For more information, see the [EventLog](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.eventlog) class.

Instrumenting your application by placing trace statements at strategic locations in your code is especially useful for distributed applications. By using trace statements you can instrument an application not only to display information when things go wrong, but also to monitor how well the application is performing.

The [TraceSource](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.tracesource) class provides enhanced tracing features and can be used in place of the static methods of the older [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) and [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) tracing classes. The familiar [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) and [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) classes are still widely used, but the [TraceSource](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.tracesource) class is recommended for new tracing commands, such as [TraceEvent](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.tracesource.traceevent) and [TraceData](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.tracesource.tracedata).

The [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) and [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) classes are identical, except that procedures and functions of the [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) class are compiled by default into release builds, but those of the [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) class are not.

The [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) and [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) classes provide the means to monitor and examine application performance either during development or after deployment. For example, you can use the [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) class to track particular types of actions in a deployed application as they occur (for example, creation of new database connections), and can therefore monitor the application's efficiency.

## Code Tracing and Debugging

During development, you can use the output methods of the [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) class to display messages in the Output window of the Visual Studio integrated development environment (IDE). For example:

C#Copy

System.Diagnostics.Trace.WriteLine("Hello World!");

System.Diagnostics.Debug.WriteLine("Hello World!");

Each of these examples will display "Hello World!" in the Output window when the application is run in the debugger.

This enables you to debug your applications and optimize their performance based on their behavior in your test environment. You can debug your application in your debug build with the [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) conditional attribute turned on so that you receive all debugging output. When your application is ready for release, you can compile your release build without turning on the [Debug](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.debug) conditional attribute, so that the compiler will not include your debugging code in the final executable. For more information, see [How to: Compile Conditionally with Trace and Debug](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-compile-conditionally-with-trace-and-debug). For more information on different build configurations for your application, see [Compiling and Building](https://learn.microsoft.com/en-us/visualstudio/ide/compiling-and-building-in-visual-studio).

You can also trace code execution in an installed application, using methods of the [Trace](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace) class. By placing [Trace Switches](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/trace-switches) in your code, you can control whether tracing occurs and how extensive it is. This lets you monitor the status of your application in a production environment. This is especially important in a business application that uses multiple components running on multiple computers. You can control how the switches are used after deployment through the configuration file. For more information, see [How to: Create, Initialize and Configure Trace Switches](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-create-initialize-and-configure-trace-switches).

When you are developing an application for which you intend to use tracing, you usually include both tracing and debugging messages in the application code. When you are ready to deploy the application, you can compile your release build without turning on the **Debug** conditional attribute. However, you can turn on the **Trace** conditional attribute so that the compiler includes your trace code in the executable. For more information, see [How to: Compile Conditionally with Trace and Debug](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-compile-conditionally-with-trace-and-debug).

### Phases of Code Tracing

There are three phases of code tracing:

1. **Instrumentation** — you add trace code to your application.
2. **Tracing** — the tracing code writes information to the specified target.
3. **Analysis** — you evaluate the tracing information to identify and understand problems in the application.

During development, all debug and trace output methods write information to the Output window in Visual Studio by default. In a deployed application, the methods write tracing information to the targets you specify. For more information on specifying an output target for tracing or debugging, see [Trace Listeners](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/trace-listeners).

The following is an overall view of the major steps typically involved in using tracing to analyze and correct potential problems in deployed applications. For more information on how to perform these steps, see the appropriate link.

##### To use tracing in an application

1. Consider which tracing output you will want to receive onsite after you have deployed the application.
2. Create a set of switches. For more information, see [How to: Configure Trace Switches](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-create-initialize-and-configure-trace-switches).
3. Add the trace statements to the application code.
4. Determine where you want the tracing output to appear and add the appropriate listeners. For more information, see [Creating and Initializing Trace Listeners](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-create-and-initialize-trace-listeners).
5. Test and debug your application and the tracing code it contains.
6. Compile the application into executable code using one of the following procedures:
   * Use the **Build** menu along with the **Debug** page of the **Property Pages** dialog box in **Solution Explorer**. Use this when compiling in Visual Studio.

- or -

* + Use the **Trace** and **Debug** compiler directives for the command-line method of compiling. For more information, see [Compiling Conditionally with Trace and Debug](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-compile-conditionally-with-trace-and-debug). Use this when compiling from the command line.

1. If a problem occurs during run time, turn on the appropriate trace switch. For more information, see [Configuring Trace Switches](https://learn.microsoft.com/en-us/dotnet/framework/debug-trace-profile/how-to-create-initialize-and-configure-trace-switches).

The tracing code writes tracing messages to a specified target, for example, a screen, a text file, or an event log. The type of listener you included in the [Trace.Listeners](https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.trace.listeners) collection determines the target.

1. Analyze the tracing messages to identify and understand the problem in the application.

## Trace Instrumentation and Distributed Applications

When you create a distributed application, you might find it difficult to test the application in the manner in which it will be used. Few development teams have the capability to test all possible combinations of operating systems or Web browsers (including all the localized language options), or to simulate the high number of users that will access the application at the same time. Under these circumstances, you cannot test how a distributed application will respond to high volumes, different setups, and unique end-user behaviors. Also, many parts of a distributed application have no user interface with which you can interact directly or view the activity of those parts.

However, you can compensate for this by enabling distributed applications to describe certain events of interest to system administrators, especially things that go wrong, by instrumenting the application — that is, by placing trace statements at strategic locations in your code. Then if something unexpected occurs at run time (for example, excessively slow response time), you can determine the likely cause.

With trace statements you can avoid the difficult task of examining the original source code, modifying it, recompiling, and attempting to produce the run-time error within the debugging environment.