Logging and tracing in .NET applications

4 minutes

100 XP

As you continue developing your application and it gets more complex, you'll want to apply additional debug diagnostics to your application.

Tracing is a way for you to monitor your application's execution while it's running. You can add tracing and debugging instrumentation to your .NET application when you develop it. You can use that instrumentation while you're developing the application and after you've deployed it.

This simple technique is surprisingly powerful. You can use it in situations where you need more than a debugger:

- Issues that occur over long periods of time can be difficult to debug with a traditional debugger. Logs allow for detailed post-mortem review that spans long periods of time. In contrast, debuggers are constrained to real-time analysis.
- Multi-threaded applications and distributed applications are often difficult to debug. Attaching a debugger tends to modify behaviors. You can analyze detailed logs as needed to understand complex systems.
- Issues in distributed applications might arise from a complex interaction between many components. It might not be reasonable to connect a debugger to every part of the system.
- Many services shouldn't be stalled. Attaching a debugger often causes timeout failures.
- Issues aren't always foreseen. Logging and tracing are designed for low overhead so that programs can always be recording in case an issue occurs.

Write information to output windows

Up to this point, we've been using the console to display information to the application user. There are other types of applications built with .NET that have user interfaces, such as mobile, web, and desktop apps, and there's no visible console. In these applications, `System.Console` logs messages "behind the scenes." These messages might show up in an output window in Visual Studio or Visual Studio Code. They also might be output to a system log such as Android's `logcat`. As a result, you should take great consideration when you use `System.Console.WriteLine` in a non-console application.

This is where you can use `System.Diagnostics.Debug` and `System.Diagnostics.Trace` in addition to `System.Console`. Both `Debug` and `Trace` are part of `System.Diagnostics` and will only write to logs when an appropriate listener is attached.

The choice of which print style API to use is up to you. The key differences are:

- **System.Console**
  - Always enabled and always writes to the console.
  - Useful for information that your customer might need to see in the release.
  - Because it's the simplest approach, it's often used for ad-hoc temporary debugging. This debug code is often never checked in to source control.
- **System.Diagnostics.Trace**
  - Only enabled when `TRACE` is defined.
  - Writes to attached Listeners, by default, the DefaultTraceListener.
  - Use this API when you create logs that will be enabled in most builds.
- **System.Diagnostics.Debug**
  - Only enabled when `DEBUG` is defined (when in debug mode).
  - Writes to an attached debugger.
  - Use this API when you create logs that will be enabled only in debug builds.

C# Copy

```
Console.WriteLine("This message is readable by the end user.");
Trace.WriteLine("This is a trace message when tracing the app.");
Debug.WriteLine("This is a debug message just for developers.");
```

When you design your tracing and debugging strategy, think about how you want the output to look. Multiple Write statements filled with unrelated information creates a log that's difficult to read. On the other hand, using WriteLine to put related statements on separate lines might make it difficult to distinguish what information belongs together. In general, use multiple Write statements when you want to combine information from multiple sources to create a single informative message. Use the WriteLine statement when you want to create a single complete message.

C# Copy

```
Debug.Write("Debug - ");
Debug.WriteLine("This is a full line.");
Debug.WriteLine("This is another full line.");
```

This output is from the preceding logging with `Debug`:

Output Copy

```
Debug - This is a full line.
This is another full line.
```

Define TRACE and DEBUG constants

By default, when an application is running under debug, the `DEBUG` constant is defined. You can control this by adding a `DefineConstants` entry in the project file in a property group. Here's an example of turning on `TRACE` for both `Debug` and `Release` configurations in addition to `DEBUG` for `Debug` configurations.

XML

Copy

```xml
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
    <DefineConstants>DEBUG;TRACE</DefineConstants>
</PropertyGroup>
<PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Release|AnyCPU'">
    <DefineConstants>TRACE</DefineConstants>
</PropertyGroup>
```

When you use `Trace` when not attached to the debugger, you'll need to configure a trace listener such as dotnet-trace.

Conditional tracing

In addition to simple `Write` and `WriteLine` methods, there's also the capability to add conditions with `WriteIf` and `WriteLineIf`. As an example, the following logic checks if the count is zero and then writes a debug message:

C#

Copy

```csharp
if(count == 0)
{
    Debug.WriteLine("The count is 0 and this may cause an exception.");
}
```

You could rewrite this in a single line of code:

C#

Copy

```csharp
Debug.WriteLineIf(count == 0, "The count is 0 and this may cause an exception.");
```

You can also use these conditions with `Trace` and with flags that you define in your application:

C#

Copy

```csharp
bool errorFlag = false;
System.Diagnostics.Trace.WriteIf(errorFlag, "Error in AppendData procedure.");
System.Diagnostics.Debug.WriteIf(errorFlag, "Transaction abandoned.");
System.Diagnostics.Trace.Write("Invalid value for data request");
```

Verify that certain conditions exist

An assertion, or `Assert` statement, tests a condition that you specify as an argument to the `Assert` statement. If the condition evaluates to `true`, no action occurs. If the condition evaluates to `false`, the assertion fails. If you're running with a debug build, your program enters break mode.

You can use the `Assert` method from either `Debug` or `Trace`, which are in the `System.Diagnostics` namespace. `Debug` class methods aren't included in a release version of your program, so they don't increase the size or reduce the speed of your release code.

Use the `System.Diagnostics.Debug.Assert` method freely to test conditions that should hold true if your code is correct. For example, suppose you've written an integer-divide function. By the rules of mathematics, the divisor can never be zero. You might test this condition by using an assertion:

C#

Copy

```csharp
int IntegerDivide(int dividend, int divisor)
{
    Debug.Assert(divisor != 0, $"{nameof(divisor)} is 0 and will cause an exception.");

    return dividend / divisor;
}
```

When you run this code under the debugger, the assertion statement is evaluated. However, the comparison isn't made in the release version, so there's no additional overhead.

Note

When you use `System.Diagnostics.Debug.Assert`, make sure that any code inside `Assert` doesn't change the results of the program if Assert is removed. Otherwise, you might accidentally introduce a bug that only shows up in the release version of your program. Be especially careful about asserts that contain function or procedure calls.

Using `Debug` and `Trace` from the `System.Diagnostics` namespace is a great way to provide additional context when you run and debug your application.