Create MSBuild inline tasks - MSBuild

By ghogen

Archived: 2026-04-05 20:25:28 UTC

MSBuild tasks are typically created by compiling a class that implements the ITask interface. For more information, see Tasks.

When you want to avoid the overhead of creating a compiled task, you can create a task inline in the project file or in an imported file. You don't have to create a separate assembly to host the task. Using an inline task makes it easier to keep track of source code and easier to deploy the task. The source code is integrated into the MSBuild project file or imported file, typically a `.targets` file.

You create an inline task by using a *code task factory*. For current development, be sure to use RoslynCodeTaskFactory, not `CodeTaskFactory`. `CodeTaskFactory` only supports C# versions up to 4.0.

Inline tasks are intended as a convenience for small tasks that don't require complicated dependencies. Debugging support for inline tasks is limited. It's recommended to create a compiled task instead of inline task when you want to write more complex code, reference a NuGet package, run external tools, or perform operations that could produce error conditions. Also, inline tasks are compiled every time you build, so there can be a noticeable impact on build performance.

The structure of an inline task

An inline task is contained by a UsingTask element. The inline task and the `UsingTask` element that contains it are typically included in a `.targets` file and imported into other project files as required. Here's a basic inline task that does nothing, but illustrates the syntax:

```
<!-- This simple inline task does nothing. -->
<UsingTask
  TaskName="DoNothing"
  TaskFactory="RoslynCodeTaskFactory"
  AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll" >
  <ParameterGroup />
  <Task>
    <Reference Include="" />
    <Using Namespace="" />
    <Code Type="Fragment" Language="cs">
    </Code>
  </Task>
</UsingTask>
```

The `UsingTask` element in the example has three attributes that describe the task and the inline task factory that compiles it.

- The `TaskName` attribute names the task, in this case, `DoNothing`.
- The `TaskFactory` attribute names the class that implements the inline task factory.
- The `AssemblyFile` attribute gives the location of the inline task factory. Alternatively, you can use the `AssemblyName` attribute to specify the fully qualified name of the inline task factory class, which is typically located in `$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll`.

The remaining elements of the `DoNothing` task are empty and are provided to illustrate the order and structure of an inline task. A complete example is presented later in this article.

- The `ParameterGroup` element is optional. When specified, it declares the parameters for the task. For more information about input and output parameters, see Input and output parameters later in this article.
- The `Task` element describes and contains the task source code.
- The `Reference` element specifies references to the .NET assemblies that you are using in your code. Using this element is equivalent to adding a reference to a project in Visual Studio. The `Include` attribute specifies the path of the referenced assembly. Assemblies in mscorlib, .NET Standard, Microsoft.Build.Framework, and Microsoft.Build.Utilities.Core, as well as some assemblies that are transitively referenced as dependencies, are available without a `Reference`.
- The `Using` element lists the namespaces that you want to access. This element is equivalent to the `using` directive in C#. The `Namespace` attribute specifies the namespace to include. It doesn't work to put a `using` directive in the inline code, because that code is put into a method body, where `using` directives aren't allowed.

`Reference` and `Using` elements are language-agnostic. Inline tasks can be written in Visual Basic or C#.

Note

Elements contained by the `Task` element are specific to the task factory, in this case, the code task factory.

Code element

The last child element to appear within the `Task` element is the `Code` element. The `Code` element contains or locates the code that you want to be compiled into a task. What you put in the `Code` element depends on how you want to write the task.

The `Language` attribute specifies the language in which your code is written. Acceptable values are `cs` for C#, `vb` for Visual Basic.

The `Type` attribute specifies the type of code that is found in the `Code` element.

- If the value of `Type` is `Class`, then the `Code` element contains code for a class that derives from the ITask interface.
- If the value of `Type` is `Method`, then the code defines an override of the `Execute` method of the ITask interface.
- If the value of `Type` is `Fragment`, then the code defines the contents of the `Execute` method, but not the signature or the `return` statement.

The code itself typically appears between a `<![CDATA[` marker and a `]]>` marker. Because the code is in a CDATA section, you don't have to worry about escaping reserved characters, for example, "<" or ">".

Alternatively, you can use the `Source` attribute of the `Code` element to specify the location of a file that contains the code for your task. The code in the source file must be of the type that is specified by the `Type` attribute. If the `Source` attribute is present, the default value of `Type` is `Class`. If `Source` isn't present, the default value is `Fragment`.

Note

When defining the task class in the source file, the class name must agree with the `TaskName` attribute of the corresponding UsingTask element.

HelloWorld

Here's an example of a simple inline task. The HelloWorld task displays "Hello, world!" on the default error logging device, which is typically the system console or the Visual Studio **Output** window.

```
 <Project>
  <!-- This simple inline task displays "Hello, world!" -->
  <UsingTask
    TaskName="HelloWorld"
    TaskFactory="RoslynCodeTaskFactory"
    AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll" >
    <ParameterGroup />
    <Task>
      <Using Namespace="System"/>
      <Using Namespace="System.IO"/>
      <Code Type="Fragment" Language="cs">
<![CDATA[
// Display "Hello, world!"
Log.LogError("Hello, world!");
]]>
      </Code>
    </Task>
  </UsingTask>
</Project>
```

You could save the HelloWorld task in a file that is named *HelloWorld.targets*, and then invoke it from a project as follows.

```
 <Project>
  <Import Project="HelloWorld.targets" />
  <Target Name="Hello">
    <HelloWorld />
  </Target>
</Project>
```

Input and output parameters

Inline task parameters are child elements of a `ParameterGroup` element. Every parameter takes the name of the element that defines it. The following code defines the parameter `Text`.

```
 <ParameterGroup>
  <Text />
</ParameterGroup>
```

Parameters may have one or more of these attributes:

- `Required` is an optional attribute that is `false` by default. If `true`, then the parameter is required and must be given a value before calling the task.
- `ParameterType` is an optional attribute that is `System.String` by default. It may be set to any fully qualified type that is either an item or a value that can be converted to and from a string by using ChangeType. (In other words, any type that can be passed to and from an external task.)
- `Output` is an optional attribute that is `false` by default. If `true`, then the parameter must be given a value before returning from the Execute method.

For example,

```
 <ParameterGroup>
  <Expression Required="true" />
  <Files ParameterType="Microsoft.Build.Framework.ITaskItem[]" Required="true" />
  <Tally ParameterType="System.Int32" Output="true" />
</ParameterGroup>
```

defines these three parameters:

- `Expression` is a required input parameter of type System.String.
- `Files` is a required item list input parameter.
- `Tally` is an output parameter of type System.Int32.

If the `Code` element has the `Type` attribute of `Fragment` or `Method`, then properties are automatically created for every parameter. Otherwise, properties must be explicitly declared in the task source code, and must exactly match their parameter definitions.

Debug an inline task

MSBuild generates a source file the inline task and writes the output to text file with a GUID filename in the temporary files folder, *AppData\Local\Temp\MSBuildTemp*. The output is normally deleted, but to preserve this output file, you can set the environment variable `MSBUILDLOGCODETASKFACTORYOUTPUT` to 1.

Example 1

The following inline task replaces every occurrence of a token in the given file with the given value.

```
 <Project>

  <UsingTask TaskName="TokenReplace" TaskFactory="RoslynCodeTaskFactory" AssemblyFile="$(MSBuildToolsPath)\Micr
    <ParameterGroup>
      <Path ParameterType="System.String" Required="true" />
      <Token ParameterType="System.String" Required="true" />
      <Replacement ParameterType="System.String" Required="true" />
    </ParameterGroup>
    <Task>
      <Code Type="Fragment" Language="cs"><![CDATA[
string content = File.ReadAllText(Path);
content = content.Replace(Token, Replacement);
File.WriteAllText(Path, content);

]]></Code>
    </Task>
  </UsingTask>

  <Target Name='Demo' >
    <TokenReplace Path="Target.config" Token="$MyToken$" Replacement="MyValue"/>
  </Target>
</Project>
```

Example 2

The following inline task generates serialized output. This example shows the use of an output parameter and a reference.

```
 <Project>
  <PropertyGroup>
```

```
    <RoslynCodeTaskFactoryAssembly Condition="$(RoslynCodeTaskFactoryAssembly) == ''">$(MSBuildToolsPath)\Micros
  </PropertyGroup>

    <UsingTask
    TaskName="MyInlineTask"
    TaskFactory="RoslynCodeTaskFactory"
    AssemblyFile="$(RoslynCodeTaskFactoryAssembly)">
    <ParameterGroup>
      <Input ParameterType="System.String" Required="true" />
      <Output ParameterType="System.String" Output="true" />
    </ParameterGroup>
    <Task>
      <Reference Include="System.Text.Json" /> <!-- Reference an assembly -->
      <Using Namespace="System.Text.Json" />   <!-- Use a namespace -->
      <Code Type="Fragment" Language="cs">
        <![CDATA[
          Output = JsonSerializer.Serialize(new { Message = Input });
        ]]>
      </Code>
    </Task>
  </UsingTask>

  <Target Name="RunInlineTask">
    <MyInlineTask Input="Hello, Roslyn!" >
      <Output TaskParameter="Output" PropertyName="SerializedOutput" />
    </MyInlineTask>
    <Message Text="Serialized Output: $(SerializedOutput)" />
  </Target>
</Project>
```

Related content

- Tasks
- Walkthrough: Create an inline task
