MSBuild - Visual Studio 2015

By mijacobs

Archived: 2026-04-05 12:46:29 UTC

Note

This article applies to Visual Studio 2015. If you're looking for the latest Visual Studio documentation, see Visual Studio documentation. We recommend upgrading to the latest version of Visual Studio. Download it here

The Microsoft Build Engine is a platform for building applications. This engine, which is also known as MSBuild, provides an XML schema for a project file that controls how the build platform processes and builds software. Visual Studio uses MSBuild, but it doesn't depend on Visual Studio. By invoking msbuild.exe on your project or solution file, you can orchestrate and build products in environments where Visual Studio isn't installed.

Visual Studio uses MSBuild to load and build managed projects. The project files in Visual Studio (.csproj,.vbproj, vcxproj, and others) contain MSBuild XML code that executes when you build a project by using the IDE. Visual Studio projects import all the necessary settings and build processes to do typical development work, but you can extend or modify them from within Visual Studio or by using an XML editor.

For information about MSBuild for C++, see MSBuild (Visual C++).

The following examples illustrate when you might run builds by using an MSBuild command line instead of the Visual Studio IDE.

- Visual Studio isn't installed.
- You want to use the 64-bit version of MSBuild. This version of MSBuild is usually unnecessary, but it allows MSBuild to access more memory.
- You want to run a build in multiple processes. However, you can use the IDE to achieve the same result on projects in C++ and C#.
- You want to modify the build system. For example, you might want to enable the following actions:
    - Preprocess files before they reach the compiler.
    - Copy the build outputs to a different place.
    - Create compressed files from build outputs.
    - Do a post-processing step. For example, you might want to stamp an assembly with a different version.

  You can write code in the Visual Studio IDE but run builds by using MSBuild. As another alternative, you can build code in the IDE on a development computer but use an MSBuild command line to build code

that's integrated from multiple developers.

Note

You can use Team Foundation Build to automatically compile, test, and deploy your application. Your build system can automatically run builds when developers check in code (for example, as part of a Continuous Integration strategy) or according to a schedule (for example, a nightly Build Verification Test build). Team Foundation Build compiles your code by using MSBuild. For more information, see Build the application.

This topic provides an overview of MSBuild. For an introductory tutorial, see Walkthrough: Using MSBuild.

In this topic

Using MSBuild at a Command Prompt

To run MSBuild at a command prompt, pass a project file to MSBuild.exe, together with the appropriate command-line options. Command-line options let you set properties, execute specific targets, and set other options that control the build process. For example, you would use the following command-line syntax to build the file `MyProj.proj` with the `Configuration` property set to `Debug`.

```
MSBuild.exe MyProj.proj /property:Configuration=Debug
```

For more information about MSBuild command-line options, see Command-Line Reference.

Important

Before you download a project, determine the trustworthiness of the code.

Project File

MSBuild uses an XML-based project file format that's straightforward and extensible. The MSBuild project file format lets developers describe the items that are to be built, and also how they are to be built for different operating systems and configurations. In addition, the project file format lets developers author reusable build rules that can be factored into separate files so that builds can be performed consistently across different projects in the product.

The following sections describe some of the basic elements of the MSBuild project file format. For a tutorial about how to create a basic project file, see Walkthrough: Creating an MSBuild Project File from Scratch.

Properties

Properties represent key/value pairs that can be used to configure builds. Properties are declared by creating an element that has the name of the property as a child of a PropertyGroup element. For example, the following code creates a property named `BuildDir` that has a value of `Build`.

```
 <PropertyGroup>
    <BuildDir>Build</BuildDir>
</PropertyGroup>
```

You can define a property conditionally by placing a `Condition` attribute in the element. The contents of conditional elements are ignored unless the condition evaluates to `true`. In the following example, the `Configuration` element is defined if it hasn't yet been defined.

```
 <Configuration  Condition=" '$(Configuration)' == '' ">Debug</Configuration>
```

Properties can be referenced throughout the project file by using the syntax $(*PropertyName*). For example, you can reference the properties in the previous examples by using `$(BuildDir)` and `$(Configuration)`.

For more information about properties, see MSBuild Properties.

Items

Items are inputs into the build system and typically represent files. Items are grouped into item types, based on user-defined item names. These item types can be used as parameters for tasks, which use the individual items to perform the steps of the build process.

Items are declared in the project file by creating an element that has the name of the item type as a child of an ItemGroup element. For example, the following code creates an item type named `Compile`, which includes two files.

```
 <ItemGroup>
    <Compile Include = "file1.cs"/>
    <Compile Include = "file2.cs"/>
</ItemGroup>
```

Item types can be referenced throughout the project file by using the syntax @(*ItemType*). For example, the item type in the example would be referenced by using `@(Compile)`.

In MSBuild, element and attribute names are case-sensitive. However, property, item, and metadata names are not. The following example creates the item type `Compile`, `comPile`, or any other case variation, and gives the item type the value "one.cs;two.cs".

```
 <ItemGroup>
  <Compile Include="one.cs" />
  <comPile Include="two.cs" />
</ItemGroup>
```

Items can be declared by using wildcard characters and may contain additional metadata for more advanced build scenarios. For more information about items, see Items.

Tasks

Tasks are units of executable code that MSBuild projects use to perform build operations. For example, a task might compile input files or run an external tool. Tasks can be reused, and they can be shared by different developers in different projects.

The execution logic of a task is written in managed code and mapped to MSBuild by using the UsingTask element. You can write your own task by authoring a managed type that implements the ITask interface. For more information about how to write tasks, see Task Writing.

MSBuild includes common tasks that you can modify to suit your requirements. Examples are Copy, which copies files, MakeDir, which creates directories, and Csc, which compiles Visual C# source code files. For a list of available tasks together with usage information, see Task Reference.

A task is executed in an MSBuild project file by creating an element that has the name of the task as a child of a Target element. Tasks typically accept parameters, which are passed as attributes of the element. Both MSBuild properties and items can be used as parameters. For example, the following code calls the MakeDir task and passes it the value of the `BuildDir` property that was declared in the earlier example.

```
 <Target Name="MakeBuildDirectory">
    <MakeDir  Directories="$(BuildDir)" />
</Target>
```

For more information about tasks, see Tasks.

Targets

Targets group tasks together in a particular order and expose sections of the project file as entry points into the build process. Targets are often grouped into logical sections to increase readability and to allow for expansion. Breaking the build steps into targets lets you call one piece of the build process from other targets without copying

that section of code into every target. For example, if several entry points into the build process require references to be built, you can create a target that builds references and then run that target from every entry point where it's required.

Targets are declared in the project file by using the Target element. For example, the following code creates a target named `Compile`, which then calls the Csc task that has the item list that was declared in the earlier example.

```
 <Target Name="Compile">
    <Csc Sources="@(Compile)" />
</Target>
```

In more advanced scenarios, targets can be used to describe relationships among one another and perform dependency analysis so that whole sections of the build process can be skipped if that target is up-to-date. For more information about targets, see Targets.

Build Logs

You can log build errors, warnings, and messages to the console or another output device. For more information, see Obtaining Build Logs and Logging in MSBuild.

Using MSBuild in Visual Studio

Visual Studio uses the MSBuild project file format to store build information about managed projects. Project settings that are added or changed by using the Visual Studio interface are reflected in the .*proj file that's generated for every project. Visual Studio uses a hosted instance of MSBuild to build managed projects. This means that a managed project can be built in Visual Studio or at a command prompt (even if Visual Studio isn't installed), and the results will be identical.

For a tutorial about how to use MSBuild in Visual Studio, see Walkthrough: Using MSBuild.

Multitargeting

By using Visual Studio, you can compile an application to run on any one of several versions of the .NET Framework. For example, you can compile an application to run on the .NET Framework 2.0 on a 32-bit platform, and you can compile the same application to run on the .NET Framework 4.5 on a 64-bit platform. The ability to compile to more than one framework is named multitargeting.

These are some of the benefits of multitargeting:

- You can develop applications that target earlier versions of the .NET Framework, for example, versions 2.0, 3.0, and 3.5.
- You can target frameworks other than the .NET Framework, for example, Silverlight.
- You can target a *framework profile*, which is a predefined subset of a target framework.

- If a service pack for the current version of the .NET Framework is released, you could target it.
- Multitargeting guarantees that an application uses only the functionality that's available in the target framework and platform.

  For more information, see Multitargeting.

Title	Description
Walkthrough: Creating an MSBuild Project File from Scratch	Shows how to create a basic project file incrementally, by using only a text editor.
Walkthrough: Using MSBuild	Introduces the building blocks of MSBuild and shows how to write, manipulate, and debug MSBuild projects without closing the Visual Studio IDE.
MSBuild Concepts	Presents the four building blocks of MSBuild: properties, items, targets, and tasks.
Items	Describes the general concepts behind the MSBuild file format and how the pieces fit together.
MSBuild Properties	Introduces properties and property collections. Properties are key/value pairs that can be used to configure builds.
Targets	Explains how to group tasks together in a particular order and enable sections of the build process to be called on the command line.
Tasks	Shows how to create a unit of executable code that can be used by MSBuild to perform atomic build operations.
Conditions	Discusses how to use the `Condition` attribute in an MSBuild element.
Advanced Concepts	Presents batching, performing transforms, multitargeting, and other advanced techniques.
Logging in MSBuild	Describes how to log build events, messages, and errors.
Additional Resources	Lists community and support resources for more information about MSBuild.

Reference

MSBuild Reference
Links to topics that contain reference information.

Glossary

Defines common MSBuild terms.
