Walkthrough: Create an MSBuild project file from scratch

6/24/2022 • 10 minutes to read • Edit Online

Programming languages that target the .NET Framework use MSBuild project files to describe and control the application build process. When you use Visual Studio to create an MSBuild project file, the appropriate XML is added to the file automatically. However, you may find it helpful to understand how the XML is organized and how you can change it to control a build.

> **NOTE**
> This tutorial works only with .NET Framework 4.x and earlier, not .NET Core or .NET 5 and later.

For information about creating a project file for a C++ project, see MSBuild (C++).

This walkthrough shows how to create a basic project file incrementally, by using only a text editor. The walkthrough follows these steps:

1. Extend the PATH environment variable.
2. Create a minimal application source file.
3. Create a minimal MSBuild project file.
4. Build the application by using the project file.
5. Add properties to control the build.
6. Control the build by changing property values.
7. Add targets to the build.
8. Control the build by specifying targets.
9. Build incrementally.

This walkthrough shows how to build the project at the command prompt and examine the results. For more information about MSBuild and how to run MSBuild at the command prompt, see Walkthrough: Use MSBuild.

To complete the walkthrough, you must have Visual Studio installed because it includes MSBuild and the Visual C# compiler, which are required for the walkthrough.

Extend the path

Before you can use MSBuild, you must extend the PATH environment variable to include all the required tools. You can use the **Developer Command Prompt for Visual Studio**. Search for it on Windows 10 in the search box in the Windows task bar. To set up the environment in an ordinary command prompt or in a scripting environment, run *VSDevCmd.bat* in the *Common7/Tools* subfolder of a Visual Studio installation.

Create a minimal application

This section shows how to create a minimal C# application source file by using a text editor.

1. At the command prompt, browse to the folder where you want to create the application, for example, *\My Documents* or *\Desktop*.
2. Type **md HelloWorld** to create a subfolder named *\HelloWorld*.
3. Type **cd HelloWorld** to change to the new folder.
4. Start Notepad or another text editor, and then type the following code.

```
using System;

class HelloWorld
{
    static void Main()
    {
#if DebugConfig
        Console.WriteLine("WE ARE IN THE DEBUG CONFIGURATION");
#endif

        Console.WriteLine("Hello, world!");
    }
}
```

5. Save this source code file and name it *Helloworld.cs*.
6. Build the application by typing **csc helloworld.cs** at the command prompt.
7. Test the application by typing **helloworld** at the command prompt.

   The **Hello, world!** message should be displayed.
8. Delete the application by typing **del helloworld.exe** at the command prompt.

Create a minimal MSBuild project file

Now that you have a minimal application source file, you can create a minimal project file to build the application. This project file contains the following elements:

- The required root `Project` node.
- An `ItemGroup` node to contain item elements.
- An item element that refers to the application source file.
- A `Target` node to contain tasks that are required to build the application.
- A `Task` element to start the Visual C# compiler to build the application.

To create a minimal MSBuild project file

1. In the text editor, create a new file and enter these two lines:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
</Project>
```

2. Insert this `ItemGroup` node as a child element of the `Project` node:

```
<ItemGroup>
  <Compile Include="helloworld.cs" />
</ItemGroup>
```

Notice that this `ItemGroup` already contains an item element.

3. Add a `Target` node as a child element of the `Project` node. Name the node `Build`.

```
<Target Name="Build">
</Target>
```

4. Insert this task element as a child element of the `Target` node:

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

5. Save this project file and name it *Helloworld.csproj*.

Your minimal project file should resemble the following code:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <Csc Sources="@(Compile)"/>
  </Target>
</Project>
```

Tasks in the Build target are executed sequentially. In this case, the Visual C# compiler `Csc` task is the only task. It expects a list of source files to compile, and this is given by the value of the `Compile` item. The `Compile` item references just one source file, *Helloworld.cs*.

NOTE

In the item element, you can use the asterisk wildcard character (*) to reference all files that have the *.cs* file name extension, as follows:

```
<Compile Include="*.cs" />
```

Build the application

Now, to build the application, use the project file that you just created.

1. At the command prompt, type **msbuild helloworld.csproj -t:Build**.

   This builds the Build target of the Helloworld project file by invoking the Visual C# compiler to create the Helloworld application.

2. Test the application by typing **helloworld**.

   The **Hello, world!** message should be displayed.

NOTE

You can see more details about the build by increasing the verbosity level. To set the verbosity level to "detailed", type this command at the command prompt:

msbuild helloworld.csproj -t:Build -verbosity:detailed

Add build properties

You can add build properties to the project file to further control the build. Now add these properties:

- An `AssemblyName` property to specify the name of the application.
- An `OutputPath` property to specify a folder to contain the application.

To add build properties

1. Delete the existing application by typing **del helloworld.exe** at the command prompt.
2. In the project file, insert this `PropertyGroup` element just after the opening `Project` element:

   ```
   <PropertyGroup>
     <AssemblyName>MSBuildSample</AssemblyName>
     <OutputPath>Bin\</OutputPath>
   </PropertyGroup>
   ```

3. Add this task to the Build target, just before the `Csc` task:

   ```
   <MakeDir Directories="$(OutputPath)"      Condition="!Exists('$(OutputPath)')" />
   ```

   The `MakeDir` task creates a folder that is named by the `OutputPath` property, provided that no folder by that name currently exists.

4. Add this `OutputAssembly` attribute to the `Csc` task:

   ```
   <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
   ```

   This instructs the Visual C# compiler to produce an assembly that is named by the `AssemblyName` property and to put it in the folder that is named by the `OutputPath` property.

5. Save your changes.

Your project file should now resemble the following code:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AssemblyName>MSBuildSample</AssemblyName>
    <OutputPath>Bin\</OutputPath>
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <MakeDir Directories="$(OutputPath)" Condition="!Exists('$(OutputPath)')" />
    <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
  </Target>
</Project>
```

NOTE

We recommend that you add the backslash (\) path delimiter at the end of the folder name when you specify it in the `OutputPath` element, instead of adding it in the `OutputAssembly` attribute of the `Csc` task. Therefore,

```
<OutputPath>Bin\</OutputPath>
```

```
OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
```

is better than

```
<OutputPath>Bin</OutputPath>
```

```
OutputAssembly="$(OutputPath)\$(AssemblyName).exe" />
```

Test the build properties

Now you can build the application by using the project file in which you used build properties to specify the output folder and application name.

1. At the command prompt, type **msbuild helloworld.csproj -t:Build**.

   This creates the *\Bin* folder and then invokes the Visual C# compiler to create the *MSBuildSample* application and puts it in the *\Bin* folder.

2. To verify that the *\Bin* folder has been created, and that it contains the *MSBuildSample* application, type **dir Bin**.

3. Test the application by typing **Bin\MSBuildSample**.

   The **Hello, world!** message should be displayed.

Add build targets

Next, add two more targets to the project file, as follows:

- A Clean target that deletes old files.
- A Rebuild target that uses the `DependsOnTargets` attribute to force the Clean task to run before the Build task.

Now that you have multiple targets, you can set the Build target as the default target.

To add build targets

1. In the project file, add these two targets just after the Build target:

   ```
   <Target Name="Clean" >
     <Delete Files="$(OutputPath)$(AssemblyName).exe" />
   </Target>
   <Target Name="Rebuild" DependsOnTargets="Clean;Build" />
   ```

   The Clean target invokes the Delete task to delete the application. The Rebuild target does not run until both the Clean target and the Build target have run. Although the Rebuild target has no tasks, it causes the Clean target to run before the Build target.

2. Add this `DefaultTargets` attribute to the opening `Project` element:

   ```
   <Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   ```

This sets the Build target as the default target.

Your project file should now resemble the following code:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AssemblyName>MSBuildSample</AssemblyName>
    <OutputPath>Bin\</OutputPath>
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <MakeDir Directories="$(OutputPath)" Condition="!Exists('$(OutputPath)')" />
    <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
  </Target>
  <Target Name="Clean" >
    <Delete Files="$(OutputPath)$(AssemblyName).exe" />
  </Target>
  <Target Name="Rebuild" DependsOnTargets="Clean;Build" />
</Project>
```

Test the build targets

You can exercise the new build targets to test these features of the project file:

- Building the default build.
- Setting the application name at the command prompt.
- Deleting the application before another application is built.
- Deleting the application without building another application.

To test the build targets

1. At the command prompt, type **msbuild helloworld.csproj -p:AssemblyName=Greetings**.

   Because you did not use the **-t** switch to explicitly set the target, MSBuild runs the default Build target. The **-p** switch overrides the `AssemblyName` property and gives it the new value, `Greetings`. This causes a new application, *Greetings.exe*, to be created in the *\Bin* folder.

2. To verify that the *\Bin* folder contains both the *MSBuildSample* application and the new *Greetings* application, type **dir Bin**.

3. Test the Greetings application by typing **Bin\Greetings**.

   The **Hello, world!** message should be displayed.

4. Delete the MSBuildSample application by typing **msbuild helloworld.csproj -t:clean**.

   This runs the Clean task to remove the application that has the default `AssemblyName` property value, `MSBuildSample`.

5. Delete the Greetings application by typing **msbuild helloworld.csproj -t:clean -p:AssemblyName=Greetings**.

   This runs the Clean task to remove the application that has the given **AssemblyName** property value, `Greetings`.

6. To verify that the *\Bin* folder is now empty, type **dir Bin**.

7. Type **msbuild**.

Although a project file is not specified, MSBuild builds the *helloworld.csproj* file because there is only one project file in the current folder. This causes the *MSBuildSample* application to be created in the *\Bin* folder.

To verify that the *\Bin* folder contains the *MSBuildSample* application, type **dir Bin**.

Build incrementally

You can tell MSBuild to build a target only if the source files or target files that the target depends on have changed. MSBuild uses the time stamp of a file to determine whether it has changed.

To build incrementally

1. In the project file, add these attributes to the opening Build target:

   ```
   Inputs="@(Compile)" Outputs="$(OutputPath)$(AssemblyName).exe"
   ```

   This specifies that the Build target depends on the input files that are specified in the `Compile` item group, and that the output target is the application file.

   The resulting Build target should resemble the following code:

   ```
   <Target Name="Build" Inputs="@(Compile)" Outputs="$(OutputPath)$(AssemblyName).exe">
     <MakeDir Directories="$(OutputPath)" Condition="!Exists('$(OutputPath)')" />
     <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
   </Target>
   ```

2. Test the Build target by typing **msbuild -v:d** at the command prompt.

   Remember that *helloworld.csproj* is the default project file, and that Build is the default target.

   The **-v:d** switch specifies a verbose description for the build process.

   These lines should be displayed:

   Skipping target "Build" because all output files are up-to-date with respect to the input files.

   Input files: HelloWorld.cs

   Output files: BinMSBuildSample.exe

   MSBuild skips the Build target because none of the source files have changed since the application was last built.

C# example

The following example shows a project file that compiles a C# application and logs a message that contains the output file name.

Code

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>HelloWorldCS</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <CSFile Include = "consolehwcs1.cs"/>
    </ItemGroup>

    <Target Name = "Compile">
        <!-- Run the Visual C# compilation using input files of type CSFile -->
        <CSC
            Sources = "@(CSFile)"
            OutputAssembly = "$(appname).exe">
            <!-- Set the OutputAssembly attribute of the CSC task
            to the name of the executable file that is created -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </CSC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

Visual Basic example

The following example shows a project file that compiles a Visual Basic application and logs a message that contains the output file name.

Code

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>HelloWorldVB</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <VBFile Include = "consolehwvb1.vb"/>
    </ItemGroup>

    <Target Name = "Compile">
        <!-- Run the Visual Basic compilation using input files of type VBFile -->
        <VBC
            Sources = "@(VBFile)"
            OutputAssembly= "$(appname).exe">
            <!-- Set the OutputAssembly attribute of the VBC task
            to the name of the executable file that is created -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </VBC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

What's next?

Visual Studio can automatically do much of the work that is shown in this walkthrough. To learn how to use Visual Studio to create, edit, build, and test MSBuild project files, see Walkthrough: Use MSBuild.

See also

- MSBuild overview
- MSBuild reference